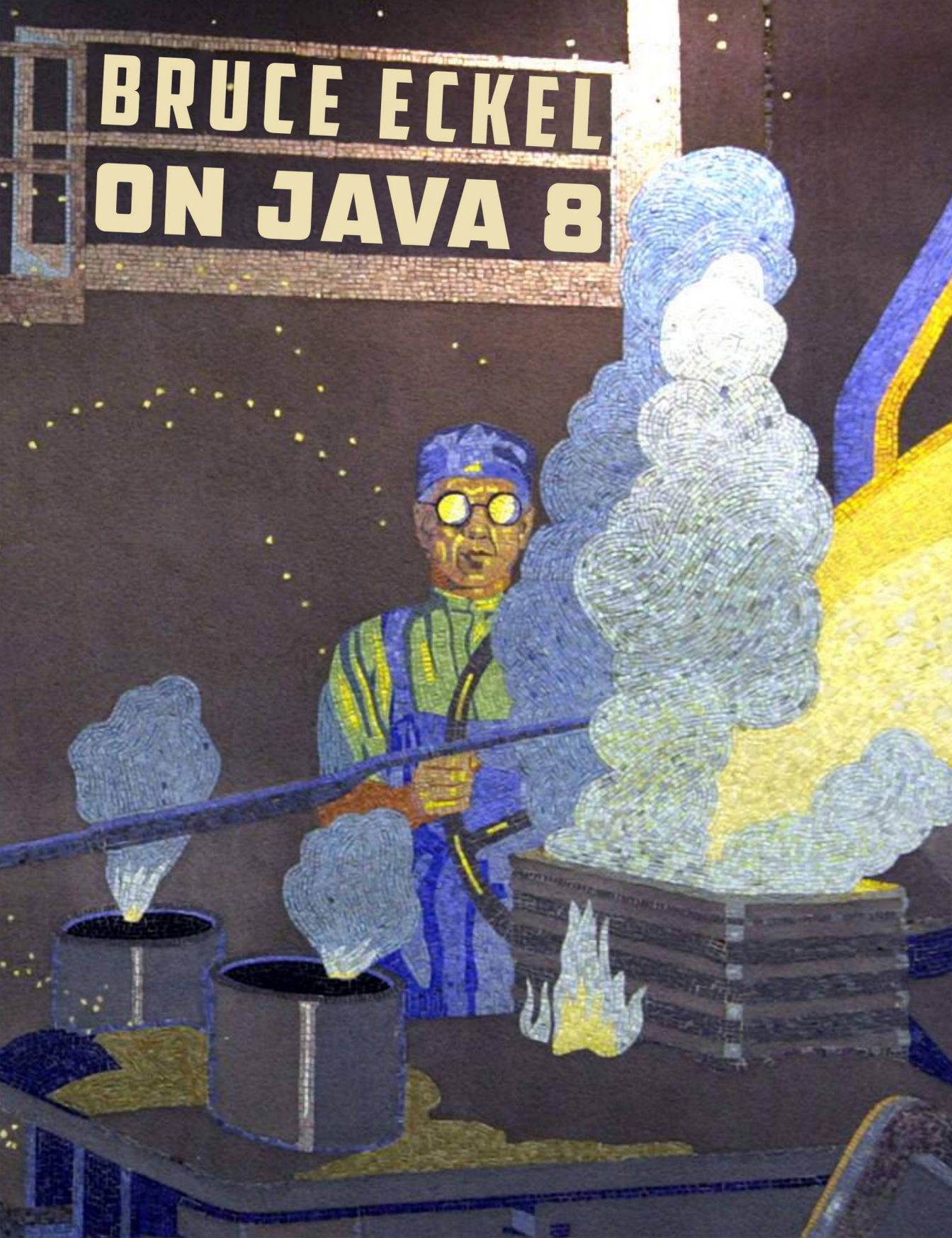
BRUCE ECKEL
ON JAVA 8



On Java 8

Bruce Eckel

This book is for sale at http://leanpub.com/onjava8

This version was published on 2021-03-05

ISBN 978-0-9818725-2-0



This is a Leanpub book. Leanpub empowers authors and publishers with the Lean Publishing process. Lean Publishing is the act of publishing an in-progress ebook using lightweight tools and many iterations to get reader feedback, pivot until you have the right book and build traction once you do.

© 2020 - 2021 Mindview LLC

Also By Bruce Eckel

Atomic Kotlin

Contents

Copyright **1**

Foreword to the Leanpub Edition **2**
Updates 2
January 2021 Update 3
March 2021 Update 3

Preface **4**
Goals 4
Language Design Errors 5
Tested Examples 7
Popularity 8
Android Programmers 8
This is Only an eBook 8
Colophon 9
Thanks 10
Dedication 10

Introduction **11**
Prerequisites 12
JDK HTML Documentation 13
Thinking in C 13
Source Code 14
Coding Standards 16
Bug Reports 17
What About User Interfaces? 17

What is an Object? **19**

The Progress of Abstraction 20
An Object Has an Interface 22
Objects Provide Services 24
The Hidden Implementation 25
Reusing the Implementation 27
Inheritance 28
Interchangeable Objects with Polymorphism 33
The Singly-Rooted Hierarchy 37
Collections 38
Object Creation & Lifetime 40
Exception Handling: Dealing with Errors 42
Summary 43

Installing Java and the Book Examples 45
Editors 45
The Shell 46
Installing Java 48
Verify Your Installation 49
Installing and Running the Book Examples 50

Objects Everywhere 52
You Manipulate Objects with References 52
You Must Create All the Objects 53
Comments 58
You Never Need to Destroy an Object 59
Creating New Data Types: `class` 61
Methods, Arguments, and Return Values 63
Writing a Java Program 66
Your First Java Program 70
Coding Style 75
Summary 75

Operators 76
Using Java Operators 76
Precedence 77
Assignment 77
Mathematical Operators 80

Auto Increment and Decrement 84
Relational Operators 85
Logical Operators 91
Literals 94
Bitwise Operators 98
Shift Operators 99
Ternary `if-else` Operator 104
`String` Operator + and += 105
Common Pitfalls When Using Operators 106
Casting Operators 107
Java Has No "sizeof" 110
A Compendium of Operators 111
Summary 121

Control Flow 123
`true` and `false` 123
`if-else` 124
Iteration Statements 125
For-in Syntax 129
`return` 133
`break` and `continue` 134
The Infamous "Goto" 135
`switch` 141
Switching on Strings 143
Summary 146

Housekeeping 147
Guaranteed Initialization with the Constructor 147
Method Overloading 150
Zero-Argument Constructors 158
The `this` Keyword 160
Cleanup: Finalization and Garbage Collection 165
Member Initialization 173
Constructor Initialization 177
Array Initialization 185
Enumerated Types 197

Summary 200

Implementation Hiding 202
`package`: the Library Unit 203
Java Access Specifiers 213
Interface and Implementation 221
Class Access 223
Summary 226

Reuse 228
Composition Syntax 228
Inheritance Syntax 232
Delegation 237
Combining Composition and Inheritance 239
Choosing Composition vs. Inheritance 247
`protected` 249
Upcasting 251
The `final` Keyword 253
Initialization and Class Loading 262
Summary 265

Polymorphism 266
Upcasting Revisited 267
The Twist 270
Constructors and Polymorphism 284
Covariant Return Types 295
Designing with Inheritance 296
Summary 301

Interfaces 303
Abstract Classes and Methods 303
Defining Interfaces 309
Abstract Classes vs. Interfaces 323
Complete Decoupling 324
Combining Multiple Interfaces 331
Extending an Interface with Inheritance 333
Adapting to an Interface 336

Fields in Interfaces . . . 340
Nesting Interfaces . . . 341
Interfaces and Factories . . . 345
Summary . . . 348

Inner Classes . . . 350
Creating Inner Classes . . . 350
The Link to the Outer Class . . . 353
Using `.this` and `.new` . . . 355
Inner Classes and Upcasting . . . 357
Inner Classes in Methods and Scopes . . . 359
Anonymous Inner Classes . . . 361
Nested Classes . . . 367
Why Inner Classes? . . . 371
Inheriting from Inner Classes . . . 385
Can Inner Classes Be Overridden? . . . 386
Local Inner Classes . . . 389
Inner-Class Identifiers . . . 391
Summary . . . 391

Collections . . . 393
Generics and Type-Safe Collections . . . 394
Basic Concepts . . . 398
Adding Groups of Elements . . . 400
Printing Collections . . . 402
`List` . . . 405
`Iterators` . . . 409
`LinkedList` . . . 414
`Stack` . . . 417
`Set` . . . 420
`Map` . . . 426
`Queue` . . . 429
`Collection` vs. `Iterator` . . . 433
for-in and Iterators . . . 438
Summary . . . 445

Functional Programming . . . 450

Old vs. New . . . 452
Lambda Expressions . . . 454
Method References . . . 459
Functional Interfaces . . . 465
Higher-Order Functions . . . 477
Closures . . . 480
Function Composition . . . 486
Currying and Partial Evaluation . . . 489
Pure Functional Programming . . . 492
Summary . . . 492

Streams . . . 494
Java 8 Stream Support . . . 497
Stream Creation . . . 497
Intermediate Operations . . . 513
Optional . . . 524
Terminal Operations . . . 538
Summary . . . 551

Exceptions . . . 552
Concepts . . . 553
Basic Exceptions . . . 554
Catching an Exception . . . 556
Creating Your Own Exceptions . . . 558
The Exception Specification . . . 566
Catching Any Exception . . . 567
Standard Java Exceptions . . . 580
Performing Cleanup with `finally` . . . 583
Exception Restrictions . . . 591
Constructors . . . 595
Try-With-Resources . . . 601
Exception Matching . . . 610
Alternative Approaches . . . 611
Exception Guidelines . . . 621
Summary . . . 621

Validating Your Code . . . 625

Testing . . . 625
Preconditions . . . 632
Test-Driven Development . . . 653
Logging . . . 662
Debugging . . . 665
Benchmarking . . . 670
Profiling and Optimizing . . . 681
Style Checking . . . 682
Static Error Analysis . . . 683
Code Reviews . . . 683
Pair Programming . . . 684
Refactoring . . . 685
Continuous Integration . . . 686
Summary . . . 688

Files . . . 690
File and Directory Paths . . . 690
Directories . . . 699
File Systems . . . 703
Watching a `Path` . . . 705
Finding Files . . . 708
Reading & Writing Files . . . 711
Summary . . . 714

Strings . . . 715
Immutable `Strings` . . . 715
Overloading + vs. `StringBuilder` . . . 716
Unintended Recursion . . . 722
Operations on `Strings` . . . 724
Formatting Output . . . 728
Regular Expressions . . . 738
Scanning Input . . . 763
`StringTokenizer` . . . 768
Summary . . . 769

Reflection . . . 770
The Need for Reflection . . . 770

The `Class` Object 773
Checking Before a Cast 787
Registered Factories 801
`Instanceof` vs. `Class` Equivalence 804
Runtime Class Information 806
Dynamic Proxies 811
Using Optional 816
Interfaces and Type Information 827
Summary 835

Generics 837
Comparison with C++ 838
Simple Generics 839
Generic Interfaces 849
Generic Methods 853
Building Complex Models 866
The Mystery of Erasure 869
Compensating for Erasure 882
Bounds 895
Wildcards 900
Issues 922
Self-Bounded Types 931
Dynamic Type Safety 940
Exceptions 942
Mixins 944
Latent Typing 952
Compensating for the Lack of (Direct) Latent Typing 960
Assisted Latent Typing in Java 8 967
Summary: Is Casting Really So Bad? 973

Arrays 976
Why Arrays are Special 976
Arrays are First-Class Objects 981
Returning an Array 984
Multidimensional Arrays 986
Arrays and Generics 991

Arrays.fill() 994
Arrays.setAll() 996
Incremental Generators 998
Random Generators 1008
Generics and Primitive Arrays 1019
Modifying Existing Array Elements 1024
An Aside On Parallelism 1024
Arrays Utilities 1027
Copying an Array 1028
Comparing Arrays 1031
Streams and Arrays 1033
Sorting Arrays 1035
Searching with Arrays.binarySearch() 1041
Accumulating with parallelPrefix() 1044
Summary 1046

Enumerations 1050
Basic enum Features 1050
Adding Methods to an enum 1053
enums in switch Statements 1056
The Mystery of values() 1057
Implements, not Inherits 1061
Random Selection 1062
Using Interfaces for Organization 1063
Using EnumSet Instead of Flags 1070
Using EnumMap 1072
Constant-Specific Methods 1074
Multiple Dispatching 1091
Summary 1105

Annotations 1107
Basic Syntax 1108
Writing Annotation Processors 1112
Using javac to Process Annotations 1122
Annotation-Based Unit Testing 1132
Summary 1159

Concurrent Programming **1160**
The Terminology Problem 1161
Concurrency Superpowers 1165
Concurrency is for Speed 1168
The Four Maxims of Java Concurrency 1170
The Brutal Truth 1174
The Rest of the Chapter 1176
Parallel Streams 1178
Creating and Running Tasks 1192
Terminating Long-Running Tasks 1206
`CompletableFutures` 1209
Deadlock 1234
Constructors are not Thread-Safe 1240
Effort, Complexity, Cost 1245
Summary 1254

Patterns **1259**
The Pattern Concept 1259
Singleton 1261
Classifying Patterns 1266
Template Method 1267
Fronting for an Implementation 1269
Factories: Encapsulating Object Creation 1277
Function Objects 1287
Changing the Interface 1295
Interpreter: Run-Time Flexibility 1299
Callbacks 1300
Multiple Dispatching 1308
Pattern Refactoring 1314
Summary 1352

Appendix: Supplements **1354**
On Java 8 Example Code 1354
Thinking in C: Foundations for Java 1354
Hands-On Java eSeminar 1354

Appendix: Programming Guidelines **1355**

Design 1355
Implementation 1361

Appendix: Javadoc 1367
Syntax 1367
Embedded HTML 1368
Some Example Tags 1369
Documentation Example 1372

Appendix: Passing and Returning Objects 1374
Passing References 1374
Making Local Copies 1378
Controlling Cloneability 1397
Immutable Classes 1409
Summary 1419

Appendix: I/O Streams 1421
Types of `InputStream` 1422
Types of `OutputStream` 1424
Adding Attributes and Useful Interfaces 1426
`Readers` & `Writers` 1429
Off By Itself: `RandomAccessFile` 1433
Typical Uses of I/O Streams 1434
Summary 1444

Appendix: Standard I/O 1445
Reading from Standard Input 1445
Changing `System.out` to a `PrintWriter` 1446
Redirecting Standard I/O 1447
Process Control 1449

Appendix: New I/O 1452
`ByteBuffers` 1452
Converting Data 1457
Fetching Primitives 1461
View Buffers 1463
Data Manipulation with Buffers 1468

Memory-Mapped Files 1473
File Locking 1478

Appendix: Understanding `equals()` and `hashCode()` 1483
A Canonical `equals()` 1483
Hashing and Hash Codes 1494
Tuning a `HashMap` 1512

Appendix: Collection Topics 1514
Sample Data 1514
`List` Behavior 1523
`Set` Behavior 1527
Using Functional Operations with any `Map` 1530
Selecting Parts of a `Map` 1532
Filling Collections 1534
Custom `Collection` and `Map` using *Flyweight* 1541
`Collection` Functionality 1554
Optional Operations 1560
`Set`s and Storage Order 1564
Queues 1570
Understanding `Map`s 1574
Utilities 1583
Holding References 1593
Java 1.0/1.1 Collections 1599
Summary 1605

Appendix: Low-Level Concurrency 1607
What is a Thread? 1607
Catching Exceptions 1613
Sharing Resources 1618
The `volatile` Keyword 1627
Atomicity 1631
Critical Sections 1642
Library Components 1650
Summary 1660

Appendix: Data Compression 1661

Simple Compression with GZIP 1662
Multifile Storage with Zip 1663
Java Archives (Jars) 1666

Appendix: Object Serialization 1669
Finding the Class 1674
Controlling Serialization 1675
Using Persistence 1686

Appendix: Benefits and Costs of Static Type Checking 1699
Foreword to the 2021 Edition 1699
Foreword to the Original Edition 1700
Static Type Checking vs. Testing 1701
How to Argue about Typing 1710
The Cost of Productivity 1727
Static vs. Dynamic 1731

Appendix: The Positive Legacy of C++ and Java 1739

Appendix: Becoming a Programmer 1742
How I Got Started in Programming 1742
A Career in Computing 1744
The Mythical 5% 1748
Writing Software Is Like ... Writing 1753
Programming as Typing 1756
Do What You Love 1761

Copyright

Copyright ©2017, ©2021

by Bruce Eckel, President, MindView LLC.

ISBN 978-0-9818725-2-0

All rights reserved. Produced in the United States of America. This publication is protected by copyright, and permission must be obtained from the publisher prior to any prohibited reproduction, storage in a retrieval system, or transmission in any form or by any means, electronic, mechanical, photocopying, recording, or likewise.

Many of the designations used by manufacturers and sellers to distinguish their products are claimed as trademarks. Where those designations appear in this book, and the publisher was aware of a trademark claim, the designations have been printed with initial capital letters or in all capitals.

Java is a trademark of Oracle, Inc. Windows 95, Windows NT, Windows 2000, Windows XP, Windows 7, Windows 8 and Windows 10 are trademarks of Microsoft Corporation. All other product names and company names mentioned herein are the property of their respective owners.

The author and publisher have taken care in the preparation of this book, but make no expressed or implied warranty of any kind and assume no responsibility for errors or omissions. No liability is assumed for incidental or consequential damages in connection with or arising out of the use of the information or programs contained herein.

This book was created as an eBook for tablets and computers. That is, it was not first created for print and then converted. It is an eBook first—all layout and formatting is designed to optimize your viewing experience on the various eBook reading platforms and systems.

Cover design by Daniel Will-Harris, www.Will-Harris.com[1].

[1] http://www.Will-Harris.com

Foreword to the Leanpub Edition

In 2017, I published this book on Google Play Books. As an experiment, it was *exclusively* on that platform. This not only turned out to be a challenging process, but because Google doesn't do business in a number of countries where I have readers, it prevented those readers from getting the book. It wasn't a good fit, but in the meantime I wrote off the experiment and focused on the next book, Atomic Kotlin[2].

Over time, awareness of the book trickled out and it turned out there was demand for it. After putting *Atomic Kotlin* on Leanpub, I decided it would be worthwhile to convert *On Java 8* to Leanpub as well, which is what you now (virtually) hold in your hands.

Leanpub generates multiple formats: PDF, ePub (especially nice on Apple iOS devices via iBooks. There are also ePub readers for most other platforms) and MOBI, for Kindle readers. In addition, there's a version that you can read directly online through the Leanpub site.

From the table of contents you can jump to any part of the book. Note that in the PDF version you must click on the *name* of the chapter or subhead you want to go to, not the page number.

Unfortunately, Leanpub does not support indices (I know of no ebook-creation systems that do), but the ebook allows you to easily search for topics.

Updates

The Leanpub edition of the book also contains a number of updates. The initial updates (*not* chronicled below) were primarily to the code, allowing the examples to

[2] https://www.atomickotlin.com/

run with newer versions of Java, along with more recent versions of the Gradle build tool and JUnit testing libraries.

Note that, although newer versions of Java have introduced a handful of features that are useful, Java 8 contained the most sweeping and dramatic changes to the language since Java 5 (covered in *Thinking in Java, 4th Edition*). After reading *On Java 8* you should be able to understand the features that have been added since.

January 2021 Update

This primarily included a rewrite of the **Reflection** chapter. It also added missing `@Override` annotations throughout the book (because Java doesn't enforce the use of `@Override`). The `/* Output:` sections of all examples have been updated, and the examples were reformatted in various ways to improve readability.

March 2021 Update

- **Patterns**: Thorough rewrite of most examples & prose. This is the largest change in this update.
- **Objects Everywhere**: Fixed primitive data type table (superscript values were missing).
- **Operators**: Rewrote "Testing Object Equivalence" section.
- **Implementation Hiding**: Rewrote `onjava/Range.java` and added `onjava/TestRange.java`.
- Changed "RTTI" to "Reflection" throughout the book, including changing the **Type Information** chapter name to **Reflection**.
- Fixed the swapped images in **Reuse** and **Polymorphism**.
- Changed from "no-arg constructor" to "zero-argument constructor" throughout, following modern terminology in Java documentation.
- Various grammar & spelling fixes throughout the book.

Preface

> This book teaches Java programming using the features in the 8th version of that language.

My previous Java book, *Thinking in Java, 4th Edition* (Prentice Hall 2006), is still useful for programming in Java 5, the version of the language used for Android programming. But with the advent of Java 8, the language has changed significantly enough that new Java code feels and reads differently. This justified the two-year effort of creating a new book.

On Java 8 is designed for someone with a basic foundation in programming. For beginners, web sites like Code.org[3] and Khan Academy[4] can provide at least some of that background, along with the free Thinking in C[5] seminar. Services like YouTube, blogs and StackOverflow have made finding answers ridiculously easy compared to just a few years ago when we relied on print media. Combine these with perseverance, and you can use this book as your first programming text. It's also intended for professional programmers who want to expand their knowledge.

I am grateful for all the benefits from *Thinking in Java*, mostly in the form of speaking engagements all over the world. It has proved invaluable in creating connections with people and companies.

Goals

Each chapter teaches a concept, or a group of associated concepts, without relying on features that haven't yet been introduced. That way you can digest each piece in the context of your current knowledge before moving on.

My goals in this book are to:

[3] https://code.org/learn
[4] https://www.khanacademy.org/computing/computer-programming
[5] https://archive.org/details/ThinkingInC

1. Present the material one step at a time so you can easily incorporate each idea before moving on, and to carefully sequence the presentation of features so you're exposed to a topic before you see it in use. This isn't always possible; in those situations, a brief introductory description is given.
2. Use examples that are as simple and short as possible. This sometimes prevents me from tackling "real world" problems, but I've found that beginners are usually happier when they can understand every detail of an example rather than being impressed by the scope of the problem it solves. For this I might receive criticism for using "toy examples," but I'm willing to accept that in favor of producing something pedagogically useful.
3. Teach what I think is important about the language, rather than everything I know. I believe there is an information importance hierarchy, and there are some facts that 95 percent of programmers will never need to know—details that just confuse people and increase their perception of the complexity of the language. If *you* must think about it, it will also confuse the reader/maintainer of that code, so I advocate choosing a simpler approach.
4. Provide you with a solid foundation so you understand the issues well enough to move on to more difficult coursework and books.

Language Design Errors

Every language has design errors. New programmers experience deep uncertainty and frustration when they must wade through features and guess at what they should use and what they shouldn't. It's embarrassing to admit mistakes, but this bad beginner experience is a lot worse than the discomfort of acknowledging you were wrong about something. Alas, every failed language/library design experiment is forever embedded in the Java distribution.

The Nobel laureate economist Joseph Stiglitz has a philosophy of life that applies here, called *The Theory of Escalating Commitment*:

> "The cost of continuing mistakes is borne by others, while the cost of admitting mistakes is borne by yourself."

If you've read my past writings, you'll know that when I find design errors in a language, I tend to point them out. Java has developed a particularly avid

following, folks who treat the language more like a country of origin and less like a programming tool. Because I've written about Java, they assume I am a fellow patriot. When I criticize the errors I find, it tends to have two effects:

1. Initially, a lot of "my-country-right-or-wrong" furor, which typically dies down to isolated pockets. Eventually—this can take years—the error is acknowledged and seen as just part of the history of Java.
2. More importantly, new programmers don't go through the struggle of wondering why "they" did it this way, especially the self-doubt that comes from finding something that just doesn't seem right and naturally assuming *I must be doing it wrong or I just don't get it.* Worse, those who teach the language often go right along with the misconceptions rather than delving in and analyzing the issue. By understanding the language design errors, new programmers can understand that something was a mistake, and move forward.

Understanding language and library design errors is essential because of the impact they have on programmer productivity. Some companies and teams choose to avoid certain features because, while seductive on the surface, those features can block your progress when you least expect it. Design errors also inform the creation and adoption of new languages. It's fun to explore what can be done with a language, but design errors tell you what *can't* be done with that language.

For many years, I honestly felt a lack of care from the Java designers regarding their users. Some of these errors seemed so blatant, so poorly thought-out, that it appeared the designers had some other motivation in mind instead of serving their users. There was a lot of notoriety around the Java language for a long time, and perhaps that's where the seduction was. This seeming lack of respect for programmers is the major reason I moved away from Java and didn't want anything to do with it for such a long time.

When I did start looking into Java again, something about Java 8 felt very different, as if a fundamental shift had occurred in the designers' attitude about the language and its users. Many features and libraries that had been warts on the language were fixed after years of ignoring user complaints. New features felt very different, as if there were new folks on board who were extremely interested in programmer experience. These features were—finally—working to make the language better rather than just quickly adding ideas without delving into their implications. And some of the new features are downright elegant (or at least, as elegant as possible given Java

constraints). I can only guess that some person or people have departed the language group and this has changed the perspective.

Because of this new focus by the language developers—and I don't think I'm imagining it—writing this book has been dramatically better than past experiences. Java 8 contains fundamental and important improvements. Alas, because of Java's rigid backwards-compatibility promise, these improvements required great effort so it's unlikely we'll see anything this dramatic again (I hope I'm wrong about this). Nonetheless, I applaud those who have turned the ship as much as they have and set the language on a better course. For the first time I can ever recall, I found myself saying "I love that!" about some of the Java code I've been able to write in Java 8.

Ultimately, the timing for this book seems good, because Java 8 introduces important features that strongly affect the way code is written, while—so far—Java 9 seems to focus on the understory of the language, bringing important infrastructure features but not those that affect the kind of coding focused on in this book. However, because it's an eBook, if I discover something I think requires an update or an addition, I can push the new version to existing customers.

Tested Examples

The code examples in this book compile with Java 8 and the Gradle[6] build tool. I have also tested it with newer versions of Java, but I recommend you choose one of the *Long-Term Support* versions of the language—at this writing, either Java 8 or Java 11. All the examples are in a freely-accessible Github repository[7].

Without a built-in test framework with tests that run every time you do a build of your system, you have no way of knowing whether your code is reliable. To accomplish this in the book, I created a test system to display and validate the output of most examples. The output from running an example is attached to that example, as a block comment, at the end of each example that produces output. In some cases only the first few lines are shown, or first and last lines. Embedded output improves the reading and learning experience, and provides yet another way to verify the correctness of the examples.

[6] https://gradle.org/
[7] https://github.com/BruceEckel/Onjava8-examples/

Popularity

Java's popularity has significant implications. If you learn it, getting a job will probably be easier. There are many training materials, courses, and other learning resources available. If you're starting a company and you choose to work in Java, it can be easier to find programmers, and that's a compelling argument.

Short-term thinking is almost always a bad idea. Don't use Java if you really don't like it—using it just to get a job is an unhappy life choice. As a company, think hard before choosing Java just because you can hire people. There might be another language that makes fewer employees far more productive for your particular need (for example, see my book Atomic Kotlin[8]). Using a newer and more exciting language might make it easier to attract programmers, as well.

But if you do enjoy it, if Java *does* call to you, then welcome. I hope this book will enrich your programming experience.

Android Programmers

Android programming with Java is tied to Java 5. I've made this book as "Java 8 as possible," so if you want to program in Java for Android devices, you're better off with *Thinking in Java, 4th edition*, available in print from Prentice-Hall and as an eBook through Leanpub[9]. In addition, there are many other resources that specialize in Android programming.

Note that Kotlin[10] is a newer and much better choice—and an official language—for Android programming.

This is Only an eBook

The English version of *On Java 8* is only available as an eBook, and only via www.OnJava8.com[11]. Any other source or delivery mechanism is illegitimate. There is no print version.

[8] https://www.AtomicKotlin.com
[9] https://leanpub.com/
[10] https://www.AtomicKotlin.com
[11] http://www.OnJava8.com

This is copyrighted work. Do not post or share it in any way without permission via mindviewinc@gmail.com[12]. You may use the examples for teaching, as long as they are not republished without permission and attribution. See the `Copyright.txt` file in the example distribution for full details.

Because of the size of this book, my intent has always been to only publish it as an eBook. Color syntax highlighting for code listings is, alone, worth the cost of admission. Searchability, font resizing, text-to-voice for the vision-impaired, the fact you can always keep it with you—there are so many benefits to eBooks it's hard to name them all.

Anyone buying this book needs a computer to run the programs and write code, and the eBook reads nicely on a computer (I was also surprised to discover that it even reads tolerably well on a phone). However, the best reading experience is on a tablet computer. Tablets are inexpensive enough that you can now buy one for less than you'd pay for an equivalent print version of this book. It's much easier to read a tablet in bed (for example) than trying to manage the pages of a physical book, especially one this big. When working at your computer, you don't have to hold the pages open when using a tablet at your side. It might feel different at first, but I think you'll find the benefits far outweigh the discomfort of adapting.

Colophon

This book was written with Markdown and produced into various eBook formats using Leanpub[13].

The body font is Linux Libertine and the headline font is Open Sans. The code font is Anonymous Pro Mono, because it is especially compact and allows more characters on a line without wrapping. I chose to place the code inline (rather than make listings into images, as I've seen some books do) because it was important to me that the reader be able to resize the font of the code listings when they resize the body font (otherwise, really, what's the point?).

The build process for the book was automated, as well as the process to extract, compile and test the code examples. All automation was achieved through fairly extensive programs I wrote in Python 3.

[12] mailto:mindviewinc@gmail.com
[13] https://leanpub.com

Cover Design

The cover of *On Java 8* is from a mosaic created through the *Works Progress Administration* (WPA, a huge project during the US Great Depression from 1935-1943 which put millions of out-of-work people back to work). It also reminds me of the illustrations from *The Wizard of Oz* series of books. My friend and designer, Daniel Will-Harris (www.will-harris.com[14]) and I just liked the image.

Thanks

Thanks to Eric Evans (author of *Domain-Driven Design*) for suggesting the book title, and to everyone else in the conference newsgroups for their help in finding the title.

Thanks to James Ward for starting me with the Gradle build tool for this book, and for his help and friendship over the years. Thanks to Ben Muschko for his work polishing the build files, and Hans Dockter for giving Ben the time.

Jeremy Cerise and Bill Frasure came to the developer retreat for the book and followed up with valuable help.

Thanks to all who have taken the time and effort to come to my conferences, workshops, developer retreats, and other events in my town of Crested Butte, Colorado. Your contributions might not be easily seen, but they are deeply important.

Dedication

For my beloved father, E. Wayne Eckel. April 1, 1924—November 23, 2016.

[14] http://www.will-harris.com

Introduction

> "The limits of my language are the limits of my world"—*Ludwig Wittgenstein (1889-1951)*

This is true of both spoken/written languages and programming languages. It's often subtle: A language gently guides you into certain modes of thought and away from others. Java is particularly opinionated.

Java is a derived language. The original language designers didn't want to use C++ for a project, so created a new language which unsurprisingly looked a lot like C++, but with improvements. The core changes were the incorporation of a *virtual machine* and *garbage collection*, both of which are described in detail in this book. Java is also responsible for pushing the industry forward in other ways; for example, most languages are now expected to include documentation markup syntax and a tool to produce HTML documentation.

One of the most predominant Java concepts came from the SmallTalk language, which insists that the "object" (described in the next chapter) is the fundamental unit of programming, so everything must be an object. Time has tested this belief and found it overenthusiastic. Some folks even declare that objects are a complete failure and should be discarded. Personally, I find that making everything an object is not only an unnecessary burden but also pushes many designs in a poor direction. However, there are still situations where objects shine. Requiring that everything be an object (especially all the way down to the lowest level) is a design mistake, but banning objects altogether seems equally draconian.

Other Java language decisions haven't panned out as promised. Throughout this book I attempt to explain these so you not only understand those features, but also why they might not feel quite right to you. It's not about declaring that Java is a good language or a bad one. If you understand the flaws and limitations of the language you will:

1. Not get stymied when you encounter a feature that seems "off."

2. Design and code better by knowing where the boundaries are.

Programming is about managing complexity: the complexity of the problem, laid upon the complexity of the machine. Because of this complexity, most of our programming projects fail.

Many language design decisions are made with complexity in mind, but at some point other issues are considered essential. Inevitably, those other issues are what cause programmers to eventually "hit the wall" with a language. For example, C++ had to be backward-compatible with C (to allow easy migration for C programmers), as well as efficient. Those are both useful goals and account for much of the success of C++, but they also expose extra complexity that prevent some projects from finishing. Certainly, you can blame programmers and management, but if a language can help by catching your mistakes, why shouldn't it?

Visual BASIC (VB) was tied to BASIC, which wasn't really designed as an extensible language. All the extensions piled upon VB have produced some truly un-maintainable syntax. The Perl language is backward-compatible with `awk`, `sed`, `grep`, and other Unix tools it was meant to replace, and as a result it is often accused of producing "write-only code" (that is, you can't read your own code). On the other hand, C++, VB, Perl, and other languages such as SmallTalk had *some* of their design efforts focused on the issue of complexity and as a result are remarkably successful in solving certain types of problems.

The communication revolution enables all of us to communicate with each other more easily: one-on-one as well as in groups and as a planet. I've heard it suggested that the next revolution is the formation of a kind of global mind that results from enough people and enough interconnectedness. Java might or might not be one of the tools for that revolution.

Prerequisites

This book assumes you have some programming familiarity, so you understand:

- A program is a collection of statements
- The idea of a subroutine/function/macro
- Control statements such as "if" and looping constructs such as "while"

- Etc.

You might have learned this in many places, typically school, books, or the Internet. As long as you you feel comfortable with the basic ideas of programming, you can work through this book.

If you don't feel comfortable with foundational concepts, the free Thinking in C[15] seminar will bring you up to speed on the fundamentals necessary to learn Java. *On Java 8* introduces the concepts of object-oriented programming (OOP) and Java's basic control mechanisms.

I occasionally refer to features in C and C++ (and sometimes other languages). These comments do not require that you understand C/C++. They are intended to help all programmers put Java in perspective with those languages, from which, after all, Java is descended. I attempt to make these references simple and to explain anything that might be unfamiliar to a non-C/C++ programmer.

JDK HTML Documentation

The *Java Development Kit* (JDK) from Oracle (a free download[16]) comes with documentation in electronic form, readable through your Web browser. Unless necessary, this book will not repeat that documentation, because it's usually much faster to find the class descriptions with your browser than to look them up in a book (also, the online documentation is current). I'll simply refer to "the JDK documentation." I'll provide extra descriptions of the classes only when it's necessary to supplement that documentation so you understand a particular example.

Thinking in C

The free Thinking in C[17] seminar gives an introduction to the C syntax, operators, and functions that are the foundation of Java syntax. *Thinking in C* also provides a gentle introduction to coding, assuming even less about the student's programming background than does this book.

[15] https://archive.org/details/ThinkingInC
[16] http://java.oracle.com
[17] https://archive.org/details/ThinkingInC

I commissioned Chuck Allison to create *Thinking in C* as a standalone product, which was later included in book CDs, and finally reworked as a free download. By freely providing this seminar online, I can ensure that everyone begins with adequate preparation.

Source Code

All the source code for this book is available as copyrighted freeware, distributed via Github[18]. To ensure you have the most current version, this is the official code distribution site. You may use this code in classroom and other educational situations.

The primary goal of the copyright is to ensure that the source of the code is properly cited, and to prevent you from republishing the code without permission. (As long as this book is cited, using examples from the book in most media is generally not a problem.)

In each source-code file you find a reference to the following copyright notice:

```
// Copyright.txt
This computer source code is Copyright ©2021 MindView LLC.
All Rights Reserved.

Permission to use, copy, modify, and distribute this
computer source code (Source Code) and its documentation
without fee and without a written agreement for the
purposes set forth below is hereby granted, provided that
the above copyright notice, this paragraph and the
following five numbered paragraphs appear in all copies.

1. Permission is granted to compile the Source Code and to
include the compiled code, in executable format only, in
personal and commercial software programs.

2. Permission is granted to use the Source Code without
modification in classroom situations, including in
presentation materials, provided that the book "On
Java 8" is cited as the origin.
```

[18] https://github.com/BruceEckel/Onjava8-examples/

3. Permission to incorporate the Source Code into printed
media may be obtained by contacting:

MindView LLC, PO Box 969, Crested Butte, CO 81224
MindViewInc@gmail.com

4. The Source Code and documentation are copyrighted by
MindView LLC. The Source code is provided without express
or implied warranty of any kind, including any implied
warranty of merchantability, fitness for a particular
purpose or non-infringement. MindView LLC does not
warrant that the operation of any program that includes the
Source Code will be uninterrupted or error-free. MindView
LLC makes no representation about the suitability of the
Source Code or of any software that includes the Source
Code for any purpose. The entire risk as to the quality
and performance of any program that includes the Source
Code is with the user of the Source Code. The user
understands that the Source Code was developed for research
and instructional purposes and is advised not to rely
exclusively for any reason on the Source Code or any
program that includes the Source Code. Should the Source
Code or any resulting software prove defective, the user
assumes the cost of all necessary servicing, repair, or
correction.

5. IN NO EVENT SHALL MINDVIEW LLC, OR ITS PUBLISHER BE
LIABLE TO ANY PARTY UNDER ANY LEGAL THEORY FOR DIRECT,
INDIRECT, SPECIAL, INCIDENTAL, OR CONSEQUENTIAL DAMAGES,
INCLUDING LOST PROFITS, BUSINESS INTERRUPTION, LOSS OF
BUSINESS INFORMATION, OR ANY OTHER PECUNIARY LOSS, OR FOR
PERSONAL INJURIES, ARISING OUT OF THE USE OF THIS SOURCE
CODE AND ITS DOCUMENTATION, OR ARISING OUT OF THE INABILITY
TO USE ANY RESULTING PROGRAM, EVEN IF MINDVIEW LLC, OR
ITS PUBLISHER HAS BEEN ADVISED OF THE POSSIBILITY OF SUCH
DAMAGE. MINDVIEW LLC SPECIFICALLY DISCLAIMS ANY
WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED
WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR
PURPOSE. THE SOURCE CODE AND DOCUMENTATION PROVIDED
HEREUNDER IS ON AN "AS IS" BASIS, WITHOUT ANY ACCOMPANYING
SERVICES FROM MINDVIEW LLC, AND MINDVIEW LLC HAS NO
OBLIGATIONS TO PROVIDE MAINTENANCE, SUPPORT, UPDATES,

```
ENHANCEMENTS, OR MODIFICATIONS.

Please note that MindView LLC maintains a Web site which
is the sole distribution point for electronic copies of the
Source Code, https://github.com/BruceEckel/OnJava8-examples,
where it is freely available under the terms stated above.

If you think you've found an error in the Source Code,
please submit a correction at:
https://github.com/BruceEckel/OnJava8-examples/issues
```

You may use the code in your projects and in the classroom (including your presentation materials) as long as the copyright notice that appears in each source file is retained.

Coding Standards

In the text of this book, identifiers (keywords, methods, variables, and class names) are set in fixed-width `code font`. Some keywords, such as `class`, are used so much that the special font can become tedious. Terms that are distinctive enough are left in the normal font.

I use a particular coding style for the examples in this book. As much as possible within the book's formatting constraints, this follows the style that Oracle itself uses in virtually all code you find at its site, and seems to be supported by most Java development environments. As the subject of formatting style is good for hours of hot debate, I'll just say I'm not trying to dictate correct style via my examples; I have my own motivation for using the style I do. Because Java is a free-form programming language, continue to use whatever style you're comfortable with. One solution to the coding style issue is to use an IDE (*integrated development environment*) tool like IntelliJ IDEA, Eclipse or NetBeans to change formatting to that which suits you.

The code files in this book are tested with an automated system, and should work without compiler errors (except those specifically tagged) in the latest version of Java.

This book focuses on and is tested with Java 8. If you must learn about earlier releases of the language not covered here, you can find the 4th edition of *Thinking in Java* through www.OnJava8.com[19].

[19] http://www.OnJava8.com

Bug Reports

No matter how many tools a writer uses to detect errors, some always creep in and these often leap off the page for a fresh reader. If you discover anything you believe to be an error, please submit the error along with your suggested correction, for either the book's prose or examples, here[20]. Your help is appreciated.

What About User Interfaces?

Graphical user interfaces and desktop programming in Java have had a tumultuous—some would say tragic—history.

The original design goal of the graphical user interface (GUI) library in Java 1.0 was to enable the programmer to build a GUI to look good on all platforms. That goal was not achieved. Instead, the Java 1.0 *Abstract Windowing Toolkit* (AWT) produced a GUI that looked equally mediocre on all systems. In addition, it was restrictive; you could use only four fonts and you could not access any of the more sophisticated GUI elements that exist in your operating system. The Java 1.0 AWT programming model was also awkward and non-object-oriented. A student in one of my seminars (who had been at Sun during the creation of Java) explained why: The original AWT had been conceived, designed, and implemented in a month. Certainly a marvel of productivity, and also an object lesson in why design is important.

The situation improved with the Java 1.1 AWT event model, which took a much clearer, object-oriented approach, along with the addition of JavaBeans, a component programming model (now dead) oriented toward the easy creation of visual programming environments. Java 2 (Java 1.2) finished the transformation away from the old Java 1.0 AWT by essentially replacing everything with the *Java Foundation Classes* (JFC), the GUI portion of which is called "Swing." These are a rich set of JavaBeans that create a reasonable GUI. The *revision 3* rule of the software industry ("a product isn't good until revision 3") seems to hold true with programming languages as well.

It seemed that Swing was the final GUI library for Java. This assumption turned out to be wrong—Sun made a final attempt, called *JavaFX*. When Oracle bought Sun they changed the original ambitious project (which included a scripting language) into a

[20] https://github.com/BruceEckel/Onjava8-examples/issues

library, and now it appears to be the only UI toolkit getting development effort (see the Wikipedia article on JavaFX)—but even that effort has diminished. JavaFX, too, seems eventually doomed.

Swing is still part of the Java distribution (but it only receives maintenance, no new development), and with Java now an open-source project it should always be available. Also, Swing and JavaFX have some limited interactivity, presumably to aid the transition to JavaFX.

Ultimately, desktop Java never took hold, and never even touched the designers' ambitions. Other pieces, such as JavaBeans, were given much fanfare (and many unfortunate authors spent a lot of effort writing books solely on Swing and even books just on JavaBeans) but never gained any traction. Most usage you'll see for desktop Java is for integrated development environments (IDEs) and some in-house corporate applications. People do develop user interfaces in Java, but it's safe to consider that a niche usage of the language.

If you must learn Swing, it's covered in *Thinking in Java, 4th Edition* (available at www.OnJava8.com[21]), and in books dedicated to the topic.

[21] http://www.OnJava8.com

What is an Object?

> "We do not realize what tremendous power the structure of an habitual language has. It is not an exaggeration to say that it enslaves us through the mechanism of semantic reactions and that the structure which a language exhibits, and impresses upon us unconsciously, is automatically projected upon the world around us."—Alfred Korzybski (1930)

The genesis of the computer revolution was in a machine. Our programming languages thus tend to look like that machine.

But computers are not so much machines as they are mind amplification tools ("bicycles for the mind," as Steve Jobs was fond of saying) and a different kind of expressive medium. As a result, tools are beginning to look less like machines and more like parts of our minds.

Programming languages are the fabric of thought for creating applications. Languages take inspiration from other forms of expression such as writing, painting, sculpture, animation, and filmmaking.

Object-oriented programming (OOP) is one experiment in using the computer as an expressive medium.

Many people feel uncomfortable wading into object-oriented programming without understanding the big picture, so the concepts introduced here give you an overview of OOP. Others might not understand such an overview until they are exposed to the mechanism, becoming lost without seeing code. If you're part of this latter group and are eager to get to the specifics of the language, feel free to jump past this chapter—skipping it now will not prevent you from writing programs or learning the language. However, come back here eventually to fill in your knowledge so you understand why objects are important and how to design with them.

This chapter assumes you have some programming experience, although not necessarily in C. If you need more preparation in programming before tackling this book, work through the free Thinking in C[22] seminar.

[22] https://archive.org/details/ThinkingInC

The Progress of Abstraction

All programming languages are abstractions. It can be argued that the complexity of the problems you're able to solve is directly related to the kind and quality of abstraction. By "kind" I mean, "What is it you are abstracting?" Assembly language[23] is a minimal abstraction of the underlying machine. Many so-called "imperative" languages (such as FORTRAN, BASIC, and C) were themselves abstractions of assembly language. Although they were big improvements, their primary abstraction still requires you to think in terms of the structure of the computer rather than the structure of the problem you are trying to solve. The programmer must establish the association between the machine model (the "solution space," the place where you're implementing that solution, such as a computer) and the model of the problem that is actually solved (the "problem space," the place where the problem exists, such as a business). The effort required to perform this mapping, and the fact it is extrinsic to the programming language, produces programs that are difficult to write and expensive to maintain.

The alternative to modeling the machine is to model the problem you're trying to solve. Early languages such as LISP and APL chose particular views of the world ("All problems are ultimately lists" or "All problems are algorithmic," respectively). Prolog casts all problems into chains of decisions. Languages have been created for constraint-based programming and for programming exclusively by manipulating graphical symbols. Each of these approaches can be a good solution to the particular class of problem they're designed to solve, but when you step outside of that domain they become awkward.

The object-oriented approach goes a step further by providing tools for the programmer to represent elements in the problem space. This representation is general enough that the programmer is not constrained to any particular type of problem. We refer to the elements in the problem space and their representations in the solution space as "objects." (Note that some objects don't have problem-space analogs.) The idea is that the program adapts itself to the lingo of the problem by adding new types of objects. When you read the code describing the solution, you're reading words that also express the problem. This is a more flexible and powerful language abstraction than what we've had before. Thus, OOP describes the problem in terms

[23] https://en.wikipedia.org/wiki/Assembly_language

of the problem, rather than in terms of the computer where the solution will run. There's still a connection, because objects look somewhat like little computers: Each has state and performs operations. This is similar to objects in the real world—they all have characteristics and behaviors.

Alan Kay summarized five basic characteristics of SmallTalk, the first successful object-oriented language and a language that inspired Java. These characteristics represent a pure approach to object-oriented programming:

1. **Everything is an object**. Think of an object as a fancy variable; it stores data, but you can "make requests", asking it to perform operations on itself. You can usually take any conceptual component in the problem you're trying to solve (dogs, buildings, services, etc.) and represent it as an object in your program.
2. **A program is a bunch of objects telling each other what to do by sending messages**. When you "send a message" to an object, it's a request to call a method that belongs to that object.
3. **Each object has its own memory made up of other objects**. Put another way, you create a new kind of object by packaging existing objects. This hides the complexity of a program behind the simplicity of objects.
4. **Every object has a type**. Each object is an *instance* of a *class*, where "class" is (approximately) synonymous with "type." The most important distinguishing characteristic of a class is "What messages can you send to it?"
5. **All objects of a particular type can receive the same messages**. This is a loaded statement, as you will see later. Because an object of type "circle" is also an object of type "shape," a circle is guaranteed to accept shape messages. This means you can write code that talks to shapes and automatically handles anything that fits the description of a shape. This *substitutability* is a foundation of OOP.

Grady Booch offers an even more succinct description of an object:

> *An object has state, behavior and identity*

This means an object can have internal data (which gives it state), methods (to produce behavior), and each object is uniquely distinguished from every other object—that is, every object has a unique address in memory.[24]

[24]This is actually a bit restrictive because objects can conceivably exist in different machines and address spaces, and they can also be stored on disk. In these cases, the identity of the object must be determined by something other than a memory address.

An Object Has an Interface

Aristotle was probably the first to begin a careful study of the concept of *type*; he spoke of "the class of fishes and the class of birds." The idea that all objects, while unique, are also part of a class of objects that have characteristics and behaviors in common was used directly in the first object-oriented language, Simula-67, with its fundamental keyword `class` that introduces a new type into a program.

Simula, as its name implies, was created for developing simulations such as the classic "bank teller problem." In this, you have numerous tellers, customers, accounts, transactions, and units of money—many "objects." Objects that are identical except for their state are grouped together into "classes of objects," and that's where the keyword `class` arose.

Creating abstract data types (classes) is a fundamental concept in object-oriented programming. Abstract data types work almost exactly like built-in types: You create variables of a type (called *objects* or *instances* in object-oriented parlance) and manipulate those variables (called *sending messages* or *requests*; you send a message and the object figures out what to do with it). The members (elements) of each class share some commonality: Every account has a balance, every teller can accept a deposit, etc. At the same time, each member has its own state: Each account has a different balance, each teller has a name. Thus, the tellers, customers, accounts, transactions, etc., can each be represented with a unique entity in the program. This entity is the object, and each object belongs to a particular class that defines its characteristics and behaviors.

So, although what we really do in object-oriented programming is create new data types, virtually all object-oriented programming languages use the "class" keyword. When you see the word "type" think "class" and vice versa.[25]

A class describes a set of objects that have identical characteristics (data elements) and behaviors (functionality), so a class is really a data type because a floating point number, for example, also has a set of characteristics and behaviors. The difference is that a programmer defines a class to fit a problem rather than being forced to use an existing data type that was designed to represent a unit of storage in a machine. You extend the programming language by adding new data types specific to your needs.

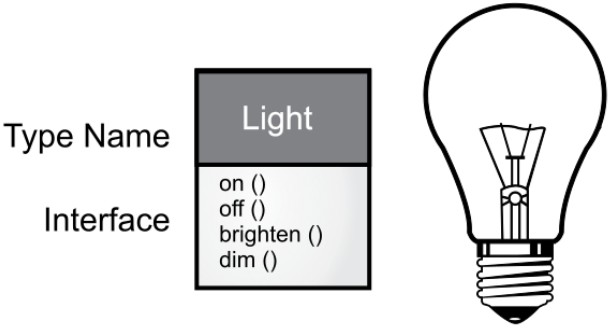
[25]In some cases we make a distinction, stating that type determines the interface while class is a particular implementation of that interface.

The programming system welcomes the new classes and gives them the same care and type checking it gives to built-in types.

The object-oriented approach is not limited to building simulations. Whether or not you agree that any program is a simulation of the system you're designing, OOP techniques help reduce a large set of problems to a simpler solution.

Once a class is established, make as many objects of that class as you like, then manipulate those objects as if they are the elements that exist in your problem. Indeed, one of the challenges of object-oriented programming is creating a one-to-one mapping between the elements in the problem space and objects in the solution space.

How do you get an object to do useful work? You make a request of that object—complete a transaction, draw something on the screen, turn on a switch. Each object accepts only certain requests, defined by its *interface* The type determines the interface. As a simple example, consider a representation for a light bulb:




```
Light lt = new Light();
lt.on();
```

The interface determines the requests you can make for a particular object. However, there must be code somewhere to satisfy that request. This, along with the hidden data, comprises the *implementation*. A type has a method associated with each possible request, and when you make a particular request to an object, that method is called. This process is usually summarized by saying you "send a message" (make a request) to an object, and the object figures out what to do with that message (it executes code).

Here, the name of the class is `Light`, the name of this particular `Light` object is `lt`, and the requests you can make of a `Light` object are to turn it on, turn it off, make it brighter, or make it dimmer. You create a `Light` object by defining a "reference" (`lt`) for that object and calling `new` to request a new object of that type. To send a message to the object, you state the name of the object and connect it to the message request with a period (dot). From the standpoint of the user of a predefined class, that's pretty much all there is to programming with objects.

The preceding diagram follows the format of the *Unified Modeling Language* (UML). Each class is represented by a box, with the type name in the top portion of the box, any *data members* you care to describe in the middle portion of the box, and the *methods* (the functions that belong to this object, which receive any messages you send to that object) in the bottom portion of the box. Often, only the name of the class and the public methods are shown in UML design diagrams, so the middle portion is not shown, as in this case. If you're interested only in the class name, the bottom portion doesn't need to be shown, either.

Objects Provide Services

When trying to develop or understand a program design, an excellent way to think about objects is as "service providers." Your program itself will provide services to the user, and it will accomplish this by using the services offered by other objects. Your goal is to produce (or better, locate in existing code libraries) a set of objects providing the ideal services to solve your problem.

A way to start doing this is to ask, "If I could magically pull them out of a hat, what objects would solve my problem right away?" For example, suppose you are creating a bookkeeping program. You might imagine objects that contain predefined bookkeeping input screens, other objects that perform bookkeeping calculations, and an object that handles printing of checks and invoices on all different kinds of printers. Maybe some of these objects already exist, and for the ones that don't, what would they look like? What services would those objects provide, and what objects would they need to fulfill their obligations? If you keep doing this, you eventually reach a point where you say either, "That object seems simple enough to sit down and write" or "I'm sure that object must exist already." This is a reasonable way to decompose a problem into a set of objects.

Thinking of an object as a service provider has an additional benefit: It helps improve the cohesiveness of the object. *High cohesion* is a fundamental quality of software design: It means the various aspects of a software component (such as an object, although this could also apply to a method or a library of objects) "fit together" well. One problem people have when designing objects is cramming too much functionality into one object. For example, in your check printing module, you might decide you need an object that knows all about formatting and printing. You'll probably discover this is too much for one object, and that what you need is three or more objects. One object might be a catalog of all the possible check layouts, which can be queried for information about how to print a check. One object or set of objects can be a generic printing interface that knows all about different kinds of printers (but nothing about bookkeeping—that is a candidate for buying rather than writing yourself). A third object uses the services of the other two to accomplish the task. Thus, each object has a cohesive set of services it offers. In good object-oriented design, each object does one thing well, but doesn't try to do too much. This not only discovers objects that might be purchased (the printer interface object), but it also produces new objects that might be reused somewhere else (the catalog of check layouts).

Treating objects as service providers is useful not only during the design process, but also when someone else is trying to understand your code or reuse an object. If they can see the value of the object based on what service it provides, it makes it much easier to fit it into the design.

The Hidden Implementation

We can break up the playing field into *class creators* (those who create new data types) and *client programmers*[26] (the class consumers who use the data types in their applications). The goal of the client programmer is to collect a toolbox full of classes to use for rapid application development. The goal of the class creator is to build a class that exposes only what's necessary to the client programmer and keeps everything else hidden. Why? Because if it's hidden, the client programmer can't access it, which means the class creator can change the hidden portion at will without worrying about the impact on anyone else. The hidden portion usually represents the

[26] I'm indebted to my friend Scott Meyers for this term.

tender insides of an object that could easily be corrupted by a careless or uninformed client programmer, so hiding the implementation reduces program bugs.

All relationships need boundaries, respected by all parties involved. When you create a library, you establish a relationship with the client programmer, who is also a programmer, but one who is putting together an application by using your library, possibly to build a bigger library. If all members of a class are available to everyone, the client programmer can do anything with that class and there's no way to enforce rules. Even though you might prefer that the client programmer not directly manipulate some of the members of your class, without access control there's no way to prevent it. Everything's naked to the world.

So the first reason for access control is to keep client programmers' hands off portions they shouldn't touch—parts necessary for the internal operation of the data type but not part of the interface that users need to solve their particular problems. This is actually a service to client programmers because they can easily see what's important and what they can ignore. (Notice this is also a philosophical decision. Some programming languages assume that if a programmer wishes to access the internals, they should be allowed.)

The second reason for access control is to enable the library designer to change the internal workings of the class without worrying about how it will affect the client programmer. For example, you might implement a particular class in a simple fashion to ease development, then later discover you must rewrite it to make it run faster. If the interface and implementation are clearly separated and protected, you can accomplish this easily.

Java has three explicit keywords to set the boundaries in a class: `public`, `private`, and `protected`. These *access specifiers* determine who can use the definitions that follow. `public` means the element is available to everyone. `private` means no one can access that element except you, the creator of the type, inside methods of that type. `private` is a brick wall between you and the client programmer. Anyone trying to access a `private` member gets a compile-time error. `protected` acts like `private`, with the exception that an inheriting class may access `protected` members, but not `private` members. Inheritance is introduced shortly.

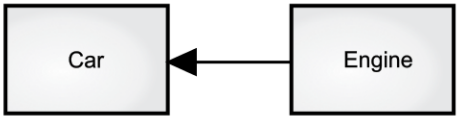
Java also has a "default" access, which comes into play if you don't use one of the aforementioned specifiers. This is usually called *package access* because classes can access the members of other classes in the same `package` (library component), but

outside the package those same members appear to be `private`.

Reusing the Implementation

Once a class is tested, it should (ideally) represent a useful unit of code. This reusability is not nearly so easy to achieve as many hope; it takes experience and insight to produce a reusable object design. But once you have such a design, it begs for reuse. Code reuse is an argument for object-oriented programming languages.

The simplest way to reuse a class is to use an object of that class directly, but you can also place an object of that class inside a new class. Your new class can be made up of any number and type of other objects, in any combination, to produce the desired functionality. Because you *compose* a new class from existing classes, this concept is called *composition* (if composition is dynamic, it's usually called *aggregation*). Composition is often called a *has-a* relationship, as in "A car has an engine."



(This diagram indicates composition with the filled diamond, which states there is one car. I typically use a simpler form: just a line, without the diamond, to indicate an association.[27]

Composition comes with a great deal of flexibility. The member objects of your new class are typically private, making them inaccessible to client programmers who use the class. This means changing those members doesn't disturb existing client code. You can also change the member objects at run time, to dynamically change the behavior of your program. Inheritance, described next, does not have this flexibility because the compiler must place compile-time restrictions on classes created using inheritance.

Inheritance is is often highly emphasized in object-oriented programming. A new programmer can get the impression that inheritance should be used everywhere. This can result in awkward and overly complicated designs. Instead, first look to

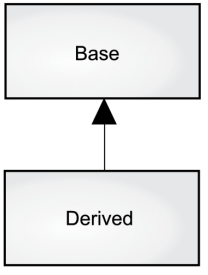
[27]This is enough detail for most diagrams, and you don't need to get specific about whether you're using aggregation or composition.

composition when creating new classes because it is simpler, more flexible, and produces cleaner designs. Once you've had some experience, it is reasonably obvious when you need inheritance.

Inheritance

By itself, the idea of an object is a convenient tool. Objects package data and functionality together by *concept* and represent an appropriate problem-space idea rather than being forced to use the idioms of the underlying machine. These concepts are expressed as fundamental units in the programming language by using the `class` keyword.

It seems a pity, however, to go to all the trouble to create a class, then be forced to create a brand new one that might have similar functionality. It's nicer if we can take the existing class, clone it, then make additions and modifications to the clone. This is effectively what you get with *inheritance*, with the exception that if the original class (called the *base class* or *superclass* or *parent class*) is changed, the modified "clone" (called the *derived class* or *inherited class* or *subclass* or *child class*) also reflects those changes.



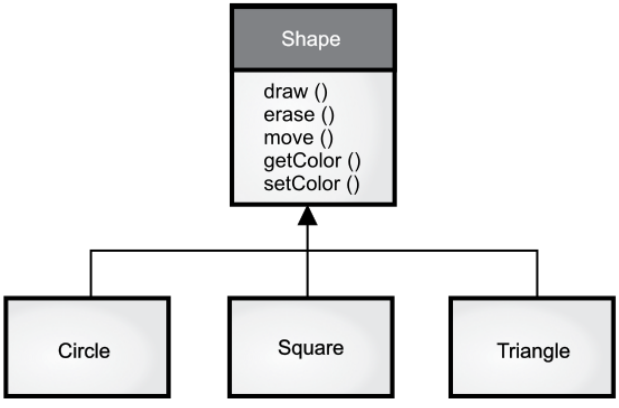
The arrow in this diagram points from the derived class to the base class. As you will see, there is commonly more than one derived class.

A type does more than describe the constraints on a set of objects; it also relates to other types. Two types can have characteristics and behaviors in common, but one type might contain more characteristics than another and might also handle more messages (or handle them differently). Inheritance expresses this similarity through the concept of base types and derived types. A base type contains all characteristics

and behaviors shared among the types derived from it. You create a base type to represent the core of your ideas. From the base type, you derive other types to express the different ways this core can be realized.

For example, a trash-recycling machine sorts pieces of trash. The base type is "trash." Each piece of trash has a weight, a value, and so on, and can be shredded, melted, or decomposed. From this, more specific types of trash are derived with additional characteristics (a bottle has a color, a steel can is magnetic) or behaviors (you can crush an aluminum can). In addition, some behaviors can be different (the value of paper depends on its type and condition). Using inheritance, you build a type hierarchy that expresses the problem you're trying to solve in terms of its types.

A second example is the common "shape" example, perhaps used in a computer-aided design system or game simulation. The base type is "shape," and each shape has a size, a color, a position, and so on. Each shape can be drawn, erased, moved, colored, etc. From this, specific types of shapes are derived (inherited)—circle, square, triangle, and so on—each of which can have additional characteristics and behaviors. Certain shapes can be flipped, for example. Some behaviors might be different, such as when you calculate the area of a shape. The type hierarchy embodies both the similarities and differences between the shapes.



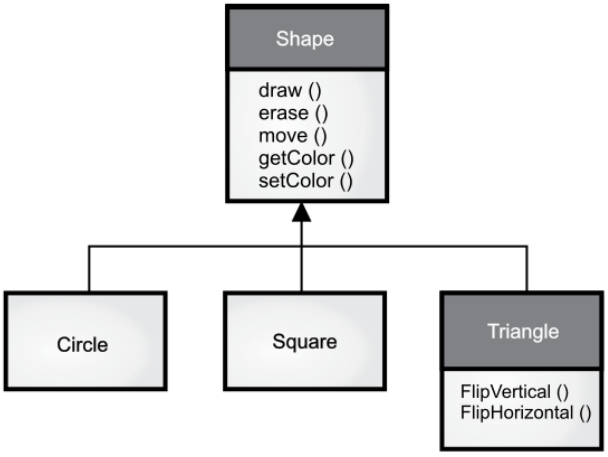
Casting the solution in the same terms as the problem is useful because you don't need intermediate models to get from a description of the problem to a description of the solution. With objects, the type hierarchy is an important aspect of the model, so you go directly from the description of the system in the real world to the description

of the system in code. Indeed, sometimes people who are trained to look for complex solutions have difficulty with the simplicity of object-oriented design.

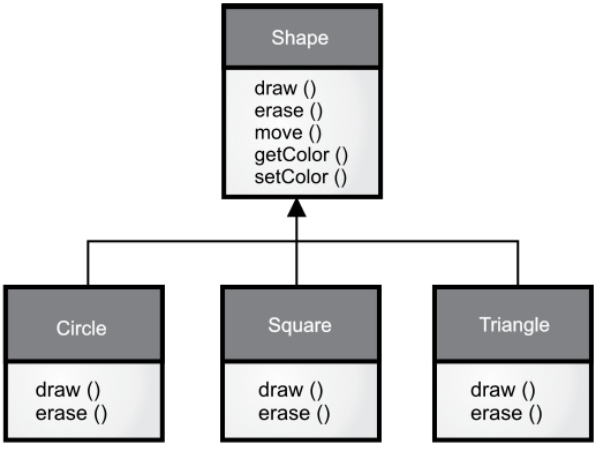
Inheriting from an existing type creates a new type. This new type contains not only all the members of the existing type (although the `private` ones are hidden away and inaccessible), but more importantly it duplicates the interface of the base class. That is, all messages accepted by base-class objects are also accepted by derived-class objects. We know the type of a class by the messages it accepts, so the derived class *is the same type as the base class*. In the previous example, "A circle is a shape." This type equivalence via inheritance is one of the fundamental gateways in understanding the meaning of object-oriented programming.

Because both base class and derived class have the same fundamental interface, there must be some implementation to go along with that interface. That is, there must be executable code when an object receives a particular message. If you inherit a class and don't do anything else, the methods from the base-class interface come right along into the derived class. That means objects of the derived class have not only the same type, they also have the same behavior, which isn't particularly interesting.

There are two ways to differentiate your new derived class from the original base class. The first is straightforward: add brand new methods to the derived class. These new methods are not part of the base-class interface. This means the base class didn't do as much as you wanted, so you added more methods. This simple and primitive use for inheritance is, at times, the perfect solution to your problem. However, look closely for the possibility that your base class might also need these additional methods (or that you should be using composition instead). This process of discovery and iteration of your design happens regularly in object-oriented programming.



Although inheritance can sometimes imply (especially in Java, where the keyword for inheritance is `extends`) that you are going to add new methods to the interface, that's not necessarily true. The second and more important way to differentiate your new class is to change the behavior of an existing base-class method. This is called *overriding* that method.



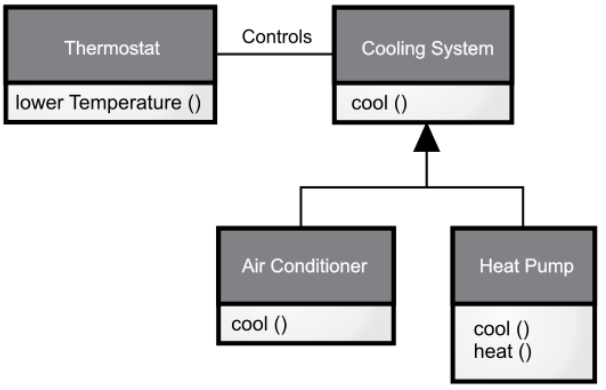
To override a method, you create a new definition for the method in the derived class. You're saying, "I'm using the same interface method here, but I want it to do something different for my new type."

Is-a vs. Is-Like-a Relationships

There's a certain debate that can occur about inheritance: Should inheritance override *only* base-class methods (and not add new methods that aren't in the base class)? This would mean that the derived class is *exactly* the same type as the base class because it has exactly the same interface. As a result, you can perfectly substitute an object of the derived class for an object of the base class. This can be thought of as *pure substitution*, and it's often called the *substitution principle*[28]. In a sense, this is the ideal way to treat inheritance. We often refer to the relationship between the base class and derived classes in this case as an *is-a* relationship, because you can say, "A circle *is a* shape." A test for inheritance is to see if the is-a relationship makes sense for your classes.

Sometimes you add new interface elements to a derived type, thus extending the interface. The new type can still substitute for the base type, but the substitution isn't perfect because your new methods are not accessible from the base type. This can be described as an *is-like-a* relationship (my term). The new type has the interface of the old type but it also contains other methods, so you can't really say it's exactly the same. For example, consider an air conditioner. Suppose your house is wired with all the controls for cooling; that is, it has an interface that to control cooling. Imagine that the air conditioner breaks down and you replace it with a heat pump, which can both heat and cool. The heat pump *is-like-an* air conditioner, but it can do more. Because the control system of your house is designed only to control cooling, it is restricted to communication with the cooling part of the new object. The interface of the new object is extended, and the existing system only knows about the original interface.

[28]Or *Liskov Substitution Principle*, after Barbara Liskov who first described it.



Once you see this design it becomes clear that the base class "cooling system" is not general enough, and should be renamed to "temperature control system" so it can also include heating—at which point the substitution principle will work. However, this diagram shows what can happen with design in the real world.

When you see the substitution principle it's easy to feel like this approach (pure substitution) is the only way to do things, and in fact it *is* nice if your design works out that way. But you'll find there are times when it's equally clear you must add new methods to the interface of a derived class (extension). With inspection both cases should be reasonably obvious.

Interchangeable Objects with Polymorphism

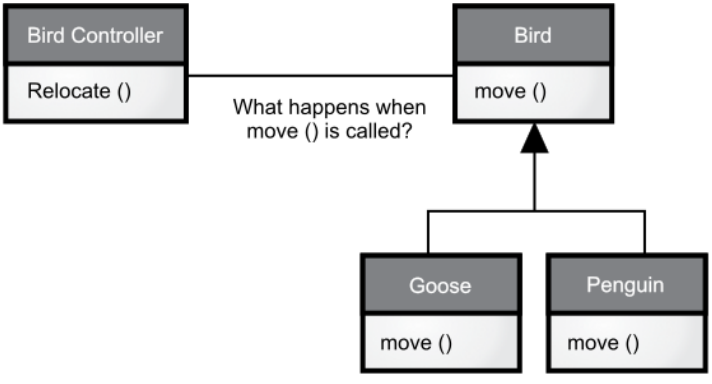
When dealing with type hierarchies, you often treat an object not as the specific type it is, but as its base type. This way you can write code that doesn't depend on specific types. In the shape example, methods manipulate generic shapes, unconcerned about whether they're circles, squares, triangles, or some shape that hasn't even been defined yet. All shapes can be drawn, erased, and moved, so these methods send a message to a shape object without worrying how the object copes with the message.

Such code is unaffected by the addition of new types, and adding new types is a common way to extend an object-oriented program to handle new situations. For example, you can derive a new subtype of shape called "pentagon" without modifying methods that deal only with generic shapes. This ability to easily extend a

design by deriving new subtypes is one of the essential ways to encapsulate change. This improves designs while reducing the cost of software maintenance.

There's a problem when attempting to treat derived-type objects as their generic base types (circles as shapes, bicycles as vehicles, cormorants as birds, etc.). If a method tells a generic shape to draw itself, or a generic vehicle to steer, or a generic bird to move, the compiler cannot know at compile time precisely what piece of code is executed. That's the whole point—when the message is sent, the programmer doesn't *want* to know what piece of code is executed; the draw method can be applied equally to a circle, a square, or a triangle, and the object will execute the proper code depending on its specific type.

If you don't need to know what piece of code is executed, when you add a new subtype, the code it executes can be different without requiring changes to the code that calls it. But what does the compiler do when it cannot know precisely what piece of code is executed? For example, in the following diagram the `BirdController` object just works with generic `Bird` objects and does not know what exact type they are. This is convenient from `BirdController`'s perspective because it doesn't require special code to determine the exact type of `Bird` it's working with or that `Birds` behavior. So how does it happen that, when `move()` is called while ignoring the specific type of `Bird`, the right behavior will occur (a `Goose` walks, flies, or swims, and a `Penguin` walks or swims)?



The answer is the primary twist of inheritance: The compiler cannot make a function call in the traditional sense. The function call generated by a non-OOP compiler produces what is called *early binding*, a term you might not have heard because

you've never thought about it any other way. It means the compiler generates a call to a specific function name, which resolves to the absolute address of the code to be executed. With inheritance, the program cannot determine the address of the code until run time, so some other scheme is necessary when a message is sent to an object.

To solve the problem, object-oriented languages use the concept of *late binding*. When you send a message to an object, the code called isn't determined until run time. The compiler does ensure that the method exists and performs type checking on the arguments and return value, but it doesn't know the exact code to execute.

To perform late binding, Java uses a special bit of code in lieu of the absolute call. This code calculates the address of the method body, using information stored in the object (this process is covered in great detail in the Polymorphism chapter). Thus, each object behaves differently according to the contents of that special bit of code. When you send a message to an object, the object actually does figure out what to do with that message.

In some languages you must explicitly grant a method the flexibility of late-binding properties. For example, C++ uses the `virtual` keyword. In such languages, methods are *not* dynamically bound by default. In Java, dynamic binding is the default behavior and you don't need extra keywords to produce polymorphism.

Consider the shape example. The family of classes (all based on the same uniform interface) was diagrammed earlier in this chapter. To demonstrate polymorphism, we write a single piece of code that ignores specific details of type and talks only to the base class. That code is *decoupled* from type-specific information and thus is simpler to write and easier to understand. And, if a new type—a `Hexagon`, for example—is added through inheritance, code works just as well for the new type of `Shape` as it did on the existing types. Thus, the program is *extensible*.

If you write a method in Java (you will soon learn how):

```
void doSomething(Shape shape) {
  shape.erase();
  // ...
  shape.draw();
}
```

This method speaks to any `Shape`, so it is independent of the specific type of object it's drawing and erasing. If some other part of the program uses the `doSomething()` method:

```
Circle circle = new Circle();
Triangle triangle = new Triangle();
Line line = new Line();
doSomething(circle);
doSomething(triangle);
doSomething(line);
```

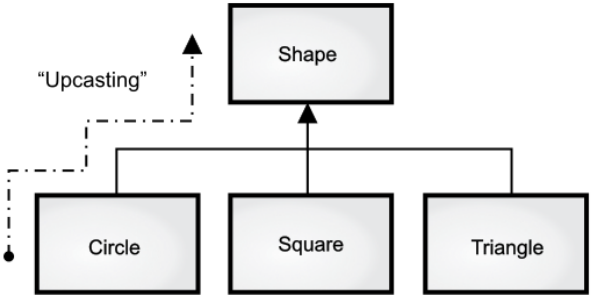
The calls to `doSomething()` automatically work correctly, regardless of the exact type of the object.

This is a rather amazing trick. Consider the line:

```
doSomething(circle);
```

What's happening here is that a `Circle` is passed into a method that expects a `Shape`. Because a `Circle` *is* a `Shape` it is treated as such by `doSomething()`. That is, any message that `doSomething()` can send to a `Shape`, a `Circle` can accept. It is a completely safe and logical thing to do.

We call this process of treating a derived type as though it were its base type *upcasting*. The name *cast* is used in the sense of casting into a mold and the *up* comes from the way the inheritance diagram is typically arranged, with the base type at the top and the derived classes fanning out downward. Thus, casting to a base type is moving up the inheritance diagram: "upcasting."



An object-oriented program contains upcasting somewhere, because that's how you decouple yourself from knowing the exact type you're working with. Look at the code in `doSomething()`:

```
shape.erase();
// ...
shape.draw();
```

Notice it doesn't say, "If you're a `Circle`, do this, if you're a `Square`, do that, etc." If you write that kind of code, which checks for all the possible types a `Shape` can actually be, it's messy and you must change it every time you add a new kind of `Shape`. Here, you just say, "You're a shape, I know you can `erase()` and `draw()` yourself, do it, and take care of the details correctly."

What's impressive about the code in `doSomething()` is that, somehow, the right thing happens. Calling `draw()` for `Circle` causes different code to be executed than calling `draw()` for a `Square` or a `Line`, but when the `draw()` message is sent to an anonymous `Shape`, the correct behavior occurs based on the actual type of the `Shape`. This is amazing because when the Java compiler is compiling the code for `doSomething()`, it cannot know exactly what types it is dealing with. Ordinarily, you'd expect it to end up calling the version of `erase()` and `draw()` for the base class `Shape`, and not for the specific `Circle`, `Square`, or `Line`. And yet the right thing happens—that's polymorphism. The compiler and runtime system handle the details; all you must know is it happens, and more importantly, how to design with it. When you send a message to an object, the object will do the right thing, even when upcasting is involved.

The Singly-Rooted Hierarchy

An OOP issue that has become especially prominent since the introduction of C++ is whether all classes should by default be inherited from a single base class. In Java (as with virtually all other OOP languages *except* for C++) the answer is yes, and the name of this ultimate base class is simply `Object`.

There are many benefits to a singly-rooted hierarchy. All objects have a common interface, so they are all ultimately the same fundamental type. The alternative (provided by C++) is that you don't know that everything is the same basic type. From a backward-compatibility standpoint this fits the model of C better and can be thought of as less restrictive, but for full-on object-oriented programming you must build your own hierarchy to provide the same convenience that's built into other OOP languages. And in any new class library you acquire, some other incompatible

interface is used. It requires effort to work the new interface into your design. Is the extra "flexibility" of C++ worth it? If you need it—if you have a large investment in C—it's quite valuable. If you're starting from scratch, alternatives such as Java can be more productive.

A singly rooted hierarchy makes it much easier to implement a *garbage collector*, one of the fundamental improvements of Java over C++. And since information about the type of an object is guaranteed to be in all objects, you'll never end up with an object whose type you cannot determine. This is especially important with system-level operations, such as *exception handling* (a language mechanism for reporting errors), and to allow greater flexibility in programming.

Collections

In general, you don't know how many objects you need to solve a particular problem, or how long they will last. You also don't know how to store those objects. How can you know how much space to create if that information isn't known until run time?

The solution to most problems in object-oriented design seems flippant: You create another type of object. The new type of object that solves this particular problem holds references to other objects. You can also do the same thing with an *array*, available in most languages. But this new object, generally called a *collection* (also called a *container*, but the Java libraries use "collection" almost universally), will expand itself whenever necessary to accommodate everything you place inside it. You don't need to know how many objects you're going to hold in a collection—just create a collection object and let it take care of the details.

Fortunately, a good OOP language comes with a set of collections as part of the package. In C++, it's part of the Standard C++ Library and is often called the *Standard Template Library* (STL). SmallTalk has a very complete set of collections. Java also has numerous collections in its standard library. In some libraries, one or two generic collections is considered good enough for all needs, and in others (Java, for example) the library has different types of collections for different needs: several different kinds of `List` classes (to hold sequences), `Maps` (also known as *associative arrays*, to associate objects with other objects), `Sets` (to hold one of each type of object), and more components such as queues, trees, stacks, etc.

From a design standpoint, all you really want is a collection you can manipulate to solve your problem. If a single type of collection satisfied all of your needs, we wouldn't need different kinds. There are two reasons you need a choice of collections:

1. Collections provide different types of interfaces and external behavior. Stacks and queues are different from sets and lists. One of these might provide a more flexible solution to your problem than another.
2. Different implementations have different efficiencies for certain operations. For example, there are two basic types of `List`: `ArrayList` and `LinkedList`. Both are simple sequences that can have identical interfaces and external behaviors. But some operations have significantly different costs. Randomly accessing elements in an `ArrayList` is a constant-time operation; it takes the same amount of time regardless of the element you select. However, in a `LinkedList` it is expensive to move through the list to randomly select an element, and it takes longer to find an element that is farther down the list. On the other hand, to insert an element in the middle of a sequence, it's cheaper in a `LinkedList` than in an `ArrayList`. These and other operations have different efficiencies depending on the underlying structure of the sequence. You might start building your program with a `LinkedList` and, when tuning for performance, change to an `ArrayList`. Because of the abstraction via the interface `List`, you can change from one to the other with minimal impact on your code.

Parameterized Types (Generics)

Before Java 5, collections held the one universal type in Java: `Object`. The singly rooted hierarchy means everything is an `Object`, so a collection that holds `Objects` can hold anything.[29] This made collections easy to reuse.

To use such a collection, you add object references to it and later ask for them back. But the collection holds only `Objects`, so when you add an object reference into the collection it is upcast to `Object`, thus losing its character. When fetching it back, you get an `Object` reference, and not a reference to the type you put in. How do you turn it back into something with the specific type of the object you put into the collection?

Here, the cast is used again, but this time you're not casting up the inheritance hierarchy to a more general type. Instead, you cast down the hierarchy to a more

[29] They do not hold primitives, but *autoboxing* simplifies this restriction somewhat. This is discussed in detail later in the book.

specific type, so this manner of casting is called *downcasting*. With upcasting, you know that a `Circle` is a type of `Shape` so it's safe to upcast, but you don't know that an `Object` is necessarily a `Circle` or a `Shape` so it's not safe to downcast unless you determine extra type information about that object.

It's not completely dangerous because if you downcast to the wrong type you'll get a runtime error called an *exception*, described shortly. When you fetch `Object` references from a collection, however, you need some way to remember exactly what they are in order to perform a proper downcast.

Downcasting and the associated runtime checks require extra time for the running program and extra effort from the programmer. Wouldn't it make sense to somehow create the collection so it knows the types it holds, eliminating the need for the downcast and a possible mistake? The solution is called a *parameterized type* mechanism. A parameterized type is a class that the compiler can automatically customize to work with particular types. For example, with a parameterized collection, the compiler can customize that collection so it accepts only `Shapes` and fetches only `Shapes`.

Java 5 added parameterized types, called *generics*, which is a major feature. You'll recognize generics by the angle brackets with types inside; for example, you can create an `ArrayList` to hold `Shape` like this:

```
ArrayList<Shape> shapes = new ArrayList<>();
```

There have also been changes to many of the standard library components to take advantage of generics. You will see that generics have an impact on much of the code in this book.

Object Creation & Lifetime

One critical issue when working with objects is the way they are created and destroyed. Each object requires resources, most notably memory, to exist. When an object is no longer needed it must be cleaned up so these resources are released for reuse. In simple programming situations the question of how an object is cleaned up doesn't seem too challenging: You create the object, use it for as long as it's needed, then it should be destroyed. However, it's not hard to encounter situations that are more complex.

Suppose, for example, you are designing a system to manage air traffic for an airport. (The same model might also work for managing crates in a warehouse, or a video rental system, or a kennel for boarding pets.) At first it seems simple: Make a collection to hold airplanes, then create a new airplane and place it in the collection for each airplane that enters the air-traffic-control zone. For cleanup, simply clean up the appropriate airplane object when a plane leaves the zone.

But suppose you have some other system to record data about the planes; perhaps data that doesn't require such immediate attention as the main controller function. Maybe it's a record of the flight plans of all the small planes that leave the airport. So you have a second collection of small planes, and whenever you create a plane object you also put it in this second collection if it's a small plane. Then some background process performs operations on the objects in this collection during idle moments.

Now the problem is more difficult: How can you possibly know when to destroy the objects? When you're done with the object, some other part of the system might not be. This same problem can arise in a number of other situations, and in programming systems (such as C++) where you must explicitly delete an object this can become quite complex.

Where is the data for an object and how is the lifetime of the object controlled? C++ takes the approach that efficiency is the most important issue, so it gives the programmer a choice. For maximum runtime speed, the storage and lifetime can be determined while the program is written, by placing the objects on the stack (these are sometimes called *automatic* or *scoped* variables) or in the static storage area. This places a priority on the speed of storage allocation and release, and this control can be very valuable in certain situations. However, you sacrifice flexibility because you must know the exact quantity, lifetime, and type of objects while you're writing the program. If you are trying to solve a more general problem such as computer-aided design, warehouse management, or air-traffic control, this is too restrictive.

The second approach is to create objects dynamically in a pool of memory called the *heap*. In this approach, you don't know until run time how many objects you need, what their lifetime is, or what their exact type is. Those are determined at the spur of the moment while the program is running. If you need a new object, you simply make it on the heap when you need it. Because the storage is managed dynamically, at run time, the amount of time required to allocate storage on the heap can be longer than the time to create storage on the stack (but not necessarily). Creating storage on

the stack is often a single assembly instruction to move the stack pointer down and another to move it back up. The time to create heap storage depends on the design of the storage mechanism.

The dynamic approach makes the generally logical assumption that objects tend to be complicated, so the extra overhead of finding storage and releasing that storage will not have an important impact on the creation of an object. In addition, the greater flexibility is essential to solve general programming problems.

Java uses dynamic memory allocation, exclusively.[30] Every time you create an object, you use the `new` operator to build a dynamic instance of that object.

There's another issue, however, and that's the lifetime of an object. With languages that allow objects to be created on the stack, the compiler determines how long the object lasts and automatically destroys it. However, if you create it on the heap the compiler has no knowledge of its lifetime. In a language like C++, you must determine programmatically when to destroy the object, which can lead to memory leaks if you don't do it correctly. Java is built upon a *garbage collector* which automatically discovers when an object is no longer in use and releases it. A garbage collector is much more convenient because it reduces the number of issues you must track and the code you must write. Thus, the garbage collector provides a much higher level of insurance against the insidious problem of memory leaks, which has brought many a C++ project to its knees.

With Java, the garbage collector is designed to take care of the problem of releasing memory (although this doesn't include other aspects of cleaning up an object). The garbage collector "knows" when an object is no longer in use, and automatically releases the memory for that object. This, combined with the fact that all objects are inherited from the single root class `Object` and you can create objects only one way—on the heap—makes the process of programming in Java much simpler than programming in C++. You have far fewer decisions to make and hurdles to overcome.

Exception Handling: Dealing with Errors

Since the beginning of programming languages, error handling has been especially difficult. Because it's so hard to design a good error-handling scheme, many languages ignore the issue, passing the problem on to library designers who come up

[30] Primitive types, which you'll learn about later, are a special case.

with halfway measures that work in many situations but can easily be circumvented, generally by just ignoring errors. A major problem with most error-handling schemes is that they rely on programmers to follow an agreed-upon convention that is not enforced by the language. If the programmer is not vigilant—often the case if they are in a hurry—these schemes can easily be forgotten.

Exception handling wires error handling directly into the programming language and sometimes even the operating system. An exception is an object that is "thrown" from the site of the error and can be "caught" by an appropriate exception handler designed for that particular type of error. It's as if exception handling is a different, parallel path of execution, taken when things go wrong. Because it uses a separate execution path, it doesn't interfere with your normally executing code. This can make that code simpler to write because you aren't constantly forced to check for errors. In addition, a thrown exception is unlike an error value returned from a method or a flag set by a method to indicate an error condition—these can be ignored. An exception cannot be ignored, so it's guaranteed to be dealt with at some point. Finally, exceptions provide a way to reliably recover from a bad situation. Instead of just exiting the program, you are sometimes able to set things right and restore execution, which produces more robust programs.

Java's exception handling stands out among programming languages, because in Java, exception handling was wired in from the beginning and you're forced to use it. It is the single acceptable way to report errors. If you don't write your code to properly handle exceptions, you'll get a compile-time error message. This guaranteed consistency can sometimes make error handling much easier.

It's worth noting that exception handling isn't an object-oriented feature, although in object-oriented languages the exception is normally represented by an object. Exception handling existed before object-oriented languages.

Summary

A procedural program contains data definitions and function calls. To find the meaning of such a program, you must work at it, looking through the function calls and low-level concepts to create a model in your mind. This is the reason we need intermediate representations when designing procedural programs—by themselves,

these programs tend to be confusing because the terms of expression are oriented more toward the computer than to the problem you're solving.

Because OOP adds many new concepts on top of what you find in a procedural language, your natural assumption might be that the resulting Java program is far more complicated than the equivalent procedural program. Here, you'll be pleasantly surprised: A well-written Java program is generally simpler and easier to understand than a procedural program. What you'll see are the definitions of the objects that represent concepts in your problem space (rather than the issues of the computer representation) and messages sent to those objects to indicate activities in that space. One of the delights of object-oriented programming is that, with a well-designed program, it's easy to understand the code by reading it. Usually, there's a lot less code as well, because many problems are solved by reusing existing library code.

OOP and Java might not be for everyone. It's important to evaluate your own needs and decide whether Java will optimally satisfy those needs, or if you might be better off with another programming system (perhaps the one you're currently using). If your needs are very specialized for the foreseeable future and you have specific constraints that might not be satisfied by Java, you owe it to yourself to investigate the alternatives (in particular, I recommend looking at Python[31]). If you still choose Java as your language, you'll at least understand what the options were and have a clear vision of why you took that direction.

[31] https://www.python.org/

Installing Java and the Book Examples

In which we provision ourselves for the journey.

Before you can begin learning the language, you must install Java and the book's source-code examples. Because it is possible for a "dedicated beginner" to learn programming from this book, I explain the process in detail, assuming you haven't previously used the computer's command-line shell. If you have, you can skip forward to the installation instructions.

If any terminology or processes described here are still not clear to you, you can usually find explanations or answers through Google[32]. For more specific issues or problems, try StackOverflow[33]. Sometimes you can find installation instructions on YouTube[34].

Editors

To create and modify Java program files—the code listings shown in this book—you need a program called an *editor*. You'll also need an editor to make changes to your system configuration files, which is sometimes required during installation.

Programming editors vary from heavyweight *Integrated Development Environments* (IDEs, like Eclipse, NetBeans and IntelliJ IDEA) to more basic text manipulation applications. If you already have an IDE and are comfortable with it, feel free to use that for this book, but in the interest of keeping things simple, I recommend the *Atom* editor. Find it at atom.io[35].

[32] https://www.google.com/
[33] http://stackoverflow.com/
[34] https://www.youtube.com/
[35] https://atom.io/

Atom is free and open-source, is very simple to install, works on all platforms (Windows, Mac and Linux), and has a built-in Java mode that is automatically invoked when you open a Java file. It isn't a heavy-duty IDE so it doesn't get confusing, which is ideal for this book. On the other hand, it has some handy editing features that you'll probably come to love. More details are on their site.

There are many other editors; these are a subculture unto themselves and people even get into heated arguments about their merits. If you find one you like better, it's not too hard to change. The important thing is to choose one and get comfortable with it.

The Shell

If you haven't programmed before, you might be unfamiliar with your operating system *shell* (also called the *command prompt* in Windows). The shell harkens back to the early days of computing when everything happened by typing commands and the computer responded by displaying responses; it was all text-based.

Although it can seem primitive in the age of graphical user interfaces, a shell provides a surprising number of valuable features. We'll use the shell regularly in this book, both as part of the installation process and to run Java programs.

Starting a Shell

Mac: Click on the *Spotlight* (the magnifying-glass icon in the upper-right corner of the screen) and type "terminal." Click on the application that looks like a little TV screen (you might also be able to hit "Return"). This starts a shell in your home directory.

Windows: First, start the Windows Explorer to navigate through your directories:

- *Windows 7*: click the "Start" button in the lower left corner of the screen. In the Start Menu search box area type "explorer" then press the "Enter" key.
- *Windows 8*: click Windows+Q, type "explorer" then press the "Enter" key.
- *Windows 10*: click Windows+E.

Once the Windows Explorer is running, move through the folders on your computer by double-clicking on them with the mouse. Navigate to the desired folder. Now click the file tab at the top left of the Explorer window and select "Open command prompt." This opens a shell in the destination directory.

Linux: To open a shell in your home directory:

- *Debian*: Press Alt+F2. In the dialog that pops up, type 'gnome-terminal'
- *Ubuntu*: Either right-click on the desktop and select 'Open Terminal', or press Ctrl+Alt+T
- *Redhat*: Right-click on the desktop and select 'Open Terminal'
- *Fedora*: Press Alt+F2. In the dialog that pops up, type 'gnome-terminal'

Directories

Directories are one of the fundamental elements of a shell. Directories hold files, as well as other directories. Think of a directory as a tree with branches. If `books` is a directory on your system and it has two other directories as branches, for example `math` and `art`, we say you have a directory `books` with two *subdirectories* `math` and `art`. We refer to them as `books/math` and `books/art` because `books` is their *parent* directory. Note that Windows uses backslashes rather than forward slashes to separate the parts of a directory.

Basic Shell Operations

The shell operations I show here are approximately identical across operating systems. For the purposes of this book, here are the essential operations in a shell:

- **Change directory**: Use `cd` followed by the name of the directory where you want to move, or `cd ..` if you want to move up a directory. If you want to move to a different directory while remembering where you came from, use `pushd` followed by the different directory name. Then, to return to the previous directory, just say `popd`.
- **Directory listing**: `ls` (`dir` in Windows) displays all the files and subdirectory names in the current directory. Use the wildcard `*` (asterisk) to narrow your search. For example, if you want to list all the files ending in ".java," you say `ls *.java` (Windows: `dir *.java`). If you want to list the files starting with "F" and ending in ".java," you say `ls F*.java` (Windows: `dir F*.java`).

- **Create a directory**: use the `mkdir` ("make directory") command (Windows: `md`), followed by the name of the directory you want to create. For example, `mkdir books` (Windows: `md books`).
- **Remove a file**: Use `rm` ("remove") followed by the name of the file you wish to remove (Windows: `del`). For example, `rm somefile.java` (Windows: `del somefile.java`).
- **Remove a directory**: use the `rm -r` command to remove the files in the directory and the directory itself (Windows: `deltree`). For example, `rm -r books` (Windows: `deltree books`).
- **Repeat a command**: The "up arrow" on all three operating systems moves through previous commands so you can edit and repeat them. On Mac/Linux, `!!` repeats the last command and `!n` repeats the nth command.
- **Command history**: Use `history` in Mac/Linux or press the F7 key in Windows. This gives you a list of all the commands you've entered. Mac/Linux provides numbers to refer to when you want to repeat a command.
- **Unpacking a zip archive**: A file name ending with `.zip` is an archive containing other files in a compressed format. Both Linux and Mac have command-line `unzip` utilities, and you can install a command-line `unzip` for Windows via the Internet. However, in all three systems the graphical file browser (Windows Explorer, the Mac Finder, or Nautilus or equivalent on Linux) will browse to the directory containing your zip file. Then right-mouse-click on the file and select "Open" on the Mac, "Extract Here" on Linux, or "Extract all ..." on Windows.

To learn more about your shell, search Wikipedia for Windows Shell[36] or, for Mac/Linux, Bash Shell[37].

Installing Java

To compile and run the examples, you must first install the *Java development kit*. In this book we use JDK 8 (Java 1.8), although newer versions of Java are intended to be backward-compatible, so the examples *should* work with any version of Java as long as it's *at least* JDK 8. If you want to choose a newer version, I recommend a *Long-Term Support* version—at this writing, the most recent LTS version is JDK 11.

[36] https://en.wikipedia.org/wiki/Windows_shell
[37] https://en.wikipedia.org/wiki/Bash_(Unix_shell)

Windows

1. Follow the instructions at this link to Install Chocolatey[38].
2. At a shell prompt, type: `choco install jdk8`. This takes some time, but when it's finished Java is installed and the necessary environment variables are set.

Macintosh

The Mac comes with a much older version of Java that won't work for the examples in this book, so you must first update it to Java 8. You will need administration rights to perform these steps.

1. Follow the instructions at this link to Install HomeBrew[39]. Then at a shell prompt, execute `brew update` to make sure you have the latest changes.
2. At a shell prompt, execute `brew cask install java`.

Once HomeBrew and Java are installed, all other activities described in this book can be accomplished within a guest account, if that suits your needs.

Linux

Use the standard package installer with the following shell commands:

Ubuntu/Debian:

1. `sudo apt-get update`
2. `sudo apt-get install default-jdk`

Fedora/Redhat:

1. `su-c "yum install java-1.8.0-openjdk"`

Verify Your Installation

Open a new shell and type:

[38] https://chocolatey.org/
[39] http://brew.sh/

```
java -version
```

You should see something like the following (Version numbers and actual text will vary):

```
java version "1.8.0_112"
Java(TM) SE Runtime Environment (build 1.8.0_112-b15)
Java HotSpot(TM) 64-Bit Server VM (build 25.112-b15, mixed mode)
```

If you see a message that the command is not found or not recognized, review the installation instructions in this chapter. If you still can't get it to work, check StackOverflow[40].

Installing and Running the Book Examples

Once you have Java installed, the process to install and run the book examples is the same for all platforms:

1. Download the book examples from the GitHub Repository[41].
2. `unzip` (as described in Basic Shell Operations) the downloaded file into the directory of your choice.
3. Use the Windows Explorer, the Mac Finder, or Nautilus or equivalent on Linux to browse to the directory where you unzipped `OnJava8-Examples`, and open a shell there.
4. If you're in the right directory, you should see files named `gradlew` and `gradlew.bat` in that directory, along with numerous other files and directories. The directories correspond to the chapters in the book.
5. At the shell prompt, type `gradlew run` (Windows) or `./gradlew run` (Mac/Linux).

The first time you do this, Gradle will install itself and numerous other packages, so it will take some time. After everything is installed, subsequent builds and runs are faster.

Note you must be connected to the Internet the first time you run `gradlew` so that Gradle can download the necessary packages.

[40] http://stackoverflow.com/search?q=installing+java
[41] https://github.com/BruceEckel/OnJava8-Examples/archive/master.zip

Basic Gradle Tasks

There are a large number of Gradle tasks automatically available with this book's build. Gradle uses an approach called *convention over configuration* which results in the availability of many tasks even if you're only trying to accomplish something very basic. Some of the tasks that "came along for the ride" with this book are inappropriate or don't successfully execute. Here is a list of the Gradle tasks you will typically use:

- `gradlew compileJava`: Compiles all the Java files in the book *that can be compiled* (some files don't compile, to demonstrate incorrect language usage).
- `gradlew run`: First compiles, then executes all the Java files in the book *that can be executed* (some files are library components).
- `gradlew test`: Executes all the *unit tests* (you'll learn about these in Validating Your Code).
- `gradlew` *chapter*:*ExampleName*: Compiles and runs a specific example program. For instance, `gradlew objects:HelloDate`.

Objects Everywhere

> "If we spoke a different language, we would perceive a somewhat different world."—*Wittgenstein*

Although it is based on C++, Java is more of a "pure" object-oriented language. Both C++ and Java are hybrid languages, but in Java the designers felt that the hybridization was not as important as it was in C++. A hybrid language allows multiple programming styles; the reason C++ is hybrid is to support backward compatibility with the C language. Because C++ is a superset of the C language, it includes many of that language's undesirable features, which can make some aspects of C++ overly complicated.

The Java language assumes you're *only* writing object-oriented programs. Before you can begin you must shift your mindset into an object-oriented world. In this chapter you'll see the basic components of a Java program and learn that (almost) everything in Java is an object.

You Manipulate Objects with References

> "What's in a name? That which we call a rose, by any other word would smell as sweet."—*Shakespeare*, Romeo & Juliet

Every programming language manipulates elements in memory. Sometimes the programmer must be constantly aware of that manipulation. Do you manipulate the element directly, or use an indirect representation that requires special syntax (for example, pointers in C or C++)?

Java simplifies the issue by considering everything an object, using a single consistent syntax. Although you *treat* everything as an object, the identifier you manipulate

is actually a "reference" to an object.[42] You might imagine a television (the object) and a remote control (the reference). As long as you're holding this reference, you have a connection to the television, but when someone says, "Change the channel" or "Lower the volume," what you're manipulating is the reference, which in turn modifies the object. To move around the room and still control the television, you take the remote/reference with you, not the television.

Also, the remote control can stand on its own, with no television. That is, just because you have a reference doesn't mean there's necessarily an object connected to it. To hold a word or sentence, you create a `String` reference:

```
String s;
```

But here you've created *only* the reference, not an object. If you now decide to send a message to `s`, you get an error because `s` isn't actually attached to anything (there's no television). A safer practice is to always initialize a reference when you create it:

```
String s = "asdf";
```

This uses a special Java feature: Strings can be initialized with quoted text. You must use a more general type of initialization for other types of objects.

You Must Create All the Objects

The point of a reference is to connect it to an object. You usually create objects with the `new` operator. The keyword `new` says, "Make one of these." So in the preceding example, you can say:

[42]This can be a flashpoint. There are those who say, "Clearly, it's a pointer," but this presumes an underlying implementation. Also, the syntax of Java references is much more akin to C++ references than to pointers. In the 1st edition of *Thinking in Java*, I chose to invent a new term, "handle," because C++ references and Java references have some important differences. I was coming out of C++ and did not want to confuse the C++ programmers whom I assumed would be the largest audience for Java. In the 2nd edition of *Thinking in Java*, I decided that "reference" was the more commonly used term, and that anyone changing from C++ would have a lot more to cope with than the terminology of references, so they might as well jump in with both feet. However, there are people who disagree even with the term "reference." In one book I read that it was "completely wrong to say that Java supports pass by reference," because Java object identifiers (according to that author) are *actually* "object references." And (he goes on) everything is *actually* pass by value. So you're not passing by reference, you're "passing an object reference by value." One could argue for the precision of such convoluted explanations, but I think my approach simplifies the understanding of the concept without hurting anything (well, language lawyers may claim I'm lying to you, but I'll say that I'm providing an appropriate abstraction).

```
String s = new String("asdf");
```

Not only does this mean "Make a new `String`," but it also gives information about *how* to make the `String` by supplying an initial group of characters.

Java comes with a plethora of ready-made types in addition to `String`. On top of that, you can create your own types. In fact, creating new types is the fundamental activity in Java programming, and it's what you'll be learning about in the rest of this book.

Where Storage Lives

It's useful to visualize the way things are laid out while the program is running—in particular, how memory is arranged. There are five different places to store data:

1. **Registers**. This is the fastest storage because it exists in a place different from that of other storage: inside the *central processing unit* (CPU)[43]. However, the number of registers is severely limited, so registers are allocated as they are needed. You don't have direct control over register allocation, nor do you see any evidence in your programs that registers even exist (C & C++, on the other hand, allow you to suggest register allocation to the compiler).
2. **The stack**. This lives in the general random-access memory (RAM) area, but has direct support from the processor via its *stack pointer*. The stack pointer is moved down to create new memory and moved up to release that memory. This is an extremely fast and efficient way to allocate storage, second only to registers. The Java system must know, while it is creating the program, the exact lifetime of all the items stored on the stack. This constraint places limits on the flexibility of your programs, so while some Java storage exists on the stack—in particular, object references—Java objects themselves are not placed on the stack.
3. **The heap**. This is a general-purpose pool of memory (also in the RAM area) where all Java objects live. Unlike the stack, the compiler doesn't need to know how long objects must stay on the heap. Thus, there's a great deal of flexibility when using heap storage. Whenever you need an object, you write the code to create it using `new`, and the storage is allocated on the heap when that code is

[43]Most microprocessor chips do have additional *cache* memory but this is organized as traditional memory and not as registers.

executed. There's a price for this flexibility: It can take more time to allocate and clean up heap storage than stack storage (if you even *could* create objects on the stack in Java, as you can in C++). Over time, however, Java's heap allocation mechanism has become very fast, so this is not an issue for concern.

4. **Constant storage**. Constant values are often placed directly in the program code, which is safe because they can never change. Sometimes constants are cordoned off by themselves so they can be optionally placed in read-only memory (ROM), in embedded systems.[44]
5. **Non-RAM storage**. If data lives completely outside a program, it can exist while the program is not running, outside the control of the program. The two primary examples of this are *serialized objects*, where objects are turned into streams of bytes, generally sent to another machine, and *persistent objects*, where the objects are placed on disk so they will hold their state even when the program is terminated. The trick with these types of storage is turning the objects into something that exist on the other medium, and yet be resurrected into a regular RAM-based object when necessary. Java provides support for *lightweight persistence*. Libraries such as JDBC[45] and Hibernate[46] provide more sophisticated support for storing and retrieving object information using databases.

Special Case: Primitive Types

One group of types that you'll often use gets special treatment. You can think of these as "primitive" types. The reason for the special treatment is that creating an object with `new`—especially a small, simple variable—isn't very efficient, because `new` places objects on the heap. For these types Java falls back on the approach taken by C and C++. That is, instead of creating the variable using `new`, an "automatic" variable is created that is *not a reference*. The variable holds the value directly, and it's placed on the stack, so it's much more efficient.

Java specifies the size of each primitive type, and these sizes don't change from one machine architecture to another as they do in some languages. This size invariance is one reason Java programs are more portable than programs in some other languages.

[44] An example of this is the `String` pool. All literal `Strings` and `String`-valued constant expressions are interned automatically and put into special static storage.

[45] http://docs.oracle.com/javase/8/docs/technotes/guides/jdbc/

[46] http://hibernate.org/

Primitive	Size	Min	Max	Wrapper
`boolean`	—	—	—	`Boolean`
`char`	16 bits	Unicode 0 `\u0000`	65,535 `\uffff`	`Character`
`byte`	8 bits	-128	+127	`Byte`
`short`	16 bits	-32,768	+32,767	`Short`
`int`	32 bits	-2^{31}	$+2^{31}-1$	`Integer`
`long`	64 bits	-2^{63}	$+2^{63}-1$	`Long`
`float`	32 bits	IEEE754	IEEE754	`Float`
`double`	64 bits	IEEE754	IEEE754	`Double`
`void`	—	—	—	`Void`

All numeric types are signed, so don't look for unsigned types.

The size of the `boolean` type is not explicitly specified; it is defined to take the literal values `true` or `false`.

"Wrapper" classes for primitive data types create a non-primitive object on the heap to represent that primitive type. For example:

```
char c = 'x';
Character ch = new Character(c);
```

You can also use:

```
Character ch = new Character('x');
```

Autoboxing automatically converts a primitive to a wrapped object:

```
Character ch = 'x';
```

and back:

```
char c = ch;
```

The reasons for wrapping primitives are shown in a later chapter.

High-Precision Numbers

Java includes two classes for performing high-precision arithmetic: `BigInteger` and `BigDecimal`. Although these fit approximately the same category as the "wrapper" classes, neither one has a corresponding primitive.

Both classes have methods that provide analogues for the operations you perform on primitive types. That is, you can do anything with a `BigInteger` or `BigDecimal` you can with an `int` or `float`, it's just that you must use method calls instead of operators. Also, because there are more calculations involved, the operations are slower. You're exchanging speed for accuracy.

`BigInteger` supports arbitrary-precision integers. This means you can accurately represent integral values of any size without losing any information during operations.

`BigDecimal` is for arbitrary-precision fixed-point numbers; you can use these for accurate monetary calculations, for example.

Consult the JDK documentation for details about these two classes.

Arrays in Java

Many programming languages support some kind of array. Using arrays in C and C++ is perilous because those arrays are only blocks of memory. If a program accesses the array outside of its memory block or uses the memory before initialization (common programming errors), the results are unpredictable.

One of the primary goals of Java is safety, so many of the problems that plague programmers in C and C++ are not repeated in Java. A Java array is guaranteed to be initialized and cannot be accessed outside of its range. This range checking comes at the price of having a small amount of memory overhead for each array as well as extra time to verify the index at run time, but the assumption is that the safety and increased productivity are worth the expense (and Java can often optimize these operations).

When you create an array of objects, you are really creating an array of references, and each of those references is automatically initialized to a special value with its own keyword: `null`. When Java sees `null`, it recognizes that the reference in question isn't pointing to an object. You must assign an object to each reference before you use it, and if you try to use a reference that's still `null`, the problem is reported at run time. Thus, typical array errors are prevented in Java.

You can also create an array of primitives. The compiler guarantees initialization by zeroing the memory for that array.

Arrays are covered in detail later in the book, and specifically in the Arrays chapter.

Comments

There are two types of comments in Java. The first are traditional C-style comments which begin with a `/*` and continue, possibly across many lines, until a `*/`. Note that many programmers begin each line of a multiline comment with a `*`, so you'll often see:

```
/* This is a comment
 * that continues
 * across lines
 */
```

Remember, however, that everything inside the `/*` and `*/` is ignored, so there's no difference if you say instead:

```
/* This is a comment that
continues across lines */
```

The second form of comment comes from C++. It is the single-line comment, which starts with a `//` and continues until the end of the line. This type of comment is convenient and commonly used because it's easy. So you often see:

```
// This is a one-line comment
```

You Never Need to Destroy an Object

In some programming languages, managing storage lifetime requires significant effort. How long does a variable last? If you are supposed to destroy it, when should you? Confusion over storage lifetime can lead to many bugs, and this section shows how Java simplifies the issue by releasing storage for you.

Scoping

Most procedural languages have the concept of *scope*. This determines both the visibility and lifetime of the names defined within that scope. In C, C++, and Java, scope is determined by the placement of curly braces {}. Here is a fragment of Java code demonstrating scope:

```
{
  int x = 12;
  // Only x available
  {
    int q = 96;
    // Both x & q available
  }
  // Only x available
  // q is "out of scope"
}
```

A variable defined within a scope is available only until the end of that scope.

Indentation makes Java code easier to read. Because Java is a free-form language, the extra spaces, tabs, and carriage returns do not affect the resulting program.

You *cannot* do the following, even though it is legal in C and C++:

```
{
  int x = 12;
  {
    int x = 96; // Illegal
  }
}
```

The Java compiler will announce that the variable `x` has already been defined. Thus the C and C++ ability to "hide" a variable in a larger scope is not allowed, because the Java designers thought it led to confusing programs.

Scope of Objects

Java objects do not have the same lifetimes as primitives. When you create a Java object using `new`, it persists past the end of the scope. Thus, if you say:

```
{
  String s = new String("a string");
} // End of scope
```

the reference `s` vanishes at the end of the scope. However, the `String` object that `s` points to is still occupying memory. In this bit of code, there is no way to access the object after the end of the scope, because the only reference to it is out of scope. In later chapters you'll see how the reference to the object can be passed around and duplicated during the course of a program.

Because objects created with `new` exist as long as you need them, a whole slew of C++ programming problems vanish in Java. In C++ you must not only make sure that the objects stay around as long as necessary, you must also destroy the objects when you're done with them.

This brings up a question. If Java leaves the objects lying around, what keeps them from filling up memory and halting your program, which is exactly the kind of problem that occurs in C++? In Java, a bit of magic happens: the *garbage collector* looks at all the objects created with `new` and finds those that are no longer referenced. It then releases the memory for those objects, so the memory can be used for new objects. This means you don't worry about reclaiming memory yourself. You simply create objects, and when you no longer need them, they go away by themselves. This prevents an important class of programming problem: the so-called "memory leak," when a programmer forgets to release memory.

Creating New Data Types: `class`

If everything is an object, what determines how a particular class of object looks and behaves? Put another way, what establishes the *type* of an object? You might expect a keyword called "type," and that would certainly make sense. Historically, however, most object-oriented languages use the keyword `class` to describe a new kind of object. The `class` keyword (so common it will often not be bold-faced throughout this book) is followed by the name of the new type. For example:

```
class ATypeName {
  // Class body goes here
}
```

This introduces a new type, although here the class body consists only of a comment, so there is not too much you can do with it. However, you can create an object of `ATypeName` using `new`:

```
ATypeName a = new ATypeName();
```

You can't tell it to do much of anything—that is, you cannot send it any interesting messages—until you define some methods for it.

Fields

When you define a class, you can put two types of elements in your class: *fields* (sometimes called *data members*), and *methods* (sometimes called *member functions*). A field is an object of any type you can talk to via its reference. A field can also be a primitive type. If it is a reference to an object, you must initialize that reference to connect it to an actual object (using `new`, as seen earlier).

Each object keeps its own storage for its fields. Ordinarily, fields are not shared among objects. Here is an example of a class with some fields:

```
class DataOnly {
  int i;
  double d;
  boolean b;
}
```

This class doesn't do anything except hold data. As before, you create an object like this:

```
DataOnly data = new DataOnly();
```

You can assign values to the fields by referring to object members. To do this, you state the name of the object reference, followed by a period (dot), followed by the name of the member inside the object:

```
objectReference.member
```

For example:

```
data.i = 47;
data.d = 1.1;
data.b = false;
```

What if your object contains other objects that contain data you want to modify? You just keep "connecting the dots." For example:

```
myPlane.leftTank.capacity = 100;
```

You can nest many objects this way (although such a design might become confusing).

Default Values for Primitive Members

When a primitive data type is a field in a class, it is guaranteed to get a default value if you do not initialize it:

Primitive	Default
`boolean`	`false`
`char`	`\u0000` (`null`)
`byte`	`(byte)0`
`short`	`(short)0`
`int`	`0`
`long`	`0L`
`float`	`0.0f`
`double`	`0.0d`

The default values are only what Java guarantees when the variable is used *as a member of a class*. This ensures that primitive fields will always be initialized (something C++ doesn't do), reducing a source of bugs. However, this initial value might not be correct or even legal for the program you are writing. It's best to always explicitly initialize your variables.

This guarantee doesn't apply to *local variables*—those that are not fields of a class. Thus, if within a method definition you have:

```
int x;
```

Then `x` will get some arbitrary value (as it does in C and C++); it will not automatically be initialized to zero. You are responsible for assigning an appropriate value before you use `x`. If you forget, Java definitely improves on C++: You get a compile-time error telling you the variable might not be initialized. (C++ compilers often warn you about uninitialized variables, but in Java these are errors.)

Methods, Arguments, and Return Values

In many languages (like C and C++), the term *function* is used to describe a named subroutine. In Java, we use the term *method*, as in "a way to do something."

Methods in Java determine the messages an object can receive. The fundamental parts of a method are the name, the arguments, the return type, and the body. Here is the basic form:

```
ReturnType methodName( /* Argument list */ ) {
  // Method body
}
```

`ReturnType` indicates the type of value produced by the method when you call it. The argument list gives the types and names for the information you pass into the method. The method name and argument list are collectively called the *signature* of the method. The signature uniquely identifies that method.

Methods in Java can only be created as part of a class. A method can be called only for an object[47], and that object must be able to perform that method call. If you try to call the wrong method for an object, you'll get an error message at compile time.

You call a method for an object by giving the object reference followed by a period (dot), followed by the name of the method and its argument list, like this:

```
objectReference.methodName(arg1, arg2, arg3);
```

Consider a method `f()` that takes no arguments and returns a value of type `int`. For a reference `a` that accepts calls to `f()`, you can say this:

```
int x = a.f();
```

The type of the return value must be compatible with the type of `x`.

This act of calling a method is sometimes termed *sending a message to an object*. In the preceding example, the message is `f()` and the object is `a`. Object-oriented programming can be summarized as "sending messages to objects."

The Argument List

The method argument list specifies the information you pass into the method. As you might guess, this information—like everything else in Java—takes the form of objects. The argument list must specify the object types and the name of each

[47] `static` methods, which you'll learn about soon, can be called *for the class*, without an object.

object. As always, where you seem to be handing objects around, you are actually passing references.[48] The type of the reference must be correct, however. If a `String` argument is expected, you must pass in a `String` or the compiler will give an error.

Here is the definition for a method that takes a `String` as its argument. It must be placed within a class for it to compile:

```
int storage(String s) {
  return s.length() * 2;
}
```

This method calculates and delivers the number of bytes required to hold the information in a particular `String`. The argument `s` is of type `String`. Once `s` is passed into `storage()`, you can treat it like any other object—you can send it messages. Here, we call `length()`, which is a `String` method that returns the number of characters in a `String`. The size of each `char` in a `String` is 16 bits, or two bytes.

You can also see the `return` keyword, which does two things. First, it means "Leave the method, I'm done." Second, if the method produces a value, that value is placed right after the `return` statement. Here, the return value is produced by evaluating the expression `s.length() * 2`.

You can return any type you want, but if you don't return anything at all, you do so by indicating that the method produces `void` (nothing). Here are some examples:

```
boolean flag() { return true; }
double naturalLogBase() { return 2.718; }
void nothing() { return; }
void nothing2() {}
```

When the return type is `void`, the `return` keyword is used only to exit the method, and is therefore unnecessary if called at the end of the method. You can return from a method at any point, but if you've given a non-`void` return type, the compiler will force you to return the appropriate type of value regardless of where you return.

It might look like a program is just a bunch of objects with methods that take other objects as arguments and send messages to those other objects. That is indeed much

[48] With the usual exception of the aforementioned "special" data types `boolean`, `char`, `byte`, `short`, `int`, `long`, `float`, and `double`. In general, though, you pass objects, which really means you pass references to objects.

of what goes on, but in the following Operators chapter you'll learn how to do the detailed low-level work by making decisions within a method. For this chapter, sending messages will suffice.

Writing a Java Program

There are several other issues you must understand before seeing your first Java program.

Name Visibility

A problem in any programming language is the control of names. If you use a name in one module of the program, and another programmer uses the same name in another module, how do you distinguish one name from another and prevent the two names from "clashing?" In C this is especially challenging because a program is often an unmanageable sea of names. C++ classes (on which Java classes are modeled) nest functions within classes so they cannot clash with function names nested within other classes. However, C++ continues to allow global data and global functions, so clashing is still possible. To solve this problem, C++ introduced *namespaces* using additional keywords.

Java avoided all these problems by taking a fresh approach. To produce an unambiguous name for a library, the Java creators want you to use your Internet domain name in reverse, because domain names are guaranteed to be unique. My domain name is `MindviewInc.com` so my `foibles` utility library is named `com.mindviewinc.utility.foibles`. Following your reversed domain name, the dots are intended to represent subdirectories.

In Java 1.0 and Java 1.1 the domain extensions `com`, `edu`, `org`, `net`, etc., were capitalized by convention, so the library would appear: `Com.mindviewinc.utility.foibles`. Partway through the development of Java 2, however, they discovered this caused problems, so now the entire package name is lowercase.

This mechanism means all your files automatically live in their own namespaces, and each class within a file has a unique identifier. This way, the language prevents name clashes.

Using reversed URLs was a new approach to namespaces, never before tried in another language. Java has a number of these "inventive" approaches to problems. As you might imagine, adding a feature without experimenting with it first risks discovering problems with that feature in the future, after the feature is used in production code, typically when it's too late to do anything about it (some mistakes were bad enough to actually remove things from the language).

The problem with associating namespaces with file paths using reversed URLs is by no means one that causes bugs, but it does make it challenging to manage source code. By using `com.mindviewinc.utility.foibles`, I create a directory hierarchy with "empty" directories `com` and `mindviewinc` whose only job is to reflect the reversed URL. This approach seemed to open the door to what you will encounter in production Java programs: deep directory hierarchies filled with empty directories, not just for the reversed URLs but also to capture other information. These long paths are basically being used to store data about what is in the directory. If you expect to use directories in the way they were originally designed, this approach lands anywhere from "frustrating" to "maddening," and for production Java code you are essentially forced to use one of the IDEs specifically designed to manage code that is laid out in this fashion, such as NetBeans, Eclipse, or IntelliJ IDEA. Indeed, those IDEs both manage and create the deep empty directory hierarchies for you.

For this book's examples, I didn't want to burden you with the extra annoyance of the deep hierarchies, which would have effectively required you to learn one of the big IDEs before getting started. The examples for each chapter are in a shallow subdirectory with a name reflecting the chapter title. This caused me occasional struggles with tools that follow the deep-hierarchy approach.

Using Other Components

Whenever you use a predefined class in your program, the compiler must locate that class. In the simplest case, the class already exists in the source-code file it's being called from. In that case, you simply use the class—even if the class doesn't get defined until later in the file (Java eliminates the so-called "forward referencing" problem).

What about a class that exists in some other file? You might think the compiler should be smart enough to go and find it, but there is a problem. Imagine you use a class with a particular name, but more than one definition for that class exists (presumably

these are different definitions). Or worse, imagine that you're writing a program, and as you're building it you add a new class to your library that conflicts with the name of an existing class.

To solve this problem, you must eliminate all potential ambiguities by telling the Java compiler exactly what classes you want using the `import` keyword. `import` tells the compiler to bring in a package, which is a library of classes. (In other languages, a library could consist of functions and data as well as classes, but remember that all activities in Java take place within classes.)

Much of the time you'll use components from the standard Java libraries that come with your compiler. With these, you don't worry about long, reversed domain names; you just say, for example:

```
import java.util.ArrayList;
```

This tells the compiler to use Java's `ArrayList` class, located in its `util` library.

However, `util` contains several classes, and you might want to use several of them without declaring them all explicitly. This is easily accomplished by using `*` to indicate a wild card:

```
import java.util.*;
```

The examples in this book are small and for simplicity's sake we'll usually use the `*` form. However, many style guides specify that each class should be individually imported.

The `static` Keyword

Creating a class describes the look of its objects and the way they behave. You don't actually get an object until you create one using `new`, at which point storage is allocated and methods become available.

This approach is insufficient in two cases. Sometimes you want only a single, shared piece of storage for a particular field, regardless of how many objects of that class are created, or even if no objects are created. The second case is if you need a method that isn't associated with any particular object of this class. That is, you need a method you can call even if no objects are created.

The `static` keyword (adopted from C++) produces both these effects. When you say something is `static`, it means the field or method is not tied to any particular object instance. Even if you've never created an object of that class, you can call a `static` method or access a `static` field. With ordinary, non-`static` fields and methods, you must create an object and use that object to access the field or method, because non-`static` fields and methods must target a particular object.[49]

Some object-oriented languages use the terms *class data* and *class methods*, meaning that the data and methods exist only for the class as a whole, and not for any particular objects of the class. Sometimes Java literature uses these terms too.

To make a field or method `static`, you place the keyword before the definition. The following produces and initializes a `static` field:

```
class StaticTest {
  static int i = 47;
}
```

Now even if you make two `StaticTest` objects, there is still only one piece of storage for `StaticTest.i`. Both objects share the same `i`. For example:

```
StaticTest st1 = new StaticTest();
StaticTest st2 = new StaticTest();
```

Both `st1.i` and `st2.i` have the same value of 47 because they are the same piece of memory.

There are two ways to refer to a `static` variable. As in the preceding example, you can name it via an object; for example, `st2.i`. You can also refer to it directly through its class name, something you cannot do with a non-`static` member:

```
StaticTest.i++;
```

The ++ operator adds one to the variable. Now both `st1.i` and `st2.i` have the value 48.

Using the class name is the preferred way to refer to a `static` variable because it emphasizes the variable's `static` nature[50].

[49] `static` methods don't require objects to be created before they are used, so they cannot *directly* access non-`static` members or methods by calling those other members without referring to a named object (because non-`static` members and methods must be tied to a particular object).

[50] In some cases it also gives the compiler better opportunities for optimization.

Similar logic applies to `static` methods. You can refer to a `static` method either through an object as you can with any method, or with the special additional syntax `ClassName.method()`. You define a `static` method like this:

```
class Incrementable {
  static void increment() { StaticTest.i++; }
}
```

The `Incrementable` method `increment()` increments the `static int i` using the `++` operator. You can call `increment()` in the typical way, through an object:

```
Incrementable sf = new Incrementable();
sf.increment();
```

However, the preferred approach is to call it directly through its class:

```
Incrementable.increment();
```

`static` applied to a field definitely changes the way the data is created—one for each class versus the non-`static` one for each object. When applied to a method, `static` allows you to call that method without creating an object. This is essential, as you will see, in defining the `main()` method that is the entry point for running an application.

Your First Java Program

Finally, here's the first complete program. It starts by displaying a `String`, followed by the date, using the `Date` class from the Java standard library.

```
// objects/HelloDate.java
import java.util.*;

public class HelloDate {
  public static void main(String[] args) {
    System.out.println("Hello, it's: ");
    System.out.println(new Date());
  }
}
```

In this book I treat the first line specially; it's always a comment line containing the the path information to the file (using the directory name `objects` for this chapter) followed by the file name. I have tools to automatically extract and test the book's code based on this information, and you will easily find the code example in the repository[51] by referring to the first line.

At the beginning of each program file, you must place `import` statements to bring in any extra classes you need for the code in that file. I say "extra" because there's a certain library of classes automatically included in every Java file: `java.lang`. Start up your Web browser and look at the documentation from Oracle. If you haven't downloaded the JDK documentation from the Oracle Java site, do so now[52], or find it on the Internet. If you look at the list of packages, you'll see all the different class libraries that come with Java. Select `java.lang`. This will bring up a list of all the classes that are part of that library. Because `java.lang` is implicitly included in every Java code file, these classes are automatically available. There's no `Date` class listed in `java.lang`, which means you must import another library to use that. If you don't know the library where a particular class is, or if you want to see all classes, select "Tree" in the Java documentation. Now you can find every single class that comes with Java. Use the browser's "find" function to find `Date`. You'll see it listed as `java.util.Date`, which tells you it's in the `util` library and that you must `import java.util.*` to use `Date`.

If inside the documentation you select `java.lang`, then `System`, you'll see that the `System` class has several fields, and if you select `out`, you'll discover it's a `static PrintStream` object. It's `static` so you don't need to use `new`—the `out` object is always there, and you can just use it. What you can do with this `out` object is

[51] https://github.com/BruceEckel/OnJava8-Examples

[52] java.oracle.com. Note the documentation doesn't come packed with the JDK; you must do a separate download to get it.

determined by its type: `PrintStream`. Conveniently, `PrintStream` is shown in the description as a hyperlink, so if you click on that, you'll see a list of all the methods you can call for `PrintStream`. There are quite a few, and these are covered later in the book. For now all we're interested in is `println()`, which in effect means "Print what I'm giving you out to the console and end with a newline." Thus, in any Java program you can write something like this:

```
System.out.println("A String of things");
```

whenever you want to display information to the console.

One of the classes in the file must have the same name as the file. (The compiler complains if you don't do this.) When you're creating a standalone program such as this one, the class with the name of the file must contain an *entry point* from which the program starts. This special method is called `main()`, with the following signature and return type:

```
public static void main(String[] args) {
```

The `public` keyword means the method is available to the outside world (described in detail in the Implementation Hiding chapter). The argument to `main()` is an array of `String` objects. The `args` won't be used in the current program, but the Java compiler insists they be there because they hold the arguments from the command line.

The line that prints the date is interesting:

```
System.out.println(new Date());
```

The argument is a `Date` object that is only created to send its value (automatically converted to a `String`) to `println()`. As soon as this statement is finished, that `Date` is unnecessary, and the garbage collector can come along and get it anytime. We don't worry about cleaning it up.

When you look at the JDK documentation, you see that `System` has many other useful methods (one of Java's assets is its large set of standard libraries). For example:

```
// objects/ShowProperties.java

public class ShowProperties {
  public static void main(String[] args) {
    System.getProperties().list(System.out);
    System.out.println(System.getProperty("user.name"));
    System.out.println(
      System.getProperty("java.library.path"));
  }
}
/* Output: (First 20 Lines)
-- listing properties --
java.runtime.name=OpenJDK Runtime Environment
sun.boot.library.path=C:\Program Files\OpenJDK\java-
se-8u41...
java.vm.version=25.40-b25
java.vm.vendor=Oracle Corporation
java.vendor.url=http://java.oracle.com/
path.separator=;
java.vm.name=OpenJDK Client VM
file.encoding.pkg=sun.io
user.script=
user.country=US
sun.java.launcher=SUN_STANDARD
sun.os.patch.level=
java.vm.specification.name=Java Virtual Machine
Specification
user.dir=C:\Git\OnJava8\ExtractedExamples\objects
java.runtime.version=1.8.0_41-b04
java.awt.graphicsenv=sun.awt.Win32GraphicsEnvironment
java.endorsed.dirs=C:\Program Files\OpenJDK\java-
se-8u41...
os.arch=x86
java.io.tmpdir=C:\Users\Bruce\AppData\Local\Temp\
                  ...
*/
```

The first line in `main()` displays all "properties" from the system where you are running the program, so it gives you environment information. The `list()` method sends the results to its argument, `System.out`. You will see later in the book that you can send the results elsewhere, to a file, for example. You can also ask for a specific property—here, `user.name` and `java.library.path`.

The `/* Output:` tag at the end of the file indicates the beginning of the output generated by this file. Most examples in this book that produce output will contain the output in this commented form, so you see the output and know it is correct. The tag allows the output to be automatically updated into the text of this book after being checked with a compiler and executed.

Compiling and Running

To compile and run this program, and all the other programs in this book, you must first have a Java programming environment. The installation process was described in Installing Java and the Book Examples. If you followed these instructions, you are using the Java Developer's Kit (JDK), free from Oracle. If you use another development system, look in the documentation for that system to determine how to compile and run programs.

Installing Java and the Book Examples also describes how to install the examples for this book. Move to the subdirectory named `objects` and type:

```
javac HelloDate.java
```

This command should produce no response. If you get any kind of an error message, it means you haven't installed the JDK properly and you must investigate those problems.

On the other hand, if you just get your command prompt back, you can type:

```
java HelloDate
```

and you'll get the message and the date as output.

This is the process to compile and run each program (containing a `main()`) in this book[53]. However, the source code for this book also has a file called `build.gradle` in the root directory, and this contains the *Gradle* configuration for automatically building, testing, and running the files for the book. When you run the `gradlew` command for the first time, Gradle will automatically install itself (assuming you have Java installed).

[53]For every program in this book to compile and run through the command line, you might also need to set your CLASSPATH.

Coding Style

The style described in the document *Code Conventions for the Java Programming Language*[54] is to capitalize the first letter of a class name. If the class name consists of several words, they are run together (that is, you don't use underscores to separate the names), and the first letter of each embedded word is capitalized, such as:

```
class AllTheColorsOfTheRainbow { // ...
```

This is sometimes called "camel-casing." For almost everything else—methods, fields (member variables), and object reference names—the accepted style is just as it is for classes *except* that the first letter of the identifier is lowercase. For example:

```
class AllTheColorsOfTheRainbow {
  int anIntegerRepresentingColors;
  void changeTheHueOfTheColor(int newHue) {
    // ...
  }
  // ...
}
```

The user must also type these long names, so be merciful.

The Java code you find in the Oracle libraries also follows the placement of open-and-close curly braces in this book.

Summary

This chapter shows you just enough Java so you understand how to write a simple program. You've also seen an overview of the language and some of its basic ideas. However, the examples so far have all been of the form "Do this, then do that, then do something else." The next two chapters will introduce the basic operators used in Java programming, and show you how to control the flow of your program.

[54](Search the Internet; also look for "Google Java Style"). To keep code listings narrow for this book, not all these guidelines could be followed, but you'll see that the style I use here matches the Java standard as much as possible.

Operators

> Operators manipulate data.

Because Java was inherited from C++, most of its operators are familiar to C and C++ programmers. Java also adds some improvements and simplifications.

If you know C or C++ syntax, you can skim through this chapter and the next, looking for places where Java is different from those languages. However, if you find yourself floundering a bit in these two chapters, make sure you go through the free Thinking in C[55] seminar. It contains audio lectures, slides, exercises, and solutions specifically designed to bring you up to speed with the fundamentals necessary to learn Java.

Using Java Operators

An operator takes one or more arguments and produces a new value. The arguments are in a different form than ordinary method calls, but the effect is the same. Addition and unary plus (+), subtraction and unary minus (-), multiplication (*), division (/), and assignment (=) all work much the same in any programming language.

All operators produce a value from their operands. In addition, some operators change the value of an operand. This is called a *side effect*. The most common use for operators that modify their operands is to generate the side effect, but keep in mind that the value produced is available for your use, just as in operators without side effects.

Almost all operators work only with primitives. The exceptions are `=`, `==` and `!=`, which work with all objects (and are a point of confusion for objects). In addition, the `String` class supports `+` and `+=`.

[55] https://archive.org/details/ThinkingInC

Precedence

Operator precedence defines expression evaluation when several operators are present. Java has specific rules that determine the order of evaluation. The easiest one to remember is that multiplication and division happen before addition and subtraction. Programmers often forget the other precedence rules, and use parentheses to make the order of evaluation explicit. For example, look at statements **[1]** and **[2]**:

```
// operators/Precedence.java

public class Precedence {
  public static void main(String[] args) {
    int x = 1, y = 2, z = 3;
    int a = x + y - 2/2 + z;          // [1]
    int b = x + (y - 2)/(2 + z);      // [2]
    System.out.println("a = " + a);
    System.out.println("b = " + b);
  }
}
/* Output:
a = 5
b = 1
*/
```

These statements look roughly the same, but from the output you see they have very different meanings depending on the use of parentheses.

Notice that `System.out.println()` uses the `+` operator. In this context, `+` means "`String` concatenation" and, if necessary, "`String` conversion." When the compiler sees a `String` followed by a `+` followed by a non-`String`, it attempts to convert the non-`String` into a `String`. The output shows it successfully converts from `int` into `String` for `a` and `b`.

Assignment

The operator = performs assignment. It means "Take the value of the right-hand side (often called the *rvalue*) and copy it into the left-hand side (often called the *lvalue*)."

An rvalue is any constant, variable, or expression that produces a value, but an lvalue must be a distinct, named variable. (That is, there must be a physical space to store the value.) For instance, you can assign a constant value to a variable:

```
a = 4;
```

but you cannot assign anything to a constant value—it cannot be an lvalue. (You can't say `4 = a;`.)

Assigning primitives is straightforward. The primitive holds the actual value and not a reference to an object, so when you assign primitives, you copy the contents from one place to another. For example, if you say `a = b` for primitives, the contents of `b` are copied into `a`. If you then go on to modify `a`, `b` is naturally unaffected by this modification. As a programmer, this is what you can expect for most situations.

When you assign objects, however, things change. Whenever you manipulate an object, what you're manipulating is the reference, so when you assign "from one object to another," you're actually copying a reference from one place to another. This means if you say `c = d` for objects, you end up with both `c` and `d` pointing to the object where, originally, only `d` pointed. This example demonstrates the behavior:

```
// operators/Assignment.java
// Assignment with objects is a bit tricky

class Tank {
  int level;
}

public class Assignment {
  public static void main(String[] args) {
    Tank t1 = new Tank();
    Tank t2 = new Tank();
    t1.level = 9;
    t2.level = 47;
    System.out.println("1: t1.level: " + t1.level +
      ", t2.level: " + t2.level);
    t1 = t2;
    System.out.println("2: t1.level: " + t1.level +
      ", t2.level: " + t2.level);
    t1.level = 27;
    System.out.println("3: t1.level: " + t1.level +
```

```
      ", t2.level: " + t2.level);
  }
}
/* Output:
1: t1.level: 9, t2.level: 47
2: t1.level: 47, t2.level: 47
3: t1.level: 27, t2.level: 27
*/
```

The `Tank` class is simple, and two instances (`t1` and `t2`) are created within `main()`. The `level` field within each `Tank` is given a different value, then `t2` is assigned to `t1`, and `t1` is changed. In many programming languages you expect `t1` and `t2` to be independent at all times, but because you've assigned a reference, changing the `t1` object appears to change the `t2` object as well! This is because both `t1` and `t2` contain references that point to the same object. (The original reference that was in `t1`, that pointed to the object holding a value of `9`, was overwritten during the assignment and effectively lost; its object is cleaned up by the garbage collector.)

This phenomenon is often called *aliasing*, and it's a fundamental way that Java works with objects. But what if you don't want aliasing to occur here? You can forego the assignment and say:

```
t1.level = t2.level;
```

This retains the two separate objects instead of discarding one and tying `t1` and `t2` to the same object. Manipulating the fields within objects goes against Java design principles. This is a nontrivial topic, so keep in mind that assignment for objects can add surprises.

Aliasing During Method Calls

Aliasing will also occur when you pass an object into a method:

```
// operators/PassObject.java
// Passing objects to methods might not be
// what you're used to

class Letter {
  char c;
}

public class PassObject {
  static void f(Letter y) {
    y.c = 'z';
  }
  public static void main(String[] args) {
    Letter x = new Letter();
    x.c = 'a';
    System.out.println("1: x.c: " + x.c);
    f(x);
    System.out.println("2: x.c: " + x.c);
  }
}
/* Output:
1: x.c: a
2: x.c: z
*/
```

In many programming languages, the method `f()` appears to make a copy of its argument `Letter y` inside the scope of the method. But once again a reference is passed, so the line

```
y.c = 'z';
```

is actually changing the object *outside* of `f()`.

Aliasing and its solution is a complex issue covered in the Appendix: Passing and Returning Objects. You're aware of it now so you can watch for pitfalls.

Mathematical Operators

The basic mathematical operators are the same as the ones available in most programming languages: addition (+), subtraction (-), division (/), multiplication

(*) and modulus (%, which produces the remainder from division). Integer division truncates, rather than rounds, the result.

Java also uses the shorthand notation from C/C++ that performs an operation and an assignment at the same time. This is denoted by an operator followed by an equal sign, and is consistent with all the operators in the language (whenever it makes sense). For example, to add 4 to the variable x and assign the result to x, use: `x += 4`.

This example shows the mathematical operators:

```
// operators/MathOps.java
// The mathematical operators
import java.util.*;

public class MathOps {
  public static void main(String[] args) {
    // Create a seeded random number generator:
    Random rand = new Random(47);
    int i, j, k;
    // Choose value from 1 to 100:
    j = rand.nextInt(100) + 1;
    System.out.println("j : " + j);
    k = rand.nextInt(100) + 1;
    System.out.println("k : " + k);
    i = j + k;
    System.out.println("j + k : " + i);
    i = j - k;
    System.out.println("j - k : " + i);
    i = k / j;
    System.out.println("k / j : " + i);
    i = k * j;
    System.out.println("k * j : " + i);
    i = k % j;
    System.out.println("k % j : " + i);
    j %= k;
    System.out.println("j %= k : " + j);
    // Floating-point number tests:
    float u, v, w; // Applies to doubles, too
    v = rand.nextFloat();
    System.out.println("v : " + v);
    w = rand.nextFloat();
```

```
    System.out.println("w : " + w);
    u = v + w;
    System.out.println("v + w : " + u);
    u = v - w;
    System.out.println("v - w : " + u);
    u = v * w;
    System.out.println("v * w : " + u);
    u = v / w;
    System.out.println("v / w : " + u);
    // The following also works for char,
    // byte, short, int, long, and double:
    u += v;
    System.out.println("u += v : " + u);
    u -= v;
    System.out.println("u -= v : " + u);
    u *= v;
    System.out.println("u *= v : " + u);
    u /= v;
    System.out.println("u /= v : " + u);
  }
}
/* Output:
j : 59
k : 56
j + k : 115
j - k : 3
k / j : 0
k * j : 3304
k % j : 56
j %= k : 3
v : 0.5309454
w : 0.0534122
v + w : 0.5843576
v - w : 0.47753322
v * w : 0.028358962
v / w : 9.940527
u += v : 10.471473
u -= v : 9.940527
u *= v : 5.2778773
u /= v : 9.940527
*/
```

To generate numbers, the program first creates a `Random` object. If you create a `Random` object with no arguments, Java uses the current time as a seed for the random number generator, and will thus produce different output for each execution of the program. However, in the examples in this book, it is important that the output at the end of each example be as consistent as possible so it can be verified with external tools. By providing a *seed* (an initialization value for the random number generator that always produces the same sequence for a particular seed value) when creating the `Random` object, the same random numbers are generated each time the program is executed, so the output is verifiable.[56] To generate more varying output, feel free to remove the seed in the examples in the book.

The program generates several different types of random numbers with the `Random` object by calling the methods `nextInt()` and `nextFloat()` (you can also call `nextLong()` or `nextDouble()`). The argument to `nextInt()` sets the upper bound on the generated number. The lower bound is zero, which we don't want because of the possibility of a divide-by-zero, so the result is offset by one.

Unary Minus and Plus Operators

The unary minus (-) and unary plus (+) are the same operators as binary minus and plus. The compiler figures out which use is intended by the way you write the expression. For instance, the statement

```
x = -a;
```

has an obvious meaning. The compiler is able to figure out:

```
x = a * -b;
```

but the reader might get confused, so it is sometimes clearer to say:

```
x = a * (-b);
```

Unary minus inverts the sign on the data. Unary plus provides symmetry with unary minus, but its only effect is to promote smaller-type operands to `int`.

[56] As an undergraduate, I attended Pomona College for two years, where the number 47 was considered a "magic number." See the Wikipedia article.

Auto Increment and Decrement

Java, like C, has a number of shortcuts. Shortcuts can make code much easier to type, and either easier or harder to read.

Two of the nicer shortcuts are the increment and decrement operators (often called the auto-increment and auto-decrement operators). The decrement operator is `--` and means "decrease by one unit." The increment operator is `++` and means "increase by one unit." If `a` is an `int`, for example, the expression `++a` is equivalent to `a = a + 1`. Increment and decrement operators not only modify the variable, but also produce the value of the variable as a result.

There are two versions of each type of operator, often called *prefix* and *postfix*. *Pre-increment* means the `++` operator appears before the variable, and *post-increment* means the `++` operator appears after the variable. Similarly, *pre-decrement* means the `--`operator appears before the variable, and *post-decrement* means the `--` operator appears after the variable. For pre-increment and pre-decrement (i.e., `++a` or `--a`), the operation is performed and the value is produced. For post-increment and post-decrement (i.e., `a++` or `a--`), the value is produced, then the operation is performed.

```
// operators/AutoInc.java
// The ++ and -- operators

public class AutoInc {
  public static void main(String[] args) {
    int i = 1;
    System.out.println("i: " + i);
    System.out.println("++i: " + ++i); // Pre-increment
    System.out.println("i++: " + i++); // Post-increment
    System.out.println("i: " + i);
    System.out.println("--i: " + --i); // Pre-decrement
    System.out.println("i--: " + i--); // Post-decrement
    System.out.println("i: " + i);
  }
}
/* Output:
i: 1
++i: 2
i++: 2
i: 3
```

```
--i: 2
i--: 2
i: 1
*/
```

For the prefix form, you get the value after the operation is performed, but with the postfix form, you get the value before the operation is performed. These are the only operators, other than those involving assignment, that have side effects—they change the operand rather than just using its value.

The increment operator is one explanation for the name C++, implying "one step beyond C." In an early Java speech, Bill Joy (one of the Java creators), said that "`Java = C++--`" (C plus plus minus minus), suggesting that Java is C++ with the unnecessary hard parts removed, and therefore a much simpler language. As you progress, you'll see that many parts are simpler, and yet in other ways Java isn't much easier than C++.

Relational Operators

Relational operators produce a `boolean` result indicating the relationship between the values of the operands. A relational expression produces `true` if the relationship is true, and `false` if the relationship is untrue. The relational operators are less than (`<`), greater than (`>`), less than or equal to (`<=`), greater than or equal to (`>=`), equivalent (`==`) and not equivalent (`!=`). Equivalence and non-equivalence work with all primitives, but the other comparisons won't work with type `boolean`. Because `boolean` values can only be `true` or `false`, "greater than" or "less than" doesn't make sense.

Testing Object Equivalence

The relational operators == and != work with all objects, but their results can be confusing:

```
// operators/Equivalence.java

public class Equivalence {
  static void show(String desc, Integer n1, Integer n2) {
    System.out.println(desc + ":");
    System.out.printf(
      "%d==%d %b %b%n", n1, n2, n1 == n2, n1.equals(n2));
  }
  @SuppressWarnings("deprecation")
  public static void test(int value) {
    Integer i1 = value;                              // [1]
    Integer i2 = value;
    show("Automatic", i1, i2);
    // Old way, deprecated in Java 9 and on:
    Integer r1 = new Integer(value);                 // [2]
    Integer r2 = new Integer(value);
    show("new Integer()", r1, r2);
    // Preferred in Java 9 and on:
    Integer v1 = Integer.valueOf(value);             // [3]
    Integer v2 = Integer.valueOf(value);
    show("Integer.valueOf()", v1, v2);
    // Primitives can't use equals():
    int x = value;                                   // [4]
    int y = value;
    // x.equals(y); // Doesn't compile
    System.out.println("Primitive int:");
    System.out.printf("%d==%d %b%n", x, y, x == y);
  }
  public static void main(String[] args) {
    test(127);
    test(128);
  }
}
/* Output:
Automatic:
127==127 true true
new Integer():
127==127 false true
Integer.valueOf():
127==127 true true
Primitive int:
127==127 true
```

```
Automatic:
128==128 false true
new Integer():
128==128 false true
Integer.valueOf():
128==128 false true
Primitive int:
128==128 true
*/
```

`show()` compares the behavior of == against the method `equals()` that exists for all objects. The `printf()` format argument uses the specifier `%d` for `int` output, `%b` for Boolean output, and `%n` to produce a newline.

For "not equals" comparisons, use `n1 != n2` and `!n1.equals(n2)`.

In `test()`, integer-valued objects are created in four different ways:

- **[1]**: Automatic conversion to `Integer`. These are translated into calls to `Integer.valueOf()`.
- **[2]**: Using standard `new` object-creation syntax. Originally this was the preferred way to create "wrapped/boxed" `Integer` objects.
- **[3]**: Starting with Java 9, `valueOf()` is preferred over **[2]**. If you try to use form **[2]** with Java 9, you'll get a warning and a suggestion to use **[3]** instead. It is difficult to determine whether **[3]** is preferred over **[1]**, and **[1]** seems much cleaner.
- **[4]**: As primitive `int`s.

The `@SuppressWarnings("deprecation")` is not necessary for Java 8, but is included in case you compile the code with Java 9 or newer.

For a `value` of 127, the comparisons produce the results you expect, *except* form **[2]** which produces `false` for ==. Although the *contents* of the objects are the same, the references point to different objects in memory. The operators == and `!=` compare object references, and have different behavior *depending on how the `Integer` objects are created*—presumably, **[1]** and **[3]** yield `Integer`s that point to the same storage in memory. `Integer` values from -128 through 127 produce this behavior for == and `!=`, but outside that range they do not, as seen with `test(128)`.

If you're using `Integer` you must only use `equals()`. If you accidentally use == and `!=` for `Integer` and don't test it for values outside -128 through 127, your code will

pass but will be quietly broken. If you're using primitive `int`s then you *can't* use `equals()` and *must* use `==` and `!=`. This can cause problems if you start by using `int`s and then later change to `Integer`s, or vice-versa.

In Java 9 and on, the use of `new Integer()` is deprecated because it is so much less efficient than `Integer.valueOf()`. Thus, you should avoid `new Integer()`, `new Double()`, etc. in Java 8 as well. Deprecating something for efficiency reasons is not something I've seen before (it might have happened, but this is the first time I've been aware of it).

When working with floating-point numbers, you encounter different equivalence problems, not because of Java but because of the nature of floating-point numbers:

```
// operators/DoubleEquivalence.java

public class DoubleEquivalence {
  static void show(String desc, Double n1, Double n2) {
    System.out.println(desc + ":");
    System.out.printf(
      "%e==%e %b %b%n", n1, n2, n1 == n2, n1.equals(n2));
  }
  @SuppressWarnings("deprecation")
  public static void test(double x1, double x2) {
    // x1.equals(x2) // Won't compile
    System.out.printf("%e==%e %b%n", x1, x2, x1 == x2);
    Double d1 = x1;
    Double d2 = x2;
    show("Automatic", d1, d2);
    Double r1 = new Double(x1);
    Double r2 = new Double(x2);
    show("new Double()", r1, r2);
    Double v1 = Double.valueOf(x1);
    Double v2 = Double.valueOf(x2);
    show("Double.valueOf()", v1, v2);
  }
  public static void main(String[] args) {
    test(0, Double.MIN_VALUE);
    System.out.println("-------------------------");
    test(Double.MAX_VALUE,
      Double.MAX_VALUE - Double.MIN_VALUE * 1_000_000);
  }
}
```

```
/* Output:
0.000000e+00==4.900000e-324 false
Automatic:
0.000000e+00==4.900000e-324 false false
new Double():
0.000000e+00==4.900000e-324 false false
Double.valueOf():
0.000000e+00==4.900000e-324 false false
-----------------------
1.797693e+308==1.797693e+308 true
Automatic:
1.797693e+308==1.797693e+308 false true
new Double():
1.797693e+308==1.797693e+308 false true
Double.valueOf():
1.797693e+308==1.797693e+308 false true
*/
```

On paper, the comparison of floating point numbers should be very strict—a number that is the tiniest fraction different from another number should still be unequal. This works for the call to `test(0, Double.MIN_VALUE)`, where `Double.MIN_VALUE` is the smallest representable value. (`%e` in the `printf()` call presents the results in exponential notation).

However, this does not hold true for the second `test()` call, where the argument `x2` is the value of `x1` minus one million times `Double.MIN_VALUE`. It seems like `x2` should be significantly different than `x1`, but the two numbers still compare as equal. You encounter this issue in virtually every programming language because when a variable holds a very large number, subtracting a relatively small number won't make a significant difference. This is called a *rounding error* and occurs because the machine cannot hold enough information to represent a tiny change to a large number.

You might be fooled into thinking that using == produces a correct result in this case, but it does not—it is simply comparing references.

Using `equals()` when you're not working with primitives seems like the straightforward answer, but it's not as simple as that. Consider class `ValA`:

```
// operators/EqualsMethod.java
// Default equals() does not compare contents

class ValA {
  int i;
}

class ValB {
  int i;
  // Works for this example, not a complete equals():
  public boolean equals(Object o) {
    ValB rval = (ValB)o;  // Cast o to be a ValB
    return i == rval.i;
  }
}

public class EqualsMethod {
  public static void main(String[] args) {
    ValA va1 = new ValA();
    ValA va2 = new ValA();
    va1.i = va2.i = 100;
    System.out.println(va1.equals(va2));
    ValB vb1 = new ValB();
    ValB vb2 = new ValB();
    vb1.i = vb2.i = 100;
    System.out.println(vb1.equals(vb2));
  }
}
/* Output:
false
true
*/
```

In `main()`, `va1` and `va2` contain identical values for `i`, but the result of comparing them using `equals()` is `false`, which is confusing again. This happens because the default behavior of `equals()` is to compare references. To produce the desired behavior of comparing the contents, you must *override* `equals()` as in class `ValB`. `ValB.equals()` contains only the minimum necessary code to solve the problem for this example, but it is not a proper `equals()`. Note that the standard argument for `equals()` is an `Object` (not a `ValB`), so we must force `o` to be a `ValB` using a *cast*: `(ValB)o` (ordinarily you must check the type first before casting, but we'll skip that

until a later chapter). Then we can compare the two values of `i`, using == because they are primitives.

You won't learn about overriding until the Reuse chapter, and you won't learn the proper way to define `equals()` until the Appendix: Collection Topics, but being aware of the way `equals()` behaves might save you some grief in the meantime.

Most of the standard library classes override `equals()` to compare the contents of objects instead of their references.

Logical Operators

Each of the logical operators AND (&&), OR (||) and NOT (!) produce a `boolean` value of `true` or `false` based on the logical relationship of its arguments. This example uses the relational and logical operators:

```
// operators/Bool.java
// Relational and logical operators
import java.util.*;

public class Bool {
  public static void main(String[] args) {
    Random rand = new Random(47);
    int i = rand.nextInt(100);
    int j = rand.nextInt(100);
    System.out.println("i = " + i);
    System.out.println("j = " + j);
    System.out.println("i > j is " + (i > j));
    System.out.println("i < j is " + (i < j));
    System.out.println("i >= j is " + (i >= j));
    System.out.println("i <= j is " + (i <= j));
    System.out.println("i == j is " + (i == j));
    System.out.println("i != j is " + (i != j));
    // Treating an int as a boolean is not legal Java:
    //- System.out.println("i && j is " + (i && j));
    //- System.out.println("i || j is " + (i || j));
    //- System.out.println("!i is " + !i);
    System.out.println("(i < 10) && (j < 10) is "
       + ((i < 10) && (j < 10)) );
    System.out.println("(i < 10) || (j < 10) is "
```

```
        + ((i < 10) || (j < 10)) );
  }
}
/* Output:
i = 58
j = 55
i > j is true
i < j is false
i >= j is true
i <= j is false
i == j is false
i != j is true
(i < 10) && (j < 10) is false
(i < 10) || (j < 10) is false
*/
```

You can apply AND, OR, or NOT to `boolean` values only. You can't use a non-`boolean` as if it were a `boolean` in a logical expression as you can in C and C++. The failed attempts at doing this are commented out with `//-`. The subsequent expressions, however, produce `boolean` values using relational comparisons, then use logical operations on the results.

Note that a `boolean` value is automatically converted to an appropriate text form if it is used where a `String` is expected.

You can replace the definition for `int` in the preceding program with any other primitive data type except `boolean`.

Short-Circuiting

Logical operators support a phenomenon called "short-circuiting." this means the expression is evaluated only *until* the truth or falsehood of the entire expression can be unambiguously determined. As a result, the latter parts of a logical expression might not be evaluated. Here's a demonstration:

```
// operators/ShortCircuit.java
// Short-circuiting behavior with logical operators

public class ShortCircuit {
  static boolean test1(int val) {
    System.out.println("test1(" + val + ")");
    System.out.println("result: " + (val < 1));
    return val < 1;
  }
  static boolean test2(int val) {
    System.out.println("test2(" + val + ")");
    System.out.println("result: " + (val < 2));
    return val < 2;
  }
  static boolean test3(int val) {
    System.out.println("test3(" + val + ")");
    System.out.println("result: " + (val < 3));
    return val < 3;
  }
  public static void main(String[] args) {
    boolean b = test1(0) && test2(2) && test3(2);
    System.out.println("expression is " + b);
  }
}
/* Output:
test1(0)
result: true
test2(2)
result: false
expression is false
*/
```

Each test performs a comparison against the argument and returns `true` or `false`. It also prints information to show you it's being called. The tests are used in the expression:

```
test1(0) && test2(2) && test3(2)
```

You might naturally expect all three tests to execute, but the output shows otherwise. The first test produces a `true` result, so the expression evaluation continues. However, the second test produces a `false` result. This means the whole expression must be

`false`, so why continue evaluating the rest of the expression? It might be expensive. The reason for short-circuiting, in fact, is that you can get a potential performance increase if all the parts of a logical expression do not need evaluation.

Literals

Ordinarily, when you insert a literal value into a program, the compiler knows exactly what type to make it. When the type is ambiguous, you must guide the compiler by adding some extra information in the form of characters associated with the literal value. The following code shows these characters:

```
// operators/Literals.java

public class Literals {
  public static void main(String[] args) {
    int i1 = 0x2f; // Hexadecimal (lowercase)
    System.out.println(
      "i1: " + Integer.toBinaryString(i1));
    int i2 = 0X2F; // Hexadecimal (uppercase)
    System.out.println(
      "i2: " + Integer.toBinaryString(i2));
    int i3 = 0177; // Octal (leading zero)
    System.out.println(
      "i3: " + Integer.toBinaryString(i3));
    char c = 0xffff; // max char hex value
    System.out.println(
      "c: " + Integer.toBinaryString(c));
    byte b = 0x7f; // max byte hex value 10101111;
    System.out.println(
      "b: " + Integer.toBinaryString(b));
    short s = 0x7fff; // max short hex value
    System.out.println(
      "s: " + Integer.toBinaryString(s));
    long n1 = 200L; // long suffix
    long n2 = 200l; // long suffix (can be confusing)
    long n3 = 200;
    // Java 7 Binary Literals:
    byte blb = (byte)0b00110101;
    System.out.println(
```

```
      "blb: " + Integer.toBinaryString(blb));
    short bls = (short)0B0010111110101111;
    System.out.println(
      "bls: " + Integer.toBinaryString(bls));
    int bli = 0b00101111110101111101011110101111;
    System.out.println(
      "bli: " + Integer.toBinaryString(bli));
    long bll = 0b0010111111010111110101111110101111;
    System.out.println(
      "bll: " + Long.toBinaryString(bll));
    float f1 = 1;
    float f2 = 1F; // float suffix
    float f3 = 1f; // float suffix
    double d1 = 1d; // double suffix
    double d2 = 1D; // double suffix
    // (Hex and Octal also work with long)
  }
}
/* Output:
i1: 101111
i2: 101111
i3: 1111111
c: 1111111111111111
b: 1111111
s: 11111111111111
blb: 110101
bls: 10111110101111
bli: 10111110101111101011111010111
bll: 10111110101111101011111010111
*/
```

A trailing character after a literal value establishes its type. Uppercase or lowercase `L` means `long` (however, using a lowercase `l` is confusing because it can look like the number one). Uppercase or lowercase `F` means `float`. Uppercase or lowercase `D` means `double`.

Hexadecimal (base 16), which works with all the integral data types, is denoted by a leading `0x` or `0X` followed by `0-9` or `a-f` either in uppercase or lowercase. If you try to initialize a variable with a value bigger than it can hold (regardless of the numerical form of the value), the compiler will give you an error message. Notice in the preceding code the maximum possible hexadecimal values for `char`, `byte`, and

`short`. If you exceed these, the compiler will automatically make the value an `int` and declare you need a *narrowing cast* for the assignment (casts are defined later in this chapter). You'll know you've stepped over the line.

Octal (base 8) is denoted by a leading zero in the number and digits from 0-7.

Java 7 introduced binary literals, denoted by a leading `0b` or `0B`, which can initialize all integral types.

When working with integral types, it's useful to display the binary form of the results. This is easily accomplished with the `static toBinaryString()` methods from the `Integer` and `Long` classes. Notice that when passing smaller types to `Integer.toBinaryString()`, the type is automatically converted to an `int`.

Underscores in Literals

There's a thoughtful addition in Java 7: you can include underscores in numeric literals so the results are easier to read. This is especially helpful for grouping digits in large values:

```
// operators/Underscores.java

public class Underscores {
  public static void main(String[] args) {
    double d = 341_435_936.445_667;
    System.out.println(d);
    int bin = 0b0010_1111_1010_1111_1010_1111_1010_1111;
    System.out.println(Integer.toBinaryString(bin));
    System.out.printf("%x%n", bin);              // [1]
    long hex = 0x7f_e9_b7_aa;
    System.out.printf("%x%n", hex);
  }
}
/* Output:
3.41435936445667E8
10111110101111101011111010101111
2fafafaf
7fe9b7aa
*/
```

There are (reasonable) rules:

1. Single underscores only—you can't double them up.
2. No underscores at the beginning or end of a number.
3. No underscores around suffixes like `F`, `D` or `L`.
4. No around binary or hex identifiers `b` and `x`.

- **[1]** Notice the use of `%n`. If you're familiar with C-style languages, you're probably used to seeing `\n` to represent a line ending. The problem with that is it gives you a "Unix style" line ending. If you are on Windows, you must specify `\r\n` instead. This difference is a needless hassle; the programming language should take care of it for you. That's what Java has achieved with `%n`, which always produces the appropriate line ending for the platform it's running on—but only when you're using `System.out.printf()` or `System.out.format()`. For `System.out.println()` you must still use `\n`; if you use `%n`, `println()` will simply emit `%n` and not a newline.

Exponential Notation

Exponents use a notation I've always found rather dismaying:

```
// operators/Exponents.java
// "e" means "10 to the power."

public class Exponents {
  public static void main(String[] args) {
    // Uppercase and lowercase 'e' are the same:
    float expFloat = 1.39e-43f;
    expFloat = 1.39E-43f;
    System.out.println(expFloat);
    double expDouble = 47e47d; // 'd' is optional
    double expDouble2 = 47e47; // Automatically double
    System.out.println(expDouble);
  }
}
/* Output:
1.39E-43
4.7E48
*/
```

In science and engineering, `e` refers to the base of natural logarithms, approximately 2.718. (A more precise `double` value is available in Java as `Math.E`.) This is used in exponentiation expressions such as 1.39 x e-43, which means 1.39 x 2.718-43. However, when the FORTRAN programming language was invented, they decided that `e` would mean "ten to the power," an odd decision because FORTRAN was designed for science and engineering, and one would think its designers would be sensitive about introducing such an ambiguity.[57] At any rate, this custom was followed in C, C++ and now Java. So if you're used to thinking in terms of `e` as the base of natural logarithms, you must do a mental translation when you see an expression such as `1.39 e-43f` in Java; it means 1.39 x 10-43.

Note you don't need the trailing character when the compiler can figure out the appropriate type. With

```
long n3 = 200;
```

there's no ambiguity, so an `L` after the 200 is superfluous. However, with

```
float f4 = 1e-43f; // 10 to the power
```

the compiler normally takes exponential numbers as doubles, so without the trailing `f`, it will give you an error declaring you must use a cast to convert `double` to `float`.

Bitwise Operators

The bitwise operators allow you to manipulate individual bits in an integral primitive data type. Bitwise operators perform Boolean algebra on the corresponding bits in the two arguments to produce the result.

The bitwise operators come from C's low-level orientation, where you often manipulate hardware directly and must set the bits in hardware registers. Java was originally

[57] John Kirkham writes, "I started computing in 1962 using FORTRAN II on an IBM 1620. At that time, and throughout the 1960s and into the 1970s, FORTRAN was an all uppercase language. This probably started because many of the early input devices were old teletype units that used 5 bit Baudot code, which had no lowercase capability. The `E` in the exponential notation was also always uppercase and was never confused with the natural logarithm base `e`, which is always lowercase. The `E` simply stood for exponential, which was for the base of the number system used—usually 10. At the time octal was also widely used by programmers. Although I never saw it used, if I had seen an octal number in exponential notation I would have considered it to be base 8. The first time I remember seeing an exponential using a lowercase `e` was in the late 1970s and I also found it confusing. The problem arose as lowercase crept into FORTRAN, not at its beginning. We actually had functions to use if you really wanted to use the natural logarithm base, but they were all uppercase."

designed to be embedded in TV control boxes, so this low-level orientation still made sense. However, you probably won't use the bitwise operators much.

The bitwise AND operator (`&`) produces a one in the output bit if both input bits are one; otherwise, it produces a zero. The bitwise OR operator (`|`) produces a one in the output bit if either input bit is a one and produces a zero only if both input bits are zero. The bitwise EXCLUSIVE OR, or XOR (`^`), produces a one in the output bit if one or the other input bit is a one, but not both. The bitwise NOT (~, also called the *ones complement* operator) is a unary operator; it takes only one argument. (All other bitwise operators are binary operators.) Bitwise NOT produces the opposite of the input bit—a one if the input bit is zero, a zero if the input bit is one.

The bitwise operators and logical operators use the same characters, so a mnemonic device helps you remember the meanings: Because bits are "small," there is only one character in the bitwise operators.

Bitwise operators can be combined with the = sign to unite the operation and assignment: `&=`, `|=` and `^=` are all legitimate. (Because ~ is a unary operator, it cannot be combined with the = sign.)

The `boolean` type is treated as a one-bit value, so it is somewhat different. You can perform a bitwise AND, OR, and XOR, but you can't perform a bitwise NOT (presumably to prevent confusion with the logical NOT). For `booleans`, the bitwise operators have the same effect as the logical operators except they do not short circuit. Also, bitwise operations on `booleans` include an XOR logical operator that is not included under the list of "logical" operators. You cannot use `booleans` in shift expressions, which are described next.

Shift Operators

The shift operators also manipulate bits. They can be used solely with primitive, integral types. The left-shift operator (`<<`) produces the operand to the left of the operator after it is shifted to the left by the number of bits specified to the right of the operator (inserting zeroes at the lower-order bits). The signed right-shift operator (`>>`) produces the operand to the left of the operator after it is shifted to the right by the number of bits specified to the right of the operator. The signed right shift `>>` uses *sign extension*: If the value is positive, zeroes are inserted at the higher-order

bits; if the value is negative, ones are inserted at the higher-order bits. Java has also added the unsigned right shift `>>>`, which uses *zero extension*: Regardless of the sign, zeroes are inserted at the higher-order bits. This operator does not exist in C or C++.

If you shift a `char`, `byte`, or `short`, it is promoted to `int` before the shift takes place, and the result is an `int`. Only the five low-order bits of the right-hand side are used. This prevents you from shifting more than the number of bits in an `int`. If you're operating on a `long`, you'll get a `long` result. Only the six low-order bits of the right-hand side are used, so you can't shift more than the number of bits in a `long`.

Shifts can be combined with the equal sign (`<<=` or `>>=` or `>>>=`). The lvalue is replaced by the lvalue shifted by the rvalue. There is a problem, however, with the unsigned right shift combined with assignment. If you use it with `byte` or `short`, you don't get the correct results. Instead, these are promoted to `int` and right shifted, but then truncated as they are assigned back into their variables, so you get `-1` in those cases. Here's a demonstration:

```
// operators/URShift.java
// Test of unsigned right shift

public class URShift {
  public static void main(String[] args) {
    int i = -1;
    System.out.println(Integer.toBinaryString(i));
    i >>>= 10;
    System.out.println(Integer.toBinaryString(i));
    long l = -1;
    System.out.println(Long.toBinaryString(l));
    l >>>= 10;
    System.out.println(Long.toBinaryString(l));
    short s = -1;
    System.out.println(Integer.toBinaryString(s));
    s >>>= 10;
    System.out.println(Integer.toBinaryString(s));
    byte b = -1;
    System.out.println(Integer.toBinaryString(b));
    b >>>= 10;
    System.out.println(Integer.toBinaryString(b));
    b = -1;
    System.out.println(Integer.toBinaryString(b));
    System.out.println(Integer.toBinaryString(b>>>10));
```

```
  }
}
/* Output:
11111111111111111111111111111111
11111111111111111111
11111111111111111111111111111111111111111111111111111111
11111111
11111111111111111111111111111111111111111111111111111111
11111111111111111111111111111111
11111111111111111111111111111111
11111111111111111111111111111111
11111111111111111111111111111111
11111111111111111111
*/
```

In the last shift, the resulting value is not assigned back into `b`, but is printed directly, so the correct behavior occurs.

Here's an example that exercises all the operators involving bits:

```
// operators/BitManipulation.java
// Using the bitwise operators
import java.util.*;

public class BitManipulation {
  public static void main(String[] args) {
    Random rand = new Random(47);
    int i = rand.nextInt();
    int j = rand.nextInt();
    printBinaryInt("-1", -1);
    printBinaryInt("+1", +1);
    int maxpos = 2147483647;
    printBinaryInt("maxpos", maxpos);
    int maxneg = -2147483648;
    printBinaryInt("maxneg", maxneg);
    printBinaryInt("i", i);
    printBinaryInt("~i", ~i);
    printBinaryInt("-i", -i);
    printBinaryInt("j", j);
    printBinaryInt("i & j", i & j);
    printBinaryInt("i | j", i | j);
```

```
    printBinaryInt("i ^ j", i ^ j);
    printBinaryInt("i << 5", i << 5);
    printBinaryInt("i >> 5", i >> 5);
    printBinaryInt("(~i) >> 5", (~i) >> 5);
    printBinaryInt("i >>> 5", i >>> 5);
    printBinaryInt("(~i) >>> 5", (~i) >>> 5);

    long l = rand.nextLong();
    long m = rand.nextLong();
    printBinaryLong("-1L", -1L);
    printBinaryLong("+1L", +1L);
    long ll = 9223372036854775807L;
    printBinaryLong("maxpos", ll);
    long lln = -9223372036854775808L;
    printBinaryLong("maxneg", lln);
    printBinaryLong("l", l);
    printBinaryLong("~l", ~l);
    printBinaryLong("-l", -l);
    printBinaryLong("m", m);
    printBinaryLong("l & m", l & m);
    printBinaryLong("l | m", l | m);
    printBinaryLong("l ^ m", l ^ m);
    printBinaryLong("l << 5", l << 5);
    printBinaryLong("l >> 5", l >> 5);
    printBinaryLong("(~l) >> 5", (~l) >> 5);
    printBinaryLong("l >>> 5", l >>> 5);
    printBinaryLong("(~l) >>> 5", (~l) >>> 5);
  }
  static void printBinaryInt(String s, int i) {
    System.out.println(
      s + ", int: " + i + ", binary:\n   " +
      Integer.toBinaryString(i));
  }
  static void printBinaryLong(String s, long l) {
    System.out.println(
      s + ", long: " + l + ", binary:\n    " +
      Long.toBinaryString(l));
  }
}
/* Output: (First 32 Lines)
-1, int: -1, binary:
   11111111111111111111111111111111
```

```
+1, int: 1, binary:
   1
maxpos, int: 2147483647, binary:
   1111111111111111111111111111111
maxneg, int: -2147483648, binary:
   10000000000000000000000000000000
i, int: -1172028779, binary:
   10111010001001000100001010010101
~i, int: 1172028778, binary:
   01000101110110111011110101101010
-i, int: 1172028779, binary:
   01000101110110111011110101101011
j, int: 1717241110, binary:
   01100110010110110000010100010110
i & j, int: 570425364, binary:
   00010000000000000000000010100
i | j, int: -25213033, binary:
   11111100111111101000111110010111
i ^ j, int: -595638397, binary:
   11011000111111101000111110000011
i << 5, int: 1149784736, binary:
   01000100100010000101001010100000
i >> 5, int: -36625900, binary:
   11111011101000100100001000010100
(~i) >> 5, int: 36625899, binary:
   00010111011011101111101011
i >>> 5, int: 97591828, binary:
   00010111011011101111101011
(~i) >>> 5, int: 36625899, binary:
   00010111011011101111101011
                  ...
*/
```

The two methods at the end, `printBinaryInt()` and `printBinaryLong()`, take an `int` or a `long`, respectively, and display it in binary format along with a descriptive `String`. As well as demonstrating the effect of all the bitwise operators for `int` and `long`, this example also shows the minimum, maximum, `+1`, and `-1` values for `int` and `long` so you see what they look like. Note that the high bit represents the sign: `0` means positive and `1` means negative. The output for the `int` portion is displayed above.

The binary representation of the numbers is called *signed twos complement*.

Ternary `if-else` Operator

The *ternary* operator, also called the *conditional* operator, is unusual because it has three operands. It is truly an operator because it produces a value, unlike the ordinary `if-else` statement that you'll see in the next section of this chapter. The expression is of the form:

```
boolean-exp ? value0 : value1
```

If *boolean-exp* evaluates to `true`, *value0* is evaluated, and its result becomes the value produced by the operator. If *boolean-exp* is `false`, *value1* is evaluated and its result becomes the value produced by the operator.

You can also use an ordinary `if-else` statement (described later), but the ternary operator is much terser. Although C (where this operator originated) prides itself on being a terse language, and the ternary operator might have been introduced partly for efficiency, be somewhat wary of using it on an everyday basis—it's easy to produce unreadable code.

The ternary operator is different from `if-else` because it produces a value. Here's an example comparing the two:

```
// operators/TernaryIfElse.java

public class TernaryIfElse {
  static int ternary(int i) {
    return i < 10 ? i * 100 : i * 10;
  }
  static int standardIfElse(int i) {
    if(i < 10)
      return i * 100;
    else
      return i * 10;
  }
  public static void main(String[] args) {
    System.out.println(ternary(9));
    System.out.println(ternary(10));
    System.out.println(standardIfElse(9));
    System.out.println(standardIfElse(10));
  }
```

```
}
/* Output:
900
100
900
100
*/
```

The code in `ternary()` is more compact than what you'd write without the ternary operator, in `standardIfElse()`. However, `standardIfElse()` is easier to understand, and doesn't require a lot more typing. Ponder your reasons when choosing the ternary operator—it's primarily warranted when you're setting a variable to one of two values.

`String` Operator + and +=

There's one special usage of an operator in Java: The + and += operators can concatenate `String`s, as you've already seen. It seems a natural use of these operators even though it doesn't fit with the traditional way they are used.

This capability seemed like a good idea in C++, so *operator overloading* was added to C++ to allow the C++ programmer to add meanings to almost any operator. Unfortunately, operator overloading combined with some of the other restrictions in C++ turns out to be a fairly complicated feature for programmers to design into their classes. Although operator overloading would have been much simpler to implement in Java than it was in C++ (as demonstrated by the C# language, which *does* have straightforward operator overloading), this feature was still considered too complex, so Java programmers cannot implement their own overloaded operators like C++ and C# programmers can.

If an expression begins with a `String`, all operands that follow must be `String`s (remember that the compiler automatically turns a double-quoted sequence of characters into a `String`):

```
// operators/StringOperators.java

public class StringOperators {
  public static void main(String[] args) {
    int x = 0, y = 1, z = 2;
    String s = "x, y, z ";
    System.out.println(s + x + y + z);
    // Converts x to a String:
    System.out.println(x + " " + s);
    s += "(summed) = "; // Concatenation operator
    System.out.println(s + (x + y + z));
    // Shorthand for Integer.toString():
    System.out.println("" + x);
  }
}
/* Output:
x, y, z 012
0 x, y, z
x, y, z (summed) = 3
0
*/
```

Note that the output from the first print statement is `012` instead of just `3`, which you'd get if it was summing the integers. This is because the Java compiler converts `x`, `y`, and `z` into their `String` representations and concatenates those `Strings`, instead of adding them together first. The second print statement converts the leading variable into a `String`, so the `String` conversion does not depend on what comes first. Finally, you see the `+=` operator to append a `String` to `s`, and parentheses to control the order of evaluation of the expression so the `ints` are actually summed before they are displayed.

Notice the last example in `main()`: you sometimes see an empty `String` followed by a `+` and a primitive as a way to perform the conversion without calling the more cumbersome explicit method (`Integer.toString()`, here).

Common Pitfalls When Using Operators

One of the pitfalls when using operators is attempting to leave out the parentheses when you are even the least bit uncertain about how an expression will evaluate.

This is still true in Java.

An extremely common error in C and C++ looks like this:

```
while(x = y) {
  // ...
}
```

The programmer was clearly trying to test for equivalence (==) rather than do an assignment. In C and C++ the result of this assignment will always be `true` if `y` is nonzero, and you'll probably get an infinite loop. In Java, the result of this expression is not a `boolean`, but the compiler expects a `boolean` and won't convert from an `int`, so it will conveniently give you a compile-time error and catch the problem before you ever try to run the program. So the pitfall never happens in Java. (The only time you won't get a compile-time error is when `x` and `y` are `boolean`, in which case `x = y` is a legal expression, and in the preceding example, probably an error.)

A similar problem in C and C++ is using bitwise AND and OR instead of the logical versions. Bitwise AND and OR use one of the characters (`&` or `|`) while logical AND and OR use two (`&&` and `||`). Just as with = and ==, it's easy to type just one character instead of two. In Java, the compiler again prevents this, because it won't let you cavalierly use one type where it doesn't belong.

Casting Operators

The word *cast* is used in the sense of "casting into a mold." Java will automatically change one type of data into another when appropriate. For instance, if you assign an integral value to a floating point variable, the compiler will automatically convert the `int` to a `float`. Casting makes this type conversion explicit, or forces it when it wouldn't normally happen.

To perform a cast, put the desired data type inside parentheses to the left of any value, as seen here:

```
// operators/Casting.java

public class Casting {
  public static void main(String[] args) {
    int i = 200;
    long lng = (long)i;
    lng = i; // "Widening," so a cast is not required
    long lng2 = (long)200;
    lng2 = 200;
    // A "narrowing conversion":
    i = (int)lng2; // Cast required
  }
}
```

Thus, you can cast a numeric value as well as a variable. Casts may be superfluous; for example, the compiler will automatically promote an `int` value to a `long` when necessary. However, you are allowed to use superfluous casts to make a point or to clarify your code. In other situations, a cast might be essential just to get the code to compile.

In C and C++, casting can cause some headaches. In Java, casting is safe, with the exception that when you perform a so-called *narrowing conversion* (that is, when you go from a data type that can hold more information to one that doesn't hold as much), you run the risk of losing information. Here the compiler forces you to use a cast, in effect saying, "This can be a dangerous thing to do—if you want me to do it anyway you must make the cast explicit." With a *widening conversion* an explicit cast is not needed, because the new type will more than hold the information from the old type so no information is ever lost.

Java can cast any primitive type to any other primitive type, except for `boolean`, which doesn't allow any casting at all. Class types do not allow casting. To convert one to the other, there must be special methods. (You'll find out later that objects can be cast within a *family* of types; an `Oak` can be cast to a `Tree` and vice versa, but not to a foreign type such as a `Rock`.)

Truncation and Rounding

When you are performing narrowing conversions, you must pay attention to issues of truncation and rounding. For example, if you cast from a floating point value to

an integral value, what does Java do? For example, if you cast the value 29.7 to an `int`, is the resulting value 30 or 29? The answer is seen here:

```
// operators/CastingNumbers.java
// What happens when you cast a float
// or double to an integral value?

public class CastingNumbers {
  public static void main(String[] args) {
    double above = 0.7, below = 0.4;
    float fabove = 0.7f, fbelow = 0.4f;
    System.out.println("(int)above: " + (int)above);
    System.out.println("(int)below: " + (int)below);
    System.out.println("(int)fabove: " + (int)fabove);
    System.out.println("(int)fbelow: " + (int)fbelow);
  }
}
/* Output:
(int)above: 0
(int)below: 0
(int)fabove: 0
(int)fbelow: 0
*/
```

So the answer is that casting from a `float` or `double` to an integral value always truncates the number. If instead you want the result rounded, use the `round()` methods in `java.lang.Math`:

```
// operators/RoundingNumbers.java
// Rounding floats and doubles

public class RoundingNumbers {
  public static void main(String[] args) {
    double above = 0.7, below = 0.4;
    float fabove = 0.7f, fbelow = 0.4f;
    System.out.println(
      "Math.round(above): " + Math.round(above));
    System.out.println(
      "Math.round(below): " + Math.round(below));
    System.out.println(
      "Math.round(fabove): " + Math.round(fabove));
    System.out.println(
```

```
      "Math.round(fbelow): " + Math.round(fbelow));
  }
}
/* Output:
Math.round(above): 1
Math.round(below): 0
Math.round(fabove): 1
Math.round(fbelow): 0
*/
```

`round()` is part of `java.lang` so you don't need an extra import to use it.

Promotion

You'll discover that if you perform any mathematical or bitwise operations on primitive data types smaller than an `int` (that is, `char`, `byte`, or `short`), those values are promoted to `int` before performing the operations, and the resulting value is of type `int`. To assign back into the smaller type, you use a cast. (And, because you're assigning back into a smaller type, you might be losing information.) In general, the largest data type in an expression is the one that determines the size of the result of that expression. If you multiply a `float` and a `double`, the result is `double`. If you add an `int` and a `long`, the result is `long`.

Java Has No "sizeof"

In C and C++, the `sizeof()` operator tells you the number of bytes allocated for data items. The most compelling reason for `sizeof()` in C and C++ is for portability. Different data types might be different sizes on different machines, so the programmer must discover how big those types are when performing operations that are sensitive to size. For example, one computer might store integers in 32 bits, whereas another might store integers as 16 bits. Programs could store larger values in integers on the first machine. As you might imagine, portability is a huge headache for C and C++ programmers.

Java does not need a `sizeof()` operator for this purpose, because all the data types are the same size on all machines. You do not need to think about portability on this level—it is designed into the language.

A Compendium of Operators

The following example shows which primitive data types can be used with particular operators. Basically, it is the same example repeated over and over, but using different primitive data types. The file will compile without error because the lines that fail are commented out with a `//-`.

```
// operators/AllOps.java
// Tests all operators on all primitive data types
// to show which ones are accepted by the Java compiler

public class AllOps {
  // To accept the results of a Boolean test:
  void f(boolean b) {}
  void boolTest(boolean x, boolean y) {
    // Arithmetic operators:
    //- x = x * y;
    //- x = x / y;
    //- x = x % y;
    //- x = x + y;
    //- x = x - y;
    //- x++;
    //- x--;
    //- x = +y;
    //- x = -y;
    // Relational and logical:
    //- f(x > y);
    //- f(x >= y);
    //- f(x < y);
    //- f(x <= y);
    f(x == y);
    f(x != y);
    f(!y);
    x = x && y;
    x = x || y;
    // Bitwise operators:
    //- x = ~y;
    x = x & y;
    x = x | y;
    x = x ^ y;
    //- x = x << 1;
```

```
    //- x = x >> 1;
    //- x = x >>> 1;
    // Compound assignment:
    //- x += y;
    //- x -= y;
    //- x *= y;
    //- x /= y;
    //- x %= y;
    //- x <<= 1;
    //- x >>= 1;
    //- x >>>= 1;
    x &= y;
    x ^= y;
    x |= y;
    // Casting:
    //- char c = (char)x;
    //- byte b = (byte)x;
    //- short s = (short)x;
    //- int i = (int)x;
    //- long l = (long)x;
    //- float f = (float)x;
    //- double d = (double)x;
  }
  void charTest(char x, char y) {
    // Arithmetic operators:
    x = (char)(x * y);
    x = (char)(x / y);
    x = (char)(x % y);
    x = (char)(x + y);
    x = (char)(x - y);
    x++;
    x--;
    x = (char) + y;
    x = (char) - y;
    // Relational and logical:
    f(x > y);
    f(x >= y);
    f(x < y);
    f(x <= y);
    f(x == y);
    f(x != y);
    //- f(!x);
```

```
    //- f(x && y);
    //- f(x || y);
    // Bitwise operators:
    x= (char)~y;
    x = (char)(x & y);
    x  = (char)(x | y);
    x = (char)(x ^ y);
    x = (char)(x << 1);
    x = (char)(x >> 1);
    x = (char)(x >>> 1);
    // Compound assignment:
    x += y;
    x -= y;
    x *= y;
    x /= y;
    x %= y;
    x <<= 1;
    x >>= 1;
    x >>>= 1;
    x &= y;
    x ^= y;
    x |= y;
    // Casting:
    //- boolean bl = (boolean)x;
    byte b = (byte)x;
    short s = (short)x;
    int i = (int)x;
    long l = (long)x;
    float f = (float)x;
    double d = (double)x;
  }
  void byteTest(byte x, byte y) {
    // Arithmetic operators:
    x = (byte)(x* y);
    x = (byte)(x / y);
    x = (byte)(x % y);
    x = (byte)(x + y);
    x = (byte)(x - y);
    x++;
    x--;
    x = (byte) + y;
    x = (byte) - y;
```

```
  // Relational and logical:
  f(x > y);
  f(x >= y);
  f(x < y);
  f(x <= y);
  f(x == y);
  f(x != y);
  //- f(!x);
  //- f(x && y);
  //- f(x || y);
  // Bitwise operators:
  x = (byte)~y;
  x = (byte)(x & y);
  x = (byte)(x | y);
  x = (byte)(x ^ y);
  x = (byte)(x << 1);
  x = (byte)(x >> 1);
  x = (byte)(x >>> 1);
  // Compound assignment:
  x += y;
  x -= y;
  x *= y;
  x /= y;
  x %= y;
  x <<= 1;
  x >>= 1;
  x >>>= 1;
  x &= y;
  x ^= y;
  x |= y;
  // Casting:
  //- boolean bl = (boolean)x;
  char c = (char)x;
  short s = (short)x;
  int i = (int)x;
  long l = (long)x;
  float f = (float)x;
  double d = (double)x;
}
void shortTest(short x, short y) {
  // Arithmetic operators:
  x = (short)(x * y);
```

```
x = (short)(x / y);
x = (short)(x % y);
x = (short)(x + y);
x = (short)(x - y);
x++;
x--;
x = (short) + y;
x = (short) - y;
// Relational and logical:
f(x > y);
f(x >= y);
f(x < y);
f(x <= y);
f(x == y);
f(x != y);
//- f(!x);
//- f(x && y);
//- f(x || y);
// Bitwise operators:
x = (short) ~ y;
x = (short)(x & y);
x = (short)(x | y);
x = (short)(x ^ y);
x = (short)(x << 1);
x = (short)(x >> 1);
x = (short)(x >>> 1);
// Compound assignment:
x += y;
x -= y;
x *= y;
x /= y;
x %= y;
x <<= 1;
x >>= 1;
x >>>= 1;
x &= y;
x ^= y;
x |= y;
// Casting:
//- boolean bl = (boolean)x;
char c = (char)x;
byte b = (byte)x;
```

```
    int i = (int)x;
    long l = (long)x;
    float f = (float)x;
    double d = (double)x;
  }
  void intTest(int x, int y) {
    // Arithmetic operators:
    x = x * y;
    x = x / y;
    x = x % y;
    x = x + y;
    x = x - y;
    x++;
    x--;
    x = +y;
    x = -y;
    // Relational and logical:
    f(x > y);
    f(x >= y);
    f(x < y);
    f(x <= y);
    f(x == y);
    f(x != y);
    //- f(!x);
    //- f(x && y);
    //- f(x || y);
    // Bitwise operators:
    x = ~y;
    x = x & y;
    x = x | y;
    x = x ^ y;
    x = x << 1;
    x = x >> 1;
    x = x >>> 1;
    // Compound assignment:
    x += y;
    x -= y;
    x *= y;
    x /= y;
    x %= y;
    x <<= 1;
    x >>= 1;
```

```
    x >>>= 1;
    x &= y;
    x ^= y;
    x |= y;
    // Casting:
    //- boolean bl = (boolean)x;
    char c = (char)x;
    byte b = (byte)x;
    short s = (short)x;
    long l = (long)x;
    float f = (float)x;
    double d = (double)x;
  }
  void longTest(long x, long y) {
    // Arithmetic operators:
    x = x * y;
    x = x / y;
    x = x % y;
    x = x + y;
    x = x - y;
    x++;
    x--;
    x = +y;
    x = -y;
    // Relational and logical:
    f(x > y);
    f(x >= y);
    f(x < y);
    f(x <= y);
    f(x == y);
    f(x != y);
    //- f(!x);
    //- f(x && y);
    //- f(x || y);
    // Bitwise operators:
    x = ~y;
    x = x & y;
    x = x | y;
    x = x ^ y;
    x = x << 1;
    x = x >> 1;
    x = x >>> 1;
```

```
    // Compound assignment:
    x += y;
    x -= y;
    x *= y;
    x /= y;
    x %= y;
    x <<= 1;
    x >>= 1;
    x >>>= 1;
    x &= y;
    x ^= y;
    x |= y;
    // Casting:
    //- boolean bl = (boolean)x;
    char c = (char)x;
    byte b = (byte)x;
    short s = (short)x;
    int i = (int)x;
    float f = (float)x;
    double d = (double)x;
  }
  void floatTest(float x, float y) {
    // Arithmetic operators:
    x = x * y;
    x = x / y;
    x = x % y;
    x = x + y;
    x = x - y;
    x++;
    x--;
    x = +y;
    x = -y;
    // Relational and logical:
    f(x > y);
    f(x >= y);
    f(x < y);
    f(x <= y);
    f(x == y);
    f(x != y);
    //- f(!x);
    //- f(x && y);
    //- f(x || y);
```

```
    // Bitwise operators:
    //- x = ~y;
    //- x = x & y;
    //- x = x | y;
    //- x = x ^ y;
    //- x = x << 1;
    //- x = x >> 1;
    //- x = x >>> 1;
    // Compound assignment:
    x += y;
    x -= y;
    x *= y;
    x /= y;
    x %= y;
    //- x <<= 1;
    //- x >>= 1;
    //- x >>>= 1;
    //- x &= y;
    //- x ^= y;
    //- x |= y;
    // Casting:
    //- boolean bl = (boolean)x;
    char c = (char)x;
    byte b = (byte)x;
    short s = (short)x;
    int i = (int)x;
    long l = (long)x;
    double d = (double)x;
  }
  void doubleTest(double x, double y) {
    // Arithmetic operators:
    x = x * y;
    x = x / y;
    x = x % y;
    x = x + y;
    x = x - y;
    x++;
    x--;
    x = +y;
    x = -y;
    // Relational and logical:
    f(x > y);
```

```
    f(x >= y);
    f(x < y);
    f(x <= y);
    f(x == y);
    f(x != y);
    //- f(!x);
    //- f(x && y);
    //- f(x || y);
    // Bitwise operators:
    //- x = ~y;
    //- x = x & y;
    //- x = x | y;
    //- x = x ^ y;
    //- x = x << 1;
    //- x = x >> 1;
    //- x = x >>> 1;
    // Compound assignment:
    x += y;
    x -= y;
    x *= y;
    x /= y;
    x %= y;
    //- x <<= 1;
    //- x >>= 1;
    //- x >>>= 1;
    //- x &= y;
    //- x ^= y;
    //- x |= y;
    // Casting:
    //- boolean bl = (boolean)x;
    char c = (char)x;
    byte b = (byte)x;
    short s = (short)x;
    int i = (int)x;
    long l = (long)x;
    float f = (float)x;
  }
}
```

Note that `boolean` is limited. You can assign it the values `true` and `false`, and you can test it for truth or falsehood, but you cannot add `boolean`s or perform any other type of operation on them.

In `char`, `byte`, and `short`, you see the effect of promotion with the arithmetic operators. Each arithmetic operation on any of those types produces an `int` result, which must be explicitly cast back to the original type (a narrowing conversion that might lose information) to assign back to that type. With `int` values, however, you do not need a cast, because everything is already an `int`. Don't be lulled into thinking everything is safe, though. If you multiply two `int`s that are big enough, you'll overflow the result. The following example demonstrates this:

```
// operators/Overflow.java
// Surprise! Java lets you overflow

public class Overflow {
  public static void main(String[] args) {
    int big = Integer.MAX_VALUE;
    System.out.println("big = " + big);
    int bigger = big * 4;
    System.out.println("bigger = " + bigger);
  }
}
/* Output:
big = 2147483647
bigger = -4
*/
```

You get no errors or warnings from the compiler, and no exceptions at run time.

Compound assignments do *not* require casts for `char`, `byte`, or `short`, even though they are performing promotions that have the same results as the direct arithmetic operations. On the other hand, the lack of a cast certainly simplifies the code.

Except for `boolean`, any primitive type can be cast to any other primitive type. Again, you must be aware of the effect of a narrowing conversion when casting to a smaller type; otherwise, you might unknowingly lose information during the cast.

Summary

If you've had experience with any languages that use C-like syntax, you see that the operators in Java are so similar there is virtually no learning curve. If you found this

chapter challenging, make sure you view the free multimedia presentation Thinking in C[58] seminar.

[58] https://archive.org/details/ThinkingInC

Control Flow

> A program manipulates its world and makes choices. In Java you make choices with execution control statements.

Java uses all of C's execution control statements, so if you've programmed with C or C++, most of what you see is familiar. Most procedural programming languages have some kind of control statements, and there is often overlap among languages. In Java, the keywords include `if-else`, `while`, `do-while`, `for`, `return`, `break`, and a selection statement called `switch`. Java does not, however, support the much-maligned `goto` (which can still be the most expedient way to solve certain types of problems). You can still do a `goto`-like jump, but it is much more constrained than `goto` in other languages.

`true` and `false`

All conditional statements use the truth or falsehood of a conditional expression to determine the execution path. An example of a conditional expression is `a == b`. This uses the conditional operator `==` to see if the value of `a` is equivalent to the value of `b`. The expression returns `true` or `false`. If you display the result of a conditional expression, it produces the strings "true" and "false" representing the `boolean` values:

```
// control/TrueFalse.java

public class TrueFalse {
  public static void main(String[] args) {
    System.out.println(1 == 1);
    System.out.println(1 == 2);
  }
}
/* Output:
true
false
*/
```

Any of the relational operators you've seen in the previous chapter can produce a conditional statement. Note that Java doesn't allow you to use a number as a `boolean`, even though it's allowed in C and C++ (where truth is nonzero and falsehood is zero). To use a non-`boolean` in a `boolean` test, such as `if(a)`, you must first convert it to a `boolean` value by using a conditional expression, such as `if(a != 0)`.

if-else

The `if-else` statement is the most basic way to control program flow. The `else` is optional, so you can use `if` in two forms:

```
if(Boolean-expression)
  statement
```

or

```
if(Boolean-expression)
  statement
else
  statement
```

The *Boolean-expression* must produce a `boolean` result. The *statement* is either a simple statement terminated by a semicolon, or a compound statement: a group of simple statements enclosed in braces. Whenever the word "*statement*" is used, it always implies that the statement can be simple or compound.

As an example of `if-else`, here is a `test()` method that tells whether a guess is above, below, or equivalent to a target number:

```
// control/IfElse.java

public class IfElse {
  static int result = 0;
  static void test(int testval, int target) {
    if(testval > target)
      result = +1;
    else if(testval < target)          // [1]
      result = -1;
    else
      result = 0; // Match
  }
  public static void main(String[] args) {
    test(10, 5);
    System.out.println(result);
    test(5, 10);
    System.out.println(result);
    test(5, 5);
    System.out.println(result);
  }
}
/* Output:
1
-1
0
*/
```

- **[1]** "`else if`" is not a new keyword, but just an `else` followed by a new `if` statement.

Although Java, like C and C++ before it, is a "free-form" language, it is conventional to indent the body of a control flow statement so the reader can easily determine where it begins and ends.

Iteration Statements

Looping is controlled by `while`, `do-while` and `for`, which are sometimes called *iteration statements*. Such statements repeat until the controlling *Boolean-expression* evaluates to `false`. The form for a `while` loop is:

```
while(Boolean-expression)
  statement
```

The *Boolean-expression* is evaluated once at the beginning of the loop and again before each further iteration of the statement.

This generates random numbers until a particular condition is met:

```
// control/WhileTest.java
// Demonstrates the while loop

public class WhileTest {
  static boolean condition() {
    boolean result = Math.random() < 0.99;
    System.out.print(result + ", ");
    return result;
  }
  public static void main(String[] args) {
    while(condition())
      System.out.println("Inside 'while'");
    System.out.println("Exited 'while'");
  }
}
/* Output: (First and Last 5 Lines)
true, Inside 'while'
true, Inside 'while'
true, Inside 'while'
true, Inside 'while'
true, Inside 'while'
...________...________...________...________...
true, Inside 'while'
true, Inside 'while'
true, Inside 'while'
true, Inside 'while'
false, Exited 'while'
*/
```

The `condition()` method uses the `static` method `random()` in the `Math` library, which generates a `double` value between 0 and 1 (It includes 0, but not 1). The `result` value comes from the comparison operator `<`, which produces a `boolean` result. The conditional expression for the `while` says: "repeat the statements in the body as long as `condition()` returns `true`."

do-while

The form for `do-while` is

```
do
  statement
while(Boolean-expression);
```

The sole difference between `while` and `do-while` is that the statement of the `do-while` always executes at least once, even if the expression evaluates to `false` the first time. In a `while`, if the conditional is `false` the first time the statement never executes. In practice, `do-while` is less common than `while`.

for

A `for` loop is perhaps the most commonly used form of iteration. This loop performs initialization before the first iteration. Then it performs conditional testing and, at the end of each iteration, some form of "stepping." The form of the `for` loop is:

```
for(initialization; Boolean-expression; step)
  statement
```

Any of the expressions *initialization*, *Boolean-expression* or *step* can be empty. The expression is tested before each iteration, and as soon as it evaluates to `false`, execution will continue at the line following the `for` statement. At the end of each loop, the *step* executes.

`for` loops are often used for "counting" tasks:

```
// control/ListCharacters.java
// List all the lowercase ASCII letters

public class ListCharacters {
  public static void main(String[] args) {
    for(char c = 0; c < 128; c++)
      if(Character.isLowerCase(c))
        System.out.println("value: " + (int)c +
          " character: " + c);
  }
}
/* Output: (First 10 Lines)
value: 97 character: a
value: 98 character: b
value: 99 character: c
value: 100 character: d
value: 101 character: e
value: 102 character: f
value: 103 character: g
value: 104 character: h
value: 105 character: i
value: 106 character: j
                ...
*/
```

Note that the variable `c` is defined when it is used, inside the control expression of the `for` loop, rather than at the beginning of `main()`. The scope of `c` is the statement controlled by the `for`.

Traditional procedural languages like C require that all variables be defined at the beginning of a block. When the compiler creates a block, it can allocate space for those variables. In Java and C++, you can spread your variable declarations throughout the block, defining them when you need them. This allows a more natural coding style and makes code easier to understand.[59]

This program uses the `java.lang.Character` "wrapper" class, which not only wraps the primitive `char` type in an object, but also provides other utilities. Here, `static isLowerCase()` detects whether the character in question is a lowercase letter.

[59] In earlier languages, a large number of decisions were based on making life easier for the compiler writers. You'll see that in modern languages, most design decisions make life easier for the *users* of the language, although at times there are compromises—which usually end up becoming regrets on the part of the language designers.

The Comma Operator

The comma *operator* (not the comma *separator*, used to separate definitions and method arguments) has only one use in Java: in the control expression of a `for` loop. In both the initialization and step portions of the control expression, you can have a number of statements separated by commas, and those statements are evaluated sequentially.

Using the comma operator, you can define multiple variables within a `for` statement, but they must be of the same type:

```
// control/CommaOperator.java

public class CommaOperator {
  public static void main(String[] args) {
    for(int i = 1, j = i + 10; i < 5; i++, j = i * 2) {
      System.out.println("i = " + i + " j = " + j);
    }
  }
}
/* Output:
i = 1 j = 11
i = 2 j = 4
i = 3 j = 6
i = 4 j = 8
*/
```

The `int` definition in the `for` statement covers both `i` and `j`. The initialization portion can have any number of definitions *of one type*. The ability to define variables in a control expression is limited to the `for` loop. You cannot use this approach with any of the other selection or iteration statements.

In both the initialization and step portions, the statements are evaluated in sequential order.

For-in Syntax

Java 5 introduced a more succinct `for` syntax, for use with arrays and collections (you'll learn more about these in the Arrays and Collections chapters). This is

sometimes called the *enhanced for*, and much of the documentation you'll see call this the *for-each syntax*, but Java 8 added `forEach()` which is heavily used. This confuses the terminology, and so I take some license and instead call it *for-in* (in Python, for example, you actually say `for x in sequence`, so there is reasonable precedent). Just keep in mind you might see it termed differently elsewhere.

The *for-in* produces each item for you, automatically, so you don't create an `int` to count through a sequence of items. For example, suppose you have an array of `float` and you'd like to select each element in that array:

```
// control/ForInFloat.java
import java.util.*;

public class ForInFloat {
  public static void main(String[] args) {
    Random rand = new Random(47);
    float[] f = new float[10];
    for(int i = 0; i < 10; i++)
      f[i] = rand.nextFloat();
    for(float x : f)
      System.out.println(x);
  }
}
/* Output:
0.72711575
0.39982635
0.5309454
0.0534122
0.16020656
0.57799757
0.18847865
0.4170137
0.51660204
0.73734957
*/
```

The array is populated using the old `for` loop, because it must be accessed with an index. You see the *for-in* syntax in the line:

```
for(float x : f) {
```

This defines a variable `x` of type `float` and sequentially assigns each element of `f` to `x`.

Any method that returns an array is a *for-in* candidate. For example, the `String` class has a method `toCharArray()` that returns an array of `char`, so you can easily iterate through the characters in a `String`:

```
// control/ForInString.java

public class ForInString {
  public static void main(String[] args) {
    for(char c : "An African Swallow".toCharArray())
      System.out.print(c + " ");
  }
}
/* Output:
A n   A f r i c a n   S w a l l o w
*/
```

As you'll see in the Collections chapter, *for-in* also works with any object that is `Iterable`.

Many `for` statements involve stepping through a sequence of integral values, like this:

```
for(int i = 0; i < 100; i++)
```

For these, the *for-in* syntax won't work unless you create an array of `int` first. To simplify this task, I've created a method called `range()` in `onjava.Range` that automatically generates the appropriate array.

The Implementation Hiding chapter introduces *static imports*. However, you don't need to know those details to begin using this library. You see the `static import` syntax in the `import` line here:

```
// control/ForInInt.java
import static onjava.Range.*;

public class ForInInt {
  public static void main(String[] args) {
    for(int i : range(10)) // 0..9
      System.out.print(i + " ");
    System.out.println();
    for(int i : range(5, 10)) // 5..9
      System.out.print(i + " ");
    System.out.println();
    for(int i : range(5, 20, 3)) // 5..20 step 3
      System.out.print(i + " ");
    System.out.println();
    for(int i : range(20, 5, -3)) // Count down
      System.out.print(i + " ");
    System.out.println();
  }
}
/* Output:
0 1 2 3 4 5 6 7 8 9
5 6 7 8 9
5 8 11 14 17
20 17 14 11 8
*/
```

The `range()` method is *overloaded*, which means the same method name can be used with different argument lists (you'll learn about overloading soon). The first overloaded form of `range()` just starts at zero and produces values up to but not including the top end of the range. The second form starts at the first value and goes until one less than the second, and the third form has a step value so it increases by that value. The fourth form shows you can also count down. `range()` is a simple version of what's called a *generator*, which you'll see later in the book.

`range()` allows *for-in* syntax in more places, and thus arguably increases readability.

Notice that `System.out.print()` does not emit a newline, so you can output a line in pieces.

The *for-in* syntax doesn't just save time when writing code. More importantly, it is far easier to read and says *what* you are trying to do (get each element of the array) rather than giving the details of *how* you are doing it ("I'm creating this index so I

can use it to select each of the array elements."). The *for-in* syntax is preferred in this book.

return

Several keywords represent *unconditional branching*, which means the branch happens without any test. These include `return`, `break`, `continue`, and a way to jump to a labeled statement, similar to `goto` in other languages.

The `return` keyword has two purposes: It specifies what value a method will return (if it doesn't have a `void` return value) and it causes the current method to exit, returning that value. The `test()` method from `IfElse.java` can be rewritten to take advantage of this:

```
// control/TestWithReturn.java

public class TestWithReturn {
  static int test(int testval, int target) {
    if(testval > target)
      return +1;
    if(testval < target)
      return -1;
    return 0; // Match
  }
  public static void main(String[] args) {
    System.out.println(test(10, 5));
    System.out.println(test(5, 10));
    System.out.println(test(5, 5));
  }
}
/* Output:
1
-1
0
*/
```

There's no need for `else`, because the method will not continue after executing a `return`.

If you do not have a `return` statement in a method that returns `void`, there's an implicit `return` at the end of that method, so it's not always necessary to include a `return` statement. However, if your method states it will return anything other than `void`, you must ensure every code path will return a value.

`break` and `continue`

You can also control the flow of the loop inside the body of any of the iteration statements by using `break` and `continue`. `break` quits the loop without executing the rest of the statements in the loop. `continue` stops the execution of the current iteration and goes back to the beginning of the loop to begin the next iteration.

Here you see `break` and `continue` within `for` and `while` loops:

```
// control/BreakAndContinue.java
// Break and continue keywords
import static onjava.Range.*;

public class BreakAndContinue {
  public static void main(String[] args) {
    for(int i = 0; i < 100; i++) {        // [1]
      if(i == 74) break; // Out of for loop
      if(i % 9 != 0) continue; // Next iteration
      System.out.print(i + " ");
    }
    System.out.println();
    // Using for-in:
    for(int i : range(100)) {             // [2]
      if(i == 74) break; // Out of for loop
      if(i % 9 != 0) continue; // Next iteration
      System.out.print(i + " ");
    }
    System.out.println();
    int i = 0;
    // An "infinite loop":
    while(true) {                         // [3]
      i++;
      int j = i * 27;
      if(j == 1269) break; // Out of loop
      if(i % 10 != 0) continue; // Top of loop
```

```
      System.out.print(i + " ");
    }
  }
}
/* Output:
0 9 18 27 36 45 54 63 72
0 9 18 27 36 45 54 63 72
10 20 30 40
*/
```

- **[1]** The value of `i` never gets to 100 because the `break` statement breaks out of the loop when `i` is 74. Normally, you use a `break` like this only if you don't know when the terminating condition occurs. The `continue` statement takes execution back to the top of the iteration loop (thus incrementing `i`) whenever `i` is not evenly divisible by 9. When it is, the value is displayed.
- **[2]** *For-in* produces the same results.
- **[3]** The "infinite" `while` loop could continue forever because its conditional expression is always `true`, but the `break` breaks out of the loop. Notice the `continue` statement moves control back to the top of the loop without completing anything after the `continue`. Thus, display occurs only when the value of `i` is divisible by `10`. In the output, the value `0` is shown because `0 % 9` produces `0`.

Another form of the infinite loop is `for(;;)`. The compiler treats both `while(true)` and `for(;;)` the same way, so the one you use is a matter of programming taste.

The Infamous "Goto"

The goto keyword[60] was present in programming languages from the beginning. Indeed, `goto` was the genesis of program control in assembly language[61]: "If condition A, then jump here; otherwise, jump there." If you read the assembly code that is ultimately generated by virtually any compiler, you'll see that program control contains many jumps (the Java compiler produces its own "assembly code," but this code is run by the Java Virtual Machine rather than directly on a hardware CPU).

[60] https://en.wikipedia.org/wiki/Goto
[61] https://en.wikipedia.org/wiki/Assembly_language

A goto is a jump at the source-code level, and that's what brought it into disrepute. If a program will always jump from one point to another, isn't there some way to reorganize the code so the flow of control is not so jumpy? goto fell into true disfavor with the publication of the famous "Goto considered harmful" paper by Edsger Dijkstra, and since then goto-bashing is a popular sport, with advocates of the cast-out keyword scurrying for cover.

As is typical in situations like this, the middle ground is the most fruitful. The problem is not goto, but the overuse of goto; in rare situations goto is actually the best way to structure control flow.

Although goto is a reserved word in Java, it is not used in the language—Java has no goto. However, it does have something that looks a bit like a jump tied in with the break and continue keywords. It's not a jump but rather a way to break out of an iteration statement. The reason it's often thrown in with discussions of goto is because it uses the same mechanism: a label.

A label is an identifier followed by a colon, like this:

```
label1:
```

The *only* place a label is useful in Java is right before an iteration statement. And that means *right* before—it does no good to put any other statement between the label and the iteration. And the sole reason to put a label before an iteration is if you're going to nest another iteration or a switch (which you'll learn about shortly) inside it. That's because the break and continue keywords will normally interrupt only the current loop, but when used with a label, they'll interrupt the loops up to where the label exists:

```
label1:
outer-iteration {
  inner-iteration {
    // ...
    break;           // [1]
    // ...
    continue;        // [2]
    // ...
    continue label1; // [3]
    // ...
    break label1;    // [4]
```

```
  }
}
```

- **[1]** The `break` breaks out of the inner iteration and you end up in the outer iteration.
- **[2]** The `continue` moves back to the beginning of the inner iteration.
- **[3]** The `continue label1` breaks out of the inner iteration *and* the outer iteration, all the way back to `label1`. Then it does in fact continue the iteration, but starting at the outer iteration.
- **[4]** The `break label1` also breaks all the way out to `label1`, but it does not reenter the iteration. It actually does break out of both iterations.

Labeled `break` and labeled `continue` can also be used with `for` loops:

```
// control/LabeledFor.java
// For loops with "labeled break"/"labeled continue."

public class LabeledFor {
  public static void main(String[] args) {
    int i = 0;
    outer: // Can't have statements here
    for(; true ;) { // infinite loop
      inner: // Can't have statements here
      for(; i < 10; i++) {
        System.out.println("i = " + i);
        if(i == 2) {
          System.out.println("continue");
          continue;
        }
        if(i == 3) {
          System.out.println("break");
          i++; // Otherwise i never
               // gets incremented.
          break;
        }
        if(i == 7) {
          System.out.println("continue outer");
          i++; // Otherwise i never
               // gets incremented.
          continue outer;
        }
```

```
        if(i == 8) {
          System.out.println("break outer");
          break outer;
        }
        for(int k = 0; k < 5; k++) {
          if(k == 3) {
            System.out.println("continue inner");
            continue inner;
          }
        }
      }
    }
    // Can't break or continue to labels here
  }
}
/* Output:
i = 0
continue inner
i = 1
continue inner
i = 2
continue
i = 3
break
i = 4
continue inner
i = 5
continue inner
i = 6
continue inner
i = 7
continue outer
i = 8
break outer
*/
```

Note that `break` breaks out of the `for` loop, and that the increment expression doesn't occur until the end of the pass through the `for` loop. Because `break` skips the increment expression, the increment is performed directly in the case of `i == 3`. The `continue outer` statement in the case of `i == 7` also goes to the top of the loop and also skips the increment, so it too is incremented directly.

Without the `break outer` statement, there's no way to get out of the outer loop from within an inner loop, because `break` by itself can only break out of the innermost loop. (The same is true for `continue`.)

In cases where breaking out of a loop also exits the method, you can simply use a `return`.

This demonstrates labeled `break` and `continue` statements with `while` loops:

```
// control/LabeledWhile.java
// "While" with "labeled break" and "labeled continue."

public class LabeledWhile {
  public static void main(String[] args) {
    int i = 0;
    outer:
    while(true) {
      System.out.println("Outer while loop");
      while(true) {
        i++;
        System.out.println("i = " + i);
        if(i == 1) {
          System.out.println("continue");
          continue;
        }
        if(i == 3) {
          System.out.println("continue outer");
          continue outer;
        }
        if(i == 5) {
          System.out.println("break");
          break;
        }
        if(i == 7) {
          System.out.println("break outer");
          break outer;
        }
      }
    }
  }
}
/* Output:
```

```
Outer while loop
i = 1
continue
i = 2
i = 3
continue outer
Outer while loop
i = 4
i = 5
break
Outer while loop
i = 6
i = 7
break outer
*/
```

The same rules hold true for `while`:

1. A plain `continue` goes to the top of the innermost loop and continues.
2. A labeled `continue` goes to the label and reenters the loop right after that label.
3. A `break` "drops out of the bottom" of the loop.
4. A labeled `break` drops out of the bottom of the end of the loop denoted by the label.

It's important to remember that the *only* reason to use labels in Java is when you have nested loops and you must `break` or `continue` through more than one nested level.

Labeled `break` and labeled `continue` have turned out to be speculative features (with little or no precedent in preceding languages) that are relatively unused, so you rarely see them in code.

In Dijkstra's "Goto considered harmful" paper, what he specifically objected to was the labels, not the `goto`. He observed that the number of bugs seems to increase with the number of labels in a program,[62] and that labels and `goto`s make programs difficult to analyze. Note that Java labels don't suffer from this problem because they are constrained in their placement and can't be used to transfer control in an ad hoc manner. This is a case where a language feature is made more valuable by restricting its use.

[62]Note that this seems a hard assertion to support, and may very well be an example of the cognitive bias called the correlation-causation fallacy.

switch

The switch is sometimes called a *selection statement*. A switch selects from among pieces of code based on the value of an integral expression. Its general form is:

```
switch(integral-selector) {
  case integral-value1 : statement; break;
  case integral-value2 : statement; break;
  case integral-value3 : statement; break;
  case integral-value4 : statement; break;
  case integral-value5 : statement; break;
  // ...
  default: statement;
}
```

Integral-selector is an expression that produces an integral value. The switch compares the result of *integral-selector* to each *integral-value*. If it finds a match, the corresponding *statement* (a single statement or multiple statements; braces are not required) executes. If no match occurs, the default *statement* executes.

Notice in the preceding definition that each case ends with a break. This jumps to the end of the switch body. The example shows the conventional way to build a switch statement, but the break is optional. If it is missing, the code for the following case statements executes until a break is encountered. Although you don't usually want this kind of behavior, it can be useful to an experienced programmer. Note that the last statement, following the default, doesn't have a break because the execution just falls through to where the break would have taken it anyway. You can put a break at the end of the default statement with no harm if you considered it important for style's sake.

The switch statement is a clean way to implement multi-way selection (i.e., selecting from among several different execution paths), but before Java 7 it only worked with a selector that evaluates to an integral value, such as int or char. For non-integral types (except String in Java 7 and beyond), you must use a series of if statements. At the end of the next chapter, you'll see that the enum feature helps ease this restriction, as enums are designed to work nicely with switch.

Here's an example that creates letters randomly and determines whether they're vowels or consonants:

```
// control/VowelsAndConsonants.java
// Demonstrates the switch statement
import java.util.*;

public class VowelsAndConsonants {
  public static void main(String[] args) {
    Random rand = new Random(47);
    for(int i = 0; i < 100; i++) {
      int c = rand.nextInt(26) + 'a';
      System.out.print((char)c + ", " + c + ": ");
      switch(c) {
        case 'a':
        case 'e':
        case 'i':
        case 'o':
        case 'u': System.out.println("vowel");
                  break;
        case 'y':
        case 'w': System.out.println("Sometimes vowel");
                  break;
        default:  System.out.println("consonant");
      }
    }
  }
}
/* Output: (First 13 Lines)
y, 121: Sometimes vowel
n, 110: consonant
z, 122: consonant
b, 98: consonant
r, 114: consonant
n, 110: consonant
y, 121: Sometimes vowel
g, 103: consonant
c, 99: consonant
f, 102: consonant
o, 111: vowel
w, 119: Sometimes vowel
z, 122: consonant
                  ...
*/
```

`Random.nextInt(26)` generates a value between 0 and 25 so you need only add an offset of `a` to produce the lowercase letters. The single-quoted characters in the `case` statements also produce integral values used for comparison.

Notice how the `cases` can be "stacked" on top of each other to provide multiple matches for a particular piece of code. It's essential to put the `break` statement at the end of a particular case; otherwise, control will drop through and continue processing on the next case.

In the statement:

```
int c = rand.nextInt(26) + 'a';
```

`Random.nextInt()` produces a random `int` value from 0 to 25, which is added to the value of `a`. This means `a` is automatically converted to an `int` to perform the addition.

To print `c` as a character, it must be cast to `char`; otherwise, you'll produce integral output.

Switching on Strings

Java 7 added the ability to switch on `Strings` as well as integral values. This example shows both the old way you would choose from a set of `String` possibilities, and the new way, using `switch`:

```
// control/StringSwitch.java

public class StringSwitch {
  public static void main(String[] args) {
    String color = "red";
    // Old way: using if-then
    if("red".equals(color)) {
      System.out.println("RED");
    } else if("green".equals(color)) {
      System.out.println("GREEN");
    } else if("blue".equals(color)) {
      System.out.println("BLUE");
    } else if("yellow".equals(color)) {
```

```
      System.out.println("YELLOW");
    } else {
      System.out.println("Unknown");
    }
    // New way: Strings in switch
    switch(color) {
      case "red":
        System.out.println("RED");
        break;
      case "green":
        System.out.println("GREEN");
        break;
      case "blue":
        System.out.println("BLUE");
        break;
      case "yellow":
        System.out.println("YELLOW");
        break;
      default:
        System.out.println("Unknown");
        break;
    }
  }
}
/* Output:
RED
RED
*/
```

Once you understand `switch`, this syntax is a logical extension. The result is cleaner and easier to understand and maintain.

As a second example of switching on `Strings`, let's revisit `Math.random()`. Does it produce a value from zero to one, inclusive or exclusive of the value "1"? In math lingo, is it (0,1), or [0,1], or (0,1] or [0,1)? (The square bracket means "includes," whereas the parenthesis means "doesn't include.")

Here's a test program that *might* provide the answer. All command-line arguments are passed as `String` objects, so we can `switch` on the argument to decide what to do. There's one problem: the user might not provide any arguments, so indexing into the `args` array would cause the program to fail. To fix this, we check the `length`

of the array and, if it's zero, we use an empty `String`, otherwise we select the first element in the `args` array:

```
// control/RandomBounds.java
// Does Math.random() produce 0.0 and 1.0?
// {java RandomBounds lower}
import onjava.*;

public class RandomBounds {
  public static void main(String[] args) {
    new TimedAbort(3);
    switch(args.length == 0 ? "" : args[0]) {
      case "lower":
        while(Math.random() != 0.0)
          ; // Keep trying
        System.out.println("Produced 0.0!");
        break;
      case "upper":
        while(Math.random() != 1.0)
          ; // Keep trying
        System.out.println("Produced 1.0!");
        break;
      default:
        System.out.println("Usage:");
        System.out.println("\tRandomBounds lower");
        System.out.println("\tRandomBounds upper");
        System.exit(1);
    }
  }
}
```

To run the program, you type a command line of either:

```
java RandomBounds lower
```

or

```
java RandomBounds upper
```

Using the `TimedAbort` class from the `onjava` package, the program aborts after three seconds, so it would appear that `Math.random()` never produces either 0.0 or 1.0. But this is where such an experiment can be deceiving. If you consider the number of

different double fractions between 0 and 1, the likelihood of reaching any one value experimentally might exceed the lifetime of one computer, or even one experimenter. It turns out 0.0 *is* included in the output of `Math.random()`, while 1.0 is not. In math lingo, it is [0,1). You must be careful to analyze your experiments and to understand their limitations.

Summary

This chapter concludes our exploration of fundamental features that appear in most programming languages: calculation, operator precedence, type casting, and selection and iteration. Now you're ready to begin taking steps that move you closer to the world of object-oriented and functional programming. The next chapter will cover the important issues of initialization and cleanup of objects, followed in the subsequent chapter by the essential concept of implementation hiding.

Housekeeping

> "Unsafe" programming is one of the major culprits that makes programming expensive.

Two of these safety issues are *initialization* and *cleanup*. Many C bugs occur when the programmer forgets to initialize a variable. This is especially true with libraries when users don't know how to initialize a library component, or even that they must. Cleanup is a special problem because it's easy to forget about an element when you're done with it because it no longer concerns you. Thus, the resources used by that element are retained and you can easily end up running out of resources (most notably, memory).

C++ introduced the concept of a *constructor*, a special method automatically called when an object is created. Java adopted the constructor, and also has a *garbage collector* that automatically releases memory resources when they're no longer used. This chapter examines the issues of initialization and cleanup, and their support in Java.

Guaranteed Initialization with the Constructor

You can imagine creating a method called `initialize()` for every class you write. The name is a hint it should be called before using the object. Unfortunately, this means the user must remember to call that method. In Java, the class designer can guarantee initialization of every object by writing a *constructor*. If a class has a constructor, Java automatically calls that constructor when an object is created, before users can even get their hands on it. So initialization is guaranteed.

The next challenge is what to name this method. There are two issues. The first is that any name you use could clash with a name you might like to use as an element in the class. The second is that because the compiler is responsible for calling the

constructor, it must always know which method to call. The C++ solution seems the easiest and most logical, so it's also used in Java: The name of the constructor is the name of the class. It makes sense that such a method is called automatically during initialization.

Here's a simple class with a constructor:

```
// housekeeping/SimpleConstructor.java
// Demonstration of a simple constructor

class Rock {
  Rock() { // This is the constructor
    System.out.print("Rock ");
  }
}

public class SimpleConstructor {
  public static void main(String[] args) {
    for(int i = 0; i < 10; i++)
      new Rock();
  }
}
/* Output:
Rock Rock Rock Rock Rock Rock Rock Rock Rock Rock
*/
```

Now, when an object is created:

```
new Rock();
```

storage is allocated and the constructor is called. The constructor guarantees that the object is properly initialized before you can get your hands on it.

Note that the coding style of making the first letter of all methods lowercase does not apply to constructors because the name of the constructor must match the name of the class *exactly*.

In C++, a constructor that takes no parameters is called the *default constructor*. This term was in use for many years before Java appeared, but for some reason the Java designers decided to use the term *no-arg* constructor, which I found awkward and unnecessary, so I resisted and attempted to continue using "default constructor." Java

8 has introduced the use of `default` as a keyword for method definitions, which would confuse the issue. Fortunately, the Java documentation has started using the phrase *zero-argument constructor*, which I will use in this book.

Like any method, the constructor can have parameters that specify how an object is created. The preceding example can easily be changed so the constructor takes a parameter:

```
// housekeeping/SimpleConstructor2.java
// Constructors can have parameters

class Rock2 {
  Rock2(int i) {
    System.out.print("Rock " + i + " ");
  }
}

public class SimpleConstructor2 {
  public static void main(String[] args) {
    for(int i = 0; i < 8; i++)
      new Rock2(i);
  }
}
/* Output:
Rock 0 Rock 1 Rock 2 Rock 3 Rock 4 Rock 5 Rock 6 Rock 7
*/
```

If a class `Tree` has a constructor that takes a single integer parameter denoting the height of the tree, you create a `Tree` object like this:

```
Tree t = new Tree(12);  // 12-foot tree
```

If `Tree(int)` is your only constructor, the compiler won't let you create a `Tree` object any other way.

Constructors eliminate an important class of problems and make the code easier to read. In the preceding code fragment, for example, you don't see an explicit call to some `initialize()` method that is conceptually separate from creation. In Java, creation and initialization are unified concepts—you can't have one without the other.

The constructor is an unusual type of method because it has no return value. This is distinctly different from a `void` return value, where the method returns nothing but you still have the option to make it return something else. Constructors return nothing and you don't have an option (the `new` expression does return a reference to the newly created object, but the constructor itself has no return value). If there were a return value, and if you could select your own, the compiler would somehow need to know what to do with that return value.

Method Overloading

Names are an important feature in any programming language. When you create an object, you give a name to a region of storage. A method is a name for an action. You refer to all objects, fields, and methods by using names. Well-chosen names create a system that is easier for people to understand and change. It's a lot like writing prose—the goal is to communicate with your readers.

A problem arises when mapping the concept of *nuance* in human language onto a programming language. Often, the same word expresses several different meanings—it's *overloaded*. This is useful, especially when it comes to trivial differences. You say, "Wash the shirt," "Wash the car," and "Wash the dog." It would be silly to be forced to say, "shirtWash the shirt," "carWash the car," and "dogWash the dog" just so the listener doesn't have to make any distinction about the action performed. Most human languages are redundant, so even if you miss a few words, you can still determine the meaning. You don't need unique identifiers—you can deduce meaning from context.

Most programming languages (C in particular) require a unique identifier for each method (often called *functions* in those languages). So you could *not* have a function called `print()` for printing integers and another called `print()` for printing floats—each function requires a unique name.

In Java (and C++), another factor forces the overloading of method names: the constructor. Because the constructor's name is predetermined by the name of the class, there can be only one constructor name. But how do you create an object in more than one way? For example, suppose you build a class that can initialize itself in a standard way or by reading information from a file. You need two constructors, the zero-argument constructor and one that takes a `String` as an parameter, which

is the name of the file from which to initialize the object. Both are constructors, so they must have the same name—the name of the class. Thus, *method overloading* is necessary to allow the same method name with different parameter types. And although method overloading is essential for constructors, it's a general convenience and can be used with any method.

Here's an example that shows both overloaded constructors and overloaded methods:

```
// housekeeping/Overloading.java
// Both constructor and ordinary method overloading

class Tree {
  int height;
  Tree() {
    System.out.println("Planting a seedling");
    height = 0;
  }
  Tree(int initialHeight) {
    height = initialHeight;
    System.out.println("Creating new Tree that is " +
      height + " feet tall");
  }
  void info() {
    System.out.println(
      "Tree is " + height + " feet tall");
  }
  void info(String s) {
    System.out.println(
      s + ": Tree is " + height + " feet tall");
  }
}

public class Overloading {
  public static void main(String[] args) {
    for(int i = 0; i < 5; i++) {
      Tree t = new Tree(i);
      t.info();
      t.info("overloaded method");
    }
    // Calls overloaded constructor:
    new Tree();
  }
```

```
}
/* Output:
Creating new Tree that is 0 feet tall
Tree is 0 feet tall
overloaded method: Tree is 0 feet tall
Creating new Tree that is 1 feet tall
Tree is 1 feet tall
overloaded method: Tree is 1 feet tall
Creating new Tree that is 2 feet tall
Tree is 2 feet tall
overloaded method: Tree is 2 feet tall
Creating new Tree that is 3 feet tall
Tree is 3 feet tall
overloaded method: Tree is 3 feet tall
Creating new Tree that is 4 feet tall
Tree is 4 feet tall
overloaded method: Tree is 4 feet tall
Planting a seedling
*/
```

A `Tree` object can be created either as a seedling, with no argument, or as a plant grown in a nursery, with an existing height. To support this, there is a zero-argument constructor, plus a constructor that takes the existing height.

You might also want to call the `info()` method in more than one way. For example, if you have an extra message you want printed, you can use `info(String)`. If you have nothing more to say, you just use `info()`. It would seem strange to give two separate names to what is obviously the same concept. With method overloading, you use the same name for both.

Distinguishing Overloaded Methods

If methods have the same name, how can Java know which method you mean? There's a simple rule: Each overloaded method must take a unique list of argument types.

If you think about this for a second, it makes sense. How else could a programmer tell the difference between two methods that have the same name, other than by the types of their parameters?

Even differences in the ordering of parameters are sufficient to distinguish two methods, although you don't normally take this approach because it produces difficult-to-maintain code:

```
// housekeeping/OverloadingOrder.java
// Overloading based on parameter order

public class OverloadingOrder {
  static void f(String s, int i) {
    System.out.println("String: " + s + ", int: " + i);
  }
  static void f(int i, String s) {
    System.out.println("int: " + i + ", String: " + s);
  }
  public static void main(String[] args) {
    f("String first", 11);
    f(99, "Int first");
  }
}
/* Output:
String: String first, int: 11
int: 99, String: Int first
*/
```

The two `f()` methods have identical parameters, but the order is different, and that's what makes them distinct.

Overloading with Primitives

A primitive can be automatically *promoted* from a smaller type to a larger one, and this can be slightly confusing in combination with overloading. Here's a demonstration of what happens when a primitive is handed to an overloaded method:

```
// housekeeping/PrimitiveOverloading.java
// Promotion of primitives and overloading

public class PrimitiveOverloading {
  void f1(char x) { System.out.print("f1(char) "); }
  void f1(byte x) { System.out.print("f1(byte) "); }
  void f1(short x) { System.out.print("f1(short) "); }
  void f1(int x) { System.out.print("f1(int) "); }
  void f1(long x) { System.out.print("f1(long) "); }
  void f1(float x) { System.out.print("f1(float) "); }
  void f1(double x) { System.out.print("f1(double) "); }

  void f2(byte x) { System.out.print("f2(byte) "); }
  void f2(short x) { System.out.print("f2(short) "); }
  void f2(int x) { System.out.print("f2(int) "); }
  void f2(long x) { System.out.print("f2(long) "); }
  void f2(float x) { System.out.print("f2(float) "); }
  void f2(double x) { System.out.print("f2(double) "); }

  void f3(short x) { System.out.print("f3(short) "); }
  void f3(int x) { System.out.print("f3(int) "); }
  void f3(long x) { System.out.print("f3(long) "); }
  void f3(float x) { System.out.print("f3(float) "); }
  void f3(double x) { System.out.print("f3(double) "); }

  void f4(int x) { System.out.print("f4(int) "); }
  void f4(long x) { System.out.print("f4(long) "); }
  void f4(float x) { System.out.print("f4(float) "); }
  void f4(double x) { System.out.print("f4(double) "); }

  void f5(long x) { System.out.print("f5(long) "); }
  void f5(float x) { System.out.print("f5(float) "); }
  void f5(double x) { System.out.print("f5(double) "); }

  void f6(float x) { System.out.print("f6(float) "); }
  void f6(double x) { System.out.print("f6(double) "); }

  void f7(double x) { System.out.print("f7(double) "); }

  void testConstVal() {
    System.out.print("5: ");
    f1(5);f2(5);f3(5);f4(5);f5(5);f6(5);f7(5);
```

```
    System.out.println();
  }
  void testChar() {
    char x = 'x';
    System.out.print("char: ");
    f1(x);f2(x);f3(x);f4(x);f5(x);f6(x);f7(x);
    System.out.println();
  }
  void testByte() {
    byte x = 0;
    System.out.print("byte: ");
    f1(x);f2(x);f3(x);f4(x);f5(x);f6(x);f7(x);
    System.out.println();
  }
  void testShort() {
    short x = 0;
    System.out.print("short: ");
    f1(x);f2(x);f3(x);f4(x);f5(x);f6(x);f7(x);
    System.out.println();
  }
  void testInt() {
    int x = 0;
    System.out.print("int: ");
    f1(x);f2(x);f3(x);f4(x);f5(x);f6(x);f7(x);
    System.out.println();
  }
  void testLong() {
    long x = 0;
    System.out.print("long: ");
    f1(x);f2(x);f3(x);f4(x);f5(x);f6(x);f7(x);
    System.out.println();
  }
  void testFloat() {
    float x = 0;
    System.out.print("float: ");
    f1(x);f2(x);f3(x);f4(x);f5(x);f6(x);f7(x);
    System.out.println();
  }
  void testDouble() {
    double x = 0;
    System.out.print("double: ");
    f1(x);f2(x);f3(x);f4(x);f5(x);f6(x);f7(x);
```

```
    System.out.println();
  }
  public static void main(String[] args) {
    PrimitiveOverloading p =
      new PrimitiveOverloading();
    p.testConstVal();
    p.testChar();
    p.testByte();
    p.testShort();
    p.testInt();
    p.testLong();
    p.testFloat();
    p.testDouble();
  }
}
/* Output:
5: f1(int) f2(int) f3(int) f4(int) f5(long) f6(float)
f7(double)
char: f1(char) f2(int) f3(int) f4(int) f5(long)
f6(float) f7(double)
byte: f1(byte) f2(byte) f3(short) f4(int) f5(long)
f6(float) f7(double)
short: f1(short) f2(short) f3(short) f4(int) f5(long)
f6(float) f7(double)
int: f1(int) f2(int) f3(int) f4(int) f5(long) f6(float)
f7(double)
long: f1(long) f2(long) f3(long) f4(long) f5(long)
f6(float) f7(double)
float: f1(float) f2(float) f3(float) f4(float)
f5(float) f6(float) f7(double)
double: f1(double) f2(double) f3(double) f4(double)
f5(double) f6(double) f7(double)
*/
```

The constant value 5 is treated as an `int`, so if an overloaded method is available that takes an `int`, it is used. In all other cases, if you have a data type that is smaller than the parameter in the method, that data type is promoted. `char` produces a slightly different effect: If it doesn't find an exact `char` match, it is promoted to `int`.

What happens if your argument is *bigger* than the parameter expected by the method? An example gives the answer:

```
// housekeeping/Demotion.java
// Demotion of primitives

public class Demotion {
  void f1(double x) {
    System.out.println("f1(double)");
  }
  void f2(float x) { System.out.println("f2(float)"); }
  void f3(long x) { System.out.println("f3(long)"); }
  void f4(int x) { System.out.println("f4(int)"); }
  void f5(short x) { System.out.println("f5(short)"); }
  void f6(byte x) { System.out.println("f6(byte)"); }
  void f7(char x) { System.out.println("f7(char)"); }

  void testDouble() {
    double x = 0;
    System.out.println("double argument:");
    f1(x);
    f2((float)x);
    f3((long)x);
    f4((int)x);
    f5((short)x);
    f6((byte)x);
    f7((char)x);
  }
  public static void main(String[] args) {
    Demotion p = new Demotion();
    p.testDouble();
  }
}
/* Output:
double argument:
f1(double)
f2(float)
f3(long)
f4(int)
f5(short)
f6(byte)
f7(char)
*/
```

If your argument is wider than what the method expects, you must perform a

narrowing conversion with a cast. If you don't do this, the compiler will issue an error message.

Overloading on Return Values

It is common to wonder, "Why only class names and method argument lists? Why not distinguish between methods based on their return values?" For example, these two methods, which have the same name and arguments, are easily distinguished from each other:

```
void f() {}
int f() { return 1; }
```

This might work fine as long as the compiler could unequivocally determine the meaning from the context, as in `int x = f()`. However, you can also call a method and ignore the return value. This is *calling a method for its side effects* because you don't care about the return value, but instead want the other effects of the method call. So if you call the method this way:

```
f();
```

how can Java determine which `f()` should be called? And how could someone reading the code see it? Because of this sort of problem, you cannot use return value types to distinguish overloaded methods. To support new features, Java 8 has added better guessing in some specialized situations, but in general it doesn't work.

Zero-Argument Constructors

As mentioned previously, a zero-argument constructor is one without arguments, used to create a "default object." If you create a class that has no constructors, the compiler will automatically create a zero-argument constructor for you. For example:

```
// housekeeping/DefaultConstructor.java

class Bird {}

public class DefaultConstructor {
  public static void main(String[] args) {
    Bird b = new Bird(); // Default!
  }
}
```

The expression

```
new Bird()
```

creates a new object and calls the zero-argument constructor, even though one was not explicitly defined. Without it, you have no method call to build the object. However, if you define any constructors (with or without arguments), the compiler will *not* synthesize one for you:

```
// housekeeping/NoSynthesis.java

class Bird2 {
  Bird2(int i) {}
  Bird2(double d) {}
}

public class NoSynthesis {
  public static void main(String[] args) {
    //- Bird2 b = new Bird2(); // No default
    Bird2 b2 = new Bird2(1);
    Bird2 b3 = new Bird2(1.0);
  }
}
```

If you say:

```
new Bird2()
```

the compiler complains it cannot find a constructor that matches. When you don't put in any constructors, it's as if the compiler says, "You are bound to need *some* constructor, so let me make one for you." But if you write a constructor, the compiler says, "You've written a constructor so you know what you're doing; if you didn't put in a default it's because you meant to leave it out."

The `this` Keyword

For two objects of the same type called `a` and `b`, you might wonder how you can call a method `peel()` for both those objects:

```
// housekeeping/BananaPeel.java

class Banana { void peel(int i) { /* ... */ } }

public class BananaPeel {
  public static void main(String[] args) {
    Banana a = new Banana(),
           b = new Banana();
    a.peel(1);
    b.peel(2);
  }
}
```

If there's only one method called `peel()`, how can that method know whether it's called for the object `a` or `b`?

The compiler does some undercover work so you can write code this way. There's a secret first argument passed to the method `peel()`, and that argument is the reference to the object that's being manipulated. So the two method calls become something like:

```
Banana.peel(a, 1);
Banana.peel(b, 2);
```

This is internal and you can't write these expressions and get the compiler to accept them, but it gives you an idea of what's happening.

Suppose you're inside a method and you'd like to get the reference to the current object. However, that reference is passed *secretly* by the compiler—it's not in the argument list. Conveniently, there's a keyword: `this`. The `this` keyword can be used only inside a non-`static` method. When you call a method on an object, `this` produces a reference to that object. You can treat the reference just like any other object reference. If you're calling a method of your class from within another method of your class, don't use `this`. Simply call the method. The current `this` reference is automatically used for the other method. Thus you can say:

```
// housekeeping/Apricot.java

public class Apricot {
  void pick() { /* ... */ }
  void pit() { pick(); /* ... */ }
}
```

Inside `pit()`, you *could* say `this.pick()` but there's no need.[63] The compiler does it for you automatically. The `this` keyword is used only for those special cases where you must explicitly use the reference to the current object. For example, it's often used in `return` statements for returning the reference to the current object:

```
// housekeeping/Leaf.java
// Simple use of the "this" keyword

public class Leaf {
  int i = 0;
  Leaf increment() {
    i++;
    return this;
  }
  void print() {
    System.out.println("i = " + i);
  }
  public static void main(String[] args) {
    Leaf x = new Leaf();
    x.increment().increment().increment().print();
  }
}
/* Output:
i = 3
*/
```

Because `increment()` returns the reference to the current object via the `this` keyword, multiple operations can easily be performed on the same object.

The `this` keyword is also useful for passing the current object to another method:

[63] Some people will obsessively put `this` in front of every method call and field reference, arguing that it makes it "clearer and more explicit." Don't do it. There's a reason that we use high-level languages. They do things for us. If you use `this` when it's not necessary, you confuse and annoy everyone who reads your code because all the rest of the code they've read *won't* use `this` everywhere. People expect `this` only when it is necessary. Following a consistent and straightforward coding style saves time and money.

```
// housekeeping/PassingThis.java

class Person {
  public void eat(Apple apple) {
    Apple peeled = apple.getPeeled();
    System.out.println("Yummy");
  }
}

class Peeler {
  static Apple peel(Apple apple) {
    // ... remove peel
    return apple; // Peeled
  }
}

class Apple {
  Apple getPeeled() { return Peeler.peel(this); }
}

public class PassingThis {
  public static void main(String[] args) {
    new Person().eat(new Apple());
  }
}
/* Output:
Yummy
*/
```

`Apple` calls `Peeler.peel()`, a foreign utility method to perform an operation that, for some reason, must be external to `Apple` (perhaps the external method can be applied across many different classes, and you don't want to repeat the code). To pass itself to the foreign method, it must use `this`.

Calling Constructors from Constructors

When you write several constructors for a class, there are times when you'd like to call one constructor from another to avoid duplicating code. You can make such a call by using the `this` keyword.

Normally, when you say `this`, it is in the sense of "this object" or "the current object," and by itself it produces the reference to the current object. Inside a constructor, the `this` keyword takes on a different meaning when you give it an argument list. It makes an explicit call to the constructor that matches that argument list, and so is a straightforward way to call other constructors:

```
// housekeeping/Flower.java
// Calling constructors with "this"

public class Flower {
  int petalCount = 0;
  String s = "initial value";
  Flower(int petals) {
    petalCount = petals;
    System.out.println(
      "Constructor w/ int arg only, petalCount= "
      + petalCount);
  }
  Flower(String ss) {
    System.out.println(
      "Constructor w/ String arg only, s = " + ss);
    s = ss;
  }
  Flower(String s, int petals) {
    this(petals);
    //- this(s); // Can't call two!
    this.s = s; // Another use of "this"
    System.out.println("String & int args");
  }
  Flower() {
    this("hi", 47);
    System.out.println("Zero-argument constructor");
  }
  void printPetalCount() {
    //- this(11); // Not inside non-constructor!
    System.out.println(
      "petalCount = " + petalCount + " s = "+ s);
  }
  public static void main(String[] args) {
    Flower x = new Flower();
    x.printPetalCount();
  }
```

```
}
/* Output:
Constructor w/ int arg only, petalCount= 47
String & int args
Zero-argument constructor
petalCount = 47 s = hi
*/
```

The constructor `Flower(String s, int petals)` shows that, while you can call one constructor using `this`, you cannot call two. In addition, the constructor call must be the first thing you do, or you'll get a compiler error message.

This example also shows another way you'll see `this` used. Because the name of the argument `s` and the name of the member data `s` are the same, there's an ambiguity. You can resolve it using `this.s`, to say that you're referring to the member data. You'll often see this form used in Java code, and it's used in numerous places in this book.

In `printPetalCount()`, the compiler won't let you call a constructor from inside any method other than a constructor.

The Meaning of `static`

With the `this` keyword in mind, you can more fully understand what it means to make a method `static`: There is no `this` for that particular method. You cannot call non-`static` methods from inside `static` methods[64] (although the reverse is possible), and you can call a `static` method for the class itself, without any object. In fact, that's the primary purpose of a `static` method. It's as if you're creating the equivalent of a global method. However, global methods are not permitted in Java, and putting the `static` method inside a class allows it access to other `static` methods and to `static` fields.

Some people argue that `static` methods are not object-oriented because they do have the semantics of a global method; with a `static` method, you don't send a message to an object because there's no `this`. This is a fair argument, and if you find yourself using a *lot* of `static` methods, rethink your strategy. However, `statics`

[64]The one case this is possible is if you pass a reference to an object into the `static` method (the `static` method could also create its own object). Then, via the reference (which is now effectively `this`), you can call non-`static` methods and access non-`static` fields. But typically, to do something like this, you'll just make an ordinary, non-`static` method.

are pragmatic, and there are times when you genuinely need them, so whether or not they are "proper OOP" should be left to the theoreticians.

Cleanup: Finalization and Garbage Collection

Programmers know the importance of initialization, but often forget the importance of cleanup. After all, who cleans up an `int`? But "letting go" of an object once you're done with it is not always safe. Java does have the garbage collector to reclaim the memory of objects that are no longer used. But now consider an unusual case: Suppose your object allocates "special" memory without using `new`. The garbage collector only knows how to release memory allocated *with* `new`, so it won't know how to release the object's "special" memory. To handle this case, Java provides a method called `finalize()` you can define for your class.

Here's how it's *supposed* to work: When the garbage collector is ready to release the storage used for your object, it first calls `finalize()`, and only on the next garbage-collection pass will it reclaim the object's memory. So if you choose to use `finalize()`, it gives you the ability to perform some important cleanup *at the time of garbage collection.*

`finalize()` is a potential programming pitfall because some programmers, especially C++ programmers, might initially mistake it for the *destructor* in C++, a function that is *always* called when an object is destroyed. It is important to distinguish between C++ and Java here, because in C++, *objects always get destroyed* (in a bug-free program), whereas in Java, objects do not always get garbage collected. Or, put another way:

- Your objects might not get garbage collected.
- Garbage collection is not destruction.

If there is some activity to perform before you no longer need an object, you must perform that activity yourself. Java has no destructor or similar concept, so you must create an ordinary method to perform this cleanup. For example, suppose that in the process of creating your object, it draws itself on the screen. If you don't explicitly erase its image from the screen, it might never get cleaned up. If you put some kind of erasing functionality inside `finalize()`, then if an object is garbage collected and

`finalize()` is called (and there's no guarantee this will happen), the image will first be removed from the screen, but if it isn't, the image will remain.

You might find that the storage for an object never gets released because your program never nears the point of running out of storage. If your program completes and the garbage collector never gets around to releasing the storage for any of your objects, that storage is returned to the operating system *en masse* as the program exits. This is a good thing, because garbage collection has some overhead, and if you never do it, you never incur that expense.

What is `finalize()` for?

So, if you can't use `finalize()` as a general-purpose cleanup method, what good is it?

A third point to remember is:

- Garbage collection is only about memory.

That is, the sole reason for the existence of the garbage collector is to recover memory your program is no longer using. So any activity associated with garbage collection, most notably your `finalize()` method, must also be only about memory and its deallocation.

Does this mean that if your object contains other objects, `finalize()` should explicitly release those objects? Well, no—the garbage collector takes care of the release of all object memory regardless of how the object is created. The need for `finalize()` is limited to special cases where your object can allocate storage in some way other than creating an object. But, you might observe, everything in Java is an object, so how can this be?

It would seem that `finalize()` is in place because you might do something C-like by allocating memory using a mechanism other than the normal one in Java. This can happen primarily through *native methods*, which are a way to call non-Java code from Java (Native methods are covered in Appendix B of the electronic 2nd edition of *Thinking in Java*, available at www.OnJava8.com[65]). C and C++ are the only languages currently supported by native methods. Those languages can call

[65] http://www.OnJava8.com

subprograms in other languages, so you can effectively call anything. Inside the non-Java code, C's `malloc()` family of functions might be called to allocate storage, and unless you call `free()`, that storage is not released, causing a memory leak. However, `free()` is a C and C++ function, so you'd call it in a native method inside your `finalize()`.

After reading this, you probably get the idea you won't use `finalize()` much.[66] You're correct; it is not the appropriate place for normal cleanup to occur. So where should normal cleanup be performed?

You Must Perform Cleanup

To clean up an object, the user of that object must call a cleanup method when cleanup is desired. This sounds pretty straightforward, but it collides a bit with the C++ concept of the destructor. In C++, all objects are destroyed. Or rather, all objects *should be* destroyed. If the C++ object is created as a local (i.e., on the stack—not possible in Java), the destruction happens at the closing curly brace of the scope where the object was created. If the object was created using `new` (like in Java), the destructor is called when the programmer calls the C++ operator `delete` (which doesn't exist in Java). If the C++ programmer forgets to call `delete`, the destructor is never called, and you have a memory leak, plus the other parts of the object never get cleaned up. This kind of bug can be very difficult to track down, and is one of the compelling reasons to move from C++ to Java.

In contrast, Java doesn't allow you to create local objects—you must always use `new`. But in Java, there's no "delete" for releasing the object, because the garbage collector releases the storage for you. So from a simplistic standpoint, you can say that because of garbage collection, Java has no destructor. You'll see as this book progresses, however, that the presence of a garbage collector does not remove the need for or the utility of destructors. (And never call `finalize()` directly, so that's not a solution.) If you want some kind of cleanup performed other than storage release, you must *still* explicitly call an appropriate method in Java: the equivalent of a C++ destructor but without the convenience.

Remember that neither garbage collection nor finalization is guaranteed. If the JVM

[66] Joshua Bloch goes further in his section titled "avoid finalizers": "Finalizers are unpredictable, often dangerous, and generally unnecessary." *Effective Java Programming Language Guide*, p. 20 (Addison-Wesley, 2001).

isn't close to running out of memory, it might not waste time recovering memory through garbage collection.

The Termination Condition

In general, you can't rely on `finalize()` being called, and you must create separate "cleanup" methods and call them explicitly. So it appears that `finalize()` is only useful for obscure memory cleanup that most programmers will never use. However, there is an interesting use of `finalize()` that does not rely on it being called every time. This is the verification of the *termination condition*[67] of an object.

When you're no longer interested in an object—when it's ready to be cleaned up—that object should be in a state whereby its memory can be safely released. For example, if the object represents an open file, that file should be closed by the programmer before the object is garbage collected. If any portions of the object are not properly cleaned up, you have a bug in your program that can be very difficult to find. `finalize()` can eventually discover this condition, even if it isn't always called. If one of the finalizations happens to reveal the bug, you discover the problem, which is all you really care about.

Here's a simple example of how you might use it:

```
// housekeeping/TerminationCondition.java
// Using finalize() to detect an object that
// hasn't been properly cleaned up
import onjava.*;

class Book {
  boolean checkedOut = false;
  Book(boolean checkOut) {
    checkedOut = checkOut;
  }
  void checkIn() {
    checkedOut = false;
  }
  @SuppressWarnings("deprecation")
  @Override public void finalize() {
    if(checkedOut)
```

[67] A term coined by Bill Venners (*www.Artima.com*) during a seminar that he and I gave together.

```
      System.out.println("Error: checked out");
    // Normally, you'll also do this:
    // super.finalize(); // Call the base-class version
  }
}

public class TerminationCondition {
  public static void main(String[] args) {
    Book novel = new Book(true);
    // Proper cleanup:
    novel.checkIn();
    // Drop the reference, forget to clean up:
    new Book(true);
    // Force garbage collection & finalization:
    System.gc();
    new Nap(1); // One second delay
  }
}
/* Output:
Error: checked out
*/
```

The termination condition says that all `Book` objects must be checked in before they are garbage collected, but `main()` doesn't check in one of the books. Without `finalize()` to verify the termination condition, this can be a difficult bug to find.

You see here the use of `@Override`. The `@` indicates an *annotation*, which is extra information about the code. Here, it tells the compiler you aren't *accidentally* redefining the `finalize()` method that is in every object—you know you're doing it. The compiler makes sure you haven't misspelled the method name, and that the method actually exists in the base class. The annotation is also a reminder to the reader. `@Override` was introduced in Java 5, and modified in Java 7, and I use it throughout the book.

This also introduces the `@SuppressWarnings("deprecation")` annotation, which prevents versions of the JDK later than 8 from reminding you that `finalize()` is no longer officially supported in the language and that, while you can still use it, it's not recommended (it's an admission that they didn't understand the issues and made a mistake when they included `finalize()`—you'll see more of these throughout the book).

Note that `System.gc()` is used to force finalization. But even if it isn't, it's highly probable that the errant `Book` will eventually be discovered through repeated executions of the program (assuming the program allocates enough storage to cause the garbage collector to execute).

Assume that the base-class version of `finalize()` does something important, and call it using `super`, as you see in `Book.finalize()`. It is commented because it requires exception handling, which we haven't covered yet.

How a Garbage Collector Works

If you come from a programming language where allocating objects on the heap is expensive, you might naturally assume that Java's scheme of allocating everything (except primitives) on the heap is also expensive. However, the garbage collector can significantly *increase* the speed of object creation. This might sound a bit odd at first—that storage release affects storage allocation—but it's the way some JVMs work, and it means allocating storage for heap objects in Java can be nearly as fast as creating storage *on the stack* in other languages.

For example, you can think of the C++ heap as a yard where each object stakes out its own piece of turf. This real estate can become abandoned sometime later and must be reused. In some JVMs, the Java heap is different; it's more like a conveyor belt that moves forward every time you allocate a new object. This means object storage allocation is remarkably rapid. The "heap pointer" is simply moved forward into virgin territory, so it's effectively the same as C++'s stack allocation. (There's some extra overhead for bookkeeping, but it's nothing like searching for storage.)

You might observe that the heap isn't in fact a conveyor belt, and if you treat it that way, you'll start paging memory—moving it on and off disk, so it appears there's more memory than actually exists. Paging significantly impacts performance. Eventually, after you create enough objects, you'll run out of memory. The trick is that the garbage collector steps in and collects the garbage. While it does this, it also compacts all the objects in the heap so you've effectively moved the "heap pointer" closer to the beginning of the conveyor belt and farther away from a page fault. The garbage collector rearranges things and makes it possible for the high-speed, infinite-free-heap model while allocating storage.

To understand garbage collection in Java, it's helpful to learn how garbage-collection

schemes work in other systems. A simple but slow garbage-collection technique is called *reference counting*. This means each object contains a reference counter, and every time a reference is attached to that object, the reference count is increased. Every time a reference goes out of scope or is set to `null`, the reference count is decreased. Thus, managing reference counts is a small but constant overhead that happens throughout the lifetime of your program. The garbage collector moves through the entire list of objects, and when it finds one with a reference count of zero it releases that storage (however, reference counting schemes often release an object as soon as the count goes to zero). The one drawback is that if objects circularly refer to each other they can have nonzero reference counts while still being garbage. Locating such self-referential groups requires significant extra work for the garbage collector. Reference counting is commonly used to explain one kind of garbage collection, but it doesn't seem to be in any JVM implementations.

Faster schemes do not use reference counting, but are based instead on the idea that any non-dead object must ultimately be traceable back to a reference that lives either on the stack or in static storage. The chain might go through several layers of objects. Thus, if you start in the stack and in the static storage area and walk through all the references, you'll find all the live objects. For each reference you find, you must trace into the object that it points to, then follow all the references in *that* object, tracing into the objects they point to, etc., until you've moved through the entire web that originated with the reference on the stack or in static storage. Each object you move through must still be alive. Note there is no problem with detached self-referential groups—these are simply not found, and are therefore automatically garbage.

In the approach described here, the JVM uses an *adaptive* garbage-collection scheme, and what it does with the live objects it locates depends on the variant currently used. One of these variants is *stop-and-copy*. This means—for reasons that become apparent—the program is first stopped (this is not a background collection scheme). Then, each live object is copied from one heap to another, leaving behind all the garbage. In addition, as the objects are copied into the new heap, they are packed end-to-end, thus compacting the new heap (and allowing new storage to be reeled off the end as previously described).

When an object is moved from one place to another, all references that point to the object must be changed. The reference that goes from the stack or the static storage area to the object can be changed right away, but there can be other references pointing to this object encountered later during the "walk." These are fixed up as

they are found (imagine a table that maps old addresses to new ones).

There are two issues that make these so-called "copy collectors" inefficient. The first is the idea that you have two heaps and you slosh all the memory back and forth between these two separate heaps, maintaining twice as much memory as you actually need. Some JVMs deal with this by allocating the heap in chunks as needed and copying from one chunk to another.

The second issue is the copying process itself. Once your program becomes stable, it might be generating little or no garbage. Despite that, a copy collector will still copy all the memory from one place to another, which is wasteful. To prevent this, some JVMs detect that no new garbage is generated and switch to a different variant (this is the "adaptive" part). This other variant is called *mark-and-sweep*, and it's what earlier versions of Sun's JVM used all the time. For general use, mark-and-sweep is fairly slow, but when you know you're generating little or no garbage, it's fast.

Mark-and-sweep follows the same logic of starting from the stack and static storage, and tracing through all the references to find live objects. However, each time it finds a live object, that object is marked by setting a flag in it—the object isn't collected yet. Only when the marking process is finished does the sweep occur. During the sweep, the dead objects are released. However, no copying happens, so if the collector chooses to compact a fragmented heap, it does so by shuffling objects around.

"Stop-and-copy" refers to the idea that this type of garbage collection is *not* done in the background; instead, the program is stopped while the garbage collection occurs. In the Oracle literature you'll find many references to garbage collection as a low-priority background process, but garbage collection was not implemented that way in earlier versions of the JVM. Instead, the garbage collector stopped the program when memory got low. Mark-and-sweep also requires that the program be stopped.

As previously mentioned, in the JVM described here memory is allocated in big blocks. If you allocate an especially large object, it gets its own block. Strict stop-and-copy requires copying every live object from the source heap to a new heap before you can free the old one, which translates to lots of memory. With blocks, the garbage collection can typically copy objects to dead blocks as it collects. Each block has a *generation count* to keep track of whether it's alive. In the normal case, only the blocks created since the last garbage collection are compacted; all other blocks get their generation count bumped if they are referenced from somewhere. This handles the normal case of lots of short-lived temporary objects. Periodically, a full sweep is

made—large objects are still not copied (they just get their generation count bumped), and blocks containing small objects are copied and compacted. The JVM monitors the efficiency of garbage collection and if it becomes a waste of time because all objects are long-lived, it switches to mark-and-sweep. Similarly, the JVM keeps track of how successful mark-and-sweep is, and if the heap starts to become fragmented, it switches back to stop-and-copy. This is where the "adaptive" part comes in, so you end up with a mouthful: "Adaptive generational stop-and-copy mark-and-sweep."

There are a number of additional speedups possible in a JVM. An especially important one involves the operation of the loader and what is called a *just-in-time* (JIT) compiler. A JIT compiler partially or fully converts a program into native machine code so it doesn't need interpretation by the JVM and thus runs much faster. When a class must be loaded (typically, the first time you create an object of that class), the `.class` file is located, and the bytecodes for that class are brought into memory. You could simply JIT compile all the loaded code, but this has two drawbacks: It takes a little more time, which, compounded throughout the life of the program, can add up; and it increases the size of the executable (bytecodes are significantly more compact than expanded JIT code), and this might cause paging, which definitely slows down a program. An alternative approach is *lazy evaluation*, which means the code is not JIT compiled until necessary. Thus, code that never gets executed might never be JIT compiled. The Java HotSpot technologies in recent JDKs take a similar approach by increasingly optimizing a piece of code each time it is executed, so the more the code is executed, the faster it gets.

Member Initialization

Java goes out of its way to guarantee that variables are properly initialized before they are used. In the case of a method's local variables, this guarantee comes in the form of a compile-time error. So if you say:

```
void f() {
  int i;
  i++;
}
```

you'll get an error message that says that `i` might not be initialized. The compiler could give `i` a default value, but an uninitialized local variable is probably a

programmer error, and a default value would cover that up. Forcing the programmer to provide an initialization value is more likely to catch a bug.

If a primitive is a field in a class, however, things are a bit different. As you saw in the Objects Everywhere chapter, each primitive field of a class is guaranteed to get an initial value. Here's a program that verifies this, and shows the values:

```
// housekeeping/InitialValues.java
// Shows default initial values

public class InitialValues {
  boolean t;
  char c;
  byte b;
  short s;
  int i;
  long l;
  float f;
  double d;
  InitialValues reference;
  void printInitialValues() {
    System.out.println("Data type   Initial value");
    System.out.println("boolean     " + t);
    System.out.println("char        [" + c + "]");
    System.out.println("byte        " + b);
    System.out.println("short       " + s);
    System.out.println("int         " + i);
    System.out.println("long        " + l);
    System.out.println("float       " + f);
    System.out.println("double      " + d);
    System.out.println("reference   " + reference);
  }
  public static void main(String[] args) {
    new InitialValues().printInitialValues();
  }
}
/* Output:
Data type   Initial value
boolean     false
char        [NUL]
byte        0
short       0
```

```
int         0
long        0
float       0.0
double      0.0
reference   null
*/
```

Even though the values are not specified, they automatically get initialized (the `char` value is a zero, which my output verification system translates to `NUL`). So at least there's no threat of working with uninitialized variables.

When you define an object reference inside a class without initializing it to a new object, that reference is given a special value of `null`.

Specifying Initialization

How do you give a variable an initial value? One direct way to do this is to assign the value when you define the variable in the class. Here the field definitions in class `InitialValues` are changed to provide initial values:

```
// housekeeping/InitialValues2.java
// Providing explicit initial values

public class InitialValues2 {
  boolean bool = true;
  char ch = 'x';
  byte b = 47;
  short s = 0xff;
  int i = 999;
  long lng = 1;
  float f = 3.14f;
  double d = 3.14159;
}
```

You can also initialize non-primitive objects in this same way. If `Depth` is a class, you can create a variable and initialize it like so:

```
// housekeeping/Measurement.java
class Depth {}

public class Measurement {
  Depth d = new Depth();
  // ...
}
```

If you haven't given `d` an initial value and you try to use it anyway, you'll get a runtime error called an *exception* (covered in the Exceptions chapter).

You can call a method to provide an initialization value:

```
// housekeeping/MethodInit.java
public class MethodInit {
  int i = f();
  int f() { return 11; }
}
```

This method can have arguments, but those arguments cannot be other class members that haven't been initialized yet. Thus, you can do this:

```
// housekeeping/MethodInit2.java
public class MethodInit2 {
  int i = f();
  int j = g(i);
  int f() { return 11; }
  int g(int n) { return n * 10; }
}
```

But you cannot do this:

```
// housekeeping/MethodInit3.java
public class MethodInit3 {
  //- int j = g(i); // Illegal forward reference
  int i = f();
  int f() { return 11; }
  int g(int n) { return n * 10; }
}
```

The compiler appropriately complains about forward referencing because it is about the order of initialization and not the way the program is compiled.

This approach to initialization is simple and straightforward. It has the limitation that *every* object of type `InitialValues` will get these same initialization values. Sometimes this is exactly what you need, but at other times you need more flexibility.

Constructor Initialization

The constructor performs initialization, and this gives you greater flexibility in your programming because you can call methods at run time to determine initial values. However, you aren't precluding automatic initialization, which happens before the constructor is entered. So, for example, if you say:

```
// housekeeping/Counter.java
public class Counter {
  int i;
  Counter() { i = 7; }
  // ...
}
```

then `i` will first be initialized to `0`, then to `7`. This is true with all the primitive types and with object references, including those given explicit initialization at the point of definition. For this reason, the compiler doesn't try to force you to initialize elements in the constructor at any particular place, or before they are used—initialization is already guaranteed.

Order of Initialization

The order variables are defined within a class determines the order of initialization. The variable definitions can be scattered throughout and in between method definitions, but the variables are initialized before any methods can be called—even the constructor. For example:

```
// housekeeping/OrderOfInitialization.java
// Demonstrates initialization order

// When the constructor is called to create a
// Window object, you'll see a message:
class Window {
  Window(int marker) {
    System.out.println("Window(" + marker + ")");
  }
}

class House {
  Window w1 = new Window(1); // Before constructor
  House() {
    // Show that we're in the constructor:
    System.out.println("House()");
    w3 = new Window(33); // Reinitialize w3
  }
  Window w2 = new Window(2); // After constructor
  void f() { System.out.println("f()"); }
  Window w3 = new Window(3); // At end
}

public class OrderOfInitialization {
  public static void main(String[] args) {
    House h = new House();
    h.f(); // Shows that construction is done
  }
}
/* Output:
Window(1)
Window(2)
Window(3)
House()
Window(33)
f()
*/
```

In House, the definitions of the Window objects are intentionally scattered about to prove they'll all get initialized before the constructor is entered or anything else can happen. In addition, w3 is reinitialized inside the constructor.

The output shows that the `w3` reference gets initialized twice: once before and once during the constructor call (The first object is dropped, so it can be garbage collected later). This might not seem efficient at first, but it guarantees proper initialization. Consider what would happen if an overloaded constructor were defined that did *not* initialize `w3` and there wasn't a "default" initialization for `w3` in its definition.

`static` Data Initialization

There's only a single piece of storage for a `static`, regardless of how many objects are created. You can't apply the `static` keyword to local variables, so it only applies to fields. If a field is a `static` primitive and you don't initialize it, it gets the standard initial value for its type. If it's a reference to an object, the default initialization value is `null`.

To place initialization at the point of definition, it looks the same as for non-`static`s.

This shows *when* `static` storage gets initialized:

```
// housekeeping/StaticInitialization.java
// Specifying initial values in a class definition

class Bowl {
  Bowl(int marker) {
    System.out.println("Bowl(" + marker + ")");
  }
  void f1(int marker) {
    System.out.println("f1(" + marker + ")");
  }
}

class Table {
  static Bowl bowl1 = new Bowl(1);
  Table() {
    System.out.println("Table()");
    bowl2.f1(1);
  }
  void f2(int marker) {
    System.out.println("f2(" + marker + ")");
  }
  static Bowl bowl2 = new Bowl(2);
```

```
}

class Cupboard {
  Bowl bowl3 = new Bowl(3);
  static Bowl bowl4 = new Bowl(4);
  Cupboard() {
    System.out.println("Cupboard()");
    bowl4.f1(2);
  }
  void f3(int marker) {
    System.out.println("f3(" + marker + ")");
  }
  static Bowl bowl5 = new Bowl(5);
}

public class StaticInitialization {
  public static void main(String[] args) {
    System.out.println("main creating new Cupboard()");
    new Cupboard();
    System.out.println("main creating new Cupboard()");
    new Cupboard();
    table.f2(1);
    cupboard.f3(1);
  }
  static Table table = new Table();
  static Cupboard cupboard = new Cupboard();
}
/* Output:
Bowl(1)
Bowl(2)
Table()
f1(1)
Bowl(4)
Bowl(5)
Bowl(3)
Cupboard()
f1(2)
main creating new Cupboard()
Bowl(3)
Cupboard()
f1(2)
main creating new Cupboard()
```

```
Bowl(3)
Cupboard()
f1(2)
f2(1)
f3(1)
*/
```

`Bowl` shows the creation of a class, and `Table` and `Cupboard` have `static` members of `Bowl` scattered through their class definitions. Note that `Cupboard` creates a non-`static` `Bowl bowl3` prior to the `static` definitions.

The output shows that `static` initialization occurs only if it's necessary. If you don't create a `Table` object and you never refer to `Table.bowl1` or `Table.bowl2`, the `static Bowl bowl1` and `bowl2` will never be created. They are initialized only when you create the *first* `Table` object (or the first `static` access occurs). After that, the `static` objects are not reinitialized.

The order of initialization is `static`s first, if they haven't already been initialized by a previous object creation, then the non-`static` objects. The evidence is in the output. To execute `main()` (a `static` method), the `StaticInitialization` class must be loaded, and its `static` fields `table` and `cupboard` are then initialized, which causes *those* classes to be loaded, and because they both contain `static Bowl` objects, `Bowl` is then loaded. Thus, all the classes in this particular program get loaded before `main()` starts. This is usually not the case, because in typical programs you won't have everything linked together by `static`s as you do in this example.

To summarize the process of creating an object, consider a class called `Dog`:

1. Even though it doesn't explicitly use the `static` keyword, the constructor is actually a `static` method. So the first time you create an object of type `Dog`, *or* the first time you access a `static` method or `static` field of class `Dog`, the Java interpreter must locate `Dog.class`, which it does by searching through the classpath.
2. As `Dog.class` is loaded (creating a `Class` object, which you'll learn about later), all of its `static` initializers are run. Thus, `static` initialization takes place only once, as the `Class` object is loaded for the first time.
3. When you create a `new Dog()`, the construction process for a `Dog` object first allocates enough storage for a `Dog` object on the heap.

4. This storage is wiped to zero, automatically setting all the primitives in that `Dog` object to their default values (zero for numbers and the equivalent for `boolean` and `char`) and the references to `null`.
5. Any initializations that occur at the point of field definition are executed.
6. Constructors are executed. As you shall see in the Reuse chapter, this might actually involve a fair amount of activity, especially when inheritance is involved.

Explicit `static` Initialization

You can group other `static` initializations inside a special "`static` clause" (sometimes called a *static block*) in a class. It looks like this:

```
// housekeeping/Spoon.java
public class Spoon {
  static int i;
  static {
    i = 47;
  }
}
```

It looks a little like a method, but it's just the `static` keyword followed by a block of code. This code, like other `static` initializations, is executed only once: the first time you make an object of that class *or* the first time you access a `static` member of that class (even if you never make an object of that class). For example:

```
// housekeeping/ExplicitStatic.java
// Explicit static initialization with "static" clause

class Cup {
  Cup(int marker) {
    System.out.println("Cup(" + marker + ")");
  }
  void f(int marker) {
    System.out.println("f(" + marker + ")");
  }
}

class Cups {
```

```
  static Cup cup1;
  static Cup cup2;
  static {
    cup1 = new Cup(1);
    cup2 = new Cup(2);
  }
  Cups() {
    System.out.println("Cups()");
  }
}

public class ExplicitStatic {
  public static void main(String[] args) {
    System.out.println("Inside main()");
    Cups.cup1.f(99);                  // [1]
  }
  // static Cups cups1 = new Cups();  // [2]
  // static Cups cups2 = new Cups();  // [2]
}
/* Output:
Inside main()
Cup(1)
Cup(2)
f(99)
*/
```

The `static` initializers for `Cups` run when either the access of the `static` object `cup1` occurs on **[1]**, or if **[1]** is commented out and the lines marked **[2]** are uncommented. If both **[1]** and **[2]** are commented out, the `static` initialization for `Cups` never occurs. Also, it doesn't matter if one or both of the lines marked **[2]** are uncommented; the static initialization only occurs once.

Non-`static` Instance Initialization

Java provides a similar syntax, called *instance initialization*, for initializing non-`static` variables for each object. Here's an example:

```
// housekeeping/Mugs.java
// Instance initialization

class Mug {
  Mug(int marker) {
    System.out.println("Mug(" + marker + ")");
  }
}

public class Mugs {
  Mug mug1;
  Mug mug2;
  {                                          // [1]
    mug1 = new Mug(1);
    mug2 = new Mug(2);
    System.out.println("mug1 & mug2 initialized");
  }
  Mugs() {
    System.out.println("Mugs()");
  }
  Mugs(int i) {
    System.out.println("Mugs(int)");
  }
  public static void main(String[] args) {
    System.out.println("Inside main()");
    new Mugs();
    System.out.println("new Mugs() completed");
    new Mugs(1);
    System.out.println("new Mugs(1) completed");
  }
}
/* Output:
Inside main()
Mug(1)
Mug(2)
mug1 & mug2 initialized
Mugs()
new Mugs() completed
Mug(1)
Mug(2)
mug1 & mug2 initialized
Mugs(int)
```

```
new Mugs(1) completed
*/
```

- **[1]** The instance initialization clause looks exactly like the static initialization clause except for the missing `static` keyword. This syntax is necessary to support the initialization of *anonymous inner classes* (see the Inner Classes chapter), but you can also guarantee that certain operations occur regardless of which explicit constructor is called.

The output shows that the instance initialization clause is executed before either one of the constructors.

Array Initialization

An array is a sequence of either objects or primitives that are all the same type and are packaged together under one identifier name. Arrays are defined and used with the square-brackets *indexing operator* `[]`. To define an array reference, you follow the type name with empty square brackets:

```
int[] a1;
```

You can also put the square brackets after the identifier to produce exactly the same meaning:

```
int a1[];
```

This conforms to expectations from C and C++ programmers. The former style, however, is probably a more sensible syntax because it says that the type is "an `int` array." That style is used in this book.

The compiler doesn't allow you to tell it how big the array is. This brings us back to that issue of "references." All you have now is a reference to an array (you've allocated enough storage for that reference), and there's been no space allocated for the array object itself. To create storage for the array, you must write an initialization expression. For arrays, initialization can appear anywhere in your code, but you can also use a special kind of initialization expression that must occur when you create the array. This special initialization is a set of values surrounded by curly braces. The storage allocation (the equivalent of using `new`) is taken care of by the compiler in this case. For example:

```
int[] a1 = { 1, 2, 3, 4, 5 };
```

Why would you ever define an array reference without an array?

```
int[] a2;
```

Well, it's possible to assign one array to another in Java, so you can say:

```
a2 = a1;
```

What you're really doing is copying a reference, as demonstrated here:

```
// housekeeping/ArraysOfPrimitives.java

public class ArraysOfPrimitives {
  public static void main(String[] args) {
    int[] a1 = { 1, 2, 3, 4, 5 };
    int[] a2;
    a2 = a1;
    for(int i = 0; i < a2.length; i++)
      a2[i] += 1;
    for(int i = 0; i < a1.length; i++)
      System.out.println("a1[" + i + "] = " + a1[i]);
  }
}
/* Output:
a1[0] = 2
a1[1] = 3
a1[2] = 4
a1[3] = 5
a1[4] = 6
*/
```

`a1` is given an initialization value but `a2` is not; `a2` is assigned later—here, to another array. `a2` and `a1` are then aliased to the same array, so the changes made via `a2` are seen in `a1`.

All arrays have an intrinsic member (whether they're arrays of objects or arrays of primitives) you can query—but not change—to tell you how many elements there are in the array. This member is `length`. Arrays in Java, just like C and C++, start counting from element zero, so the largest element you can index is `length - 1`. If

you go out of bounds, C and C++ quietly accept this and allow you to stomp all over your memory, which is the source of many infamous bugs. However, Java protects you against such problems by causing a runtime error (an *exception*) if you step out of bounds.[68]

Dynamic Array Creation

What if you don't know how many elements you're going to need in your array while you're writing the program? You simply use `new` to create the elements in the array. Here, `new` works even though it's creating an array of primitives (`new` won't create a non-array primitive):

```
// housekeeping/ArrayNew.java
// Creating arrays with new
import java.util.*;

public class ArrayNew {
  public static void main(String[] args) {
    int[] a;
    Random rand = new Random(47);
    a = new int[rand.nextInt(20)];
    System.out.println("length of a = " + a.length);
    System.out.println(Arrays.toString(a));
  }
}
/* Output:
length of a = 18
[0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0]
*/
```

The size of the array is chosen at random by using the `Random.nextInt()` method, which produces a value between zero and that of its argument. Because of the randomness, it's clear that array creation is actually happening at run time. In addition, the output of this program shows that array elements of primitive types

[68] Of course, checking every array access costs time and code and there's no way to turn it off, which means that array accesses might be a source of inefficiency in your program if they occur at a critical juncture. For Internet security and programmer productivity, the Java designers saw that this was a worthwhile trade-off. Although you may be tempted to write code you think might make array accesses more efficient, this is a waste of time because automatic compile-time and runtime optimizations will speed array accesses.

are automatically initialized to "empty" values. (For numerics and `char`, this is zero, and for `boolean`, it's `false`.)

The `Arrays.toString()` method, part of the standard `java.util` library, produces a printable version of a one-dimensional array.

The array can also be defined and initialized in the same statement:

```
int[] a = new int[rand.nextInt(20)];
```

This is the preferred way to do it, if you can.

If you create a non-primitive array, you create an array of references. Consider the wrapper type `Integer`, which is a class and not a primitive:

```
// housekeeping/ArrayClassObj.java
// Creating an array of nonprimitive objects
import java.util.*;

public class ArrayClassObj {
  public static void main(String[] args) {
    Random rand = new Random(47);
    Integer[] a = new Integer[rand.nextInt(20)];
    System.out.println("length of a = " + a.length);
    for(int i = 0; i < a.length; i++)
      a[i] = rand.nextInt(500); // Autoboxing
    System.out.println(Arrays.toString(a));
  }
}
/* Output:
length of a = 18
[55, 193, 361, 461, 429, 368, 200, 22, 207, 288, 128,
51, 89, 309, 278, 498, 361, 20]
*/
```

Here, even after `new` is called to create the array:

```
Integer[] a = new Integer[rand.nextInt(20)];
```

it's only an array of references, and the initialization is not complete until the reference itself is initialized by creating a new `Integer` object (via autoboxing, in this case):

```
a[i] = rand.nextInt(500);
```

If you forget to create the object, however, you'll get an exception at run time when you try to use the empty array location.

It's also possible to initialize arrays of objects by using a curly brace-enclosed list. There are two forms:

```
// housekeeping/ArrayInit.java
// Array initialization
import java.util.*;

public class ArrayInit {
  public static void main(String[] args) {
    Integer[] a = {
      1, 2,
      3, // Autoboxing
    };
    Integer[] b = new Integer[]{
      1, 2,
      3, // Autoboxing
    };
    System.out.println(Arrays.toString(a));
    System.out.println(Arrays.toString(b));
  }
}
/* Output:
[1, 2, 3]
[1, 2, 3]
*/
```

In both cases, the final comma in the list of initializers is optional. (This feature makes for easier maintenance of long lists.)

Although the first form is useful, it's more limited because it can only be used at the point the array is defined. You can use the second and third forms anywhere, even inside a method call. For example, you can create an array of `String` objects to pass to the `main()` of another class, to provide alternate command-line arguments to that `main()`:

```
// housekeeping/DynamicArray.java
// Array initialization

public class DynamicArray {
  public static void main(String[] args) {
    Other.main(new String[]{ "fiddle", "de", "dum" });
  }
}

class Other {
  public static void main(String[] args) {
    for(String s : args)
      System.out.print(s + " ");
  }
}
/* Output:
fiddle de dum
*/
```

The array argument for `Other.main()` is created at the point of the method call, so you can even provide alternate arguments at the time of the call.

Variable Argument Lists

You create and call methods in a way that produces an effect similar to C's *variable argument lists* (known as "varargs" in C). These can include unknown quantities of arguments as well as unknown types. All classes are ultimately inherited from the common root class `Object` (a subject you learn more about as this book progresses), so you can create a method that takes an array of `Object` and call it like this:

```
// housekeeping/VarArgs.java
// Using array syntax to create variable argument lists

class A {}

public class VarArgs {
  static void printArray(Object[] args) {
    for(Object obj : args)
      System.out.print(obj + " ");
    System.out.println();
  }
  public static void main(String[] args) {
    printArray(new Object[]{
      47, (float) 3.14, 11.11});
    printArray(new Object[]{"one", "two", "three" });
    printArray(new Object[]{new A(), new A(), new A()});
  }
}
/* Output:
47 3.14 11.11
one two three
A@19e0bfd A@139a55 A@1db9742
*/
```

`printArray()` takes an array of `Object`, then steps through the array using the *for-in* syntax and prints each one. The standard Java library classes produce sensible output, but the objects of the classes created here print the class name, followed by an `@` sign and hexadecimal digits. Thus, the default behavior (if you don't define a `toString()` method for your class, which is described later in the book) is to print the class name and the address of the object.

You might see pre-Java 5 code written like the above to produce variable argument lists. In Java 5 this long-requested feature was finally added, so ellipses define a variable argument list, as in `printArray()`:

```
// housekeeping/NewVarArgs.java
// Using array syntax to create variable argument lists

public class NewVarArgs {
  static void printArray(Object... args) {
    for(Object obj : args)
      System.out.print(obj + " ");
    System.out.println();
  }
  public static void main(String[] args) {
    // Can take individual elements:
    printArray(47, (float) 3.14, 11.11);
    printArray(47, 3.14F, 11.11);
    printArray("one", "two", "three");
    printArray(new A(), new A(), new A());
    // Or an array:
    printArray((Object[])new Integer[]{ 1, 2, 3, 4 });
    printArray(); // Empty list is OK
  }
}
/* Output:
47 3.14 11.11
47 3.14 11.11
one two three
A@19e0bfd A@139a55 A@1db9742
1 2 3 4
*/
```

With varargs, you no longer explicitly write out the array syntax—the compiler will actually fill it in for you when you specify varargs. You're still getting an array, which is why `printArray()` is able to use *for-in* to iterate through the array. However, it's more than just an automatic conversion from a list of elements to an array. Notice the second-to-last line in the program, where an array of `Integer` (created using autoboxing) is cast to an `Object` array (to remove a compiler warning) and passed to `printArray()`. Clearly, the compiler sees this is already an array and performs no conversion on it. So if you have a group of items you can pass them in as a list, and if you already have an array it will accept that as the variable argument list.

The last line of the program shows it's possible to pass zero arguments to a vararg list. This is helpful when you have optional trailing arguments:

```
// housekeeping/OptionalTrailingArguments.java

public class OptionalTrailingArguments {
  static void f(int required, String... trailing) {
    System.out.print("required: " + required + " ");
    for(String s : trailing)
      System.out.print(s + " ");
    System.out.println();
  }
  public static void main(String[] args) {
    f(1, "one");
    f(2, "two", "three");
    f(0);
  }
}
/* Output:
required: 1 one
required: 2 two three
required: 0
*/
```

This also shows how you can use varargs with a specified type other than `Object`. Here, all the varargs must be `String` objects. It's possible to use any type of argument in varargs, including a primitive type. The following example also shows that the vararg list becomes an array, and if there's nothing in the list it's an array of size zero:

```
// housekeeping/VarargType.java

public class VarargType {
  static void f(Character... args) {
    System.out.print(args.getClass());
    System.out.println(" length " + args.length);
  }
  static void g(int... args) {
    System.out.print(args.getClass());
    System.out.println(" length " + args.length);
  }
  public static void main(String[] args) {
    f('a');
    f();
    g(1);
```

```
    g();
    System.out.println("int[]: " +
      new int[0].getClass());
  }
}
/* Output:
class [Ljava.lang.Character; length 1
class [Ljava.lang.Character; length 0
class [I length 1
class [I length 0
int[]: class [I
*/
```

The `getClass()` method is part of `Object`, and is explored fully in the Reflection chapter. It produces the class of an object, and when you print this class, you see an encoded `String` representing the class type. The leading `[` indicates this is an array of the type that follows. The `I` is for a primitive `int`; to double-check, I created an array of `int` in the last line and printed its type. This verifies that using varargs does not depend on autoboxing, but it actually uses the primitive types.

Varargs work harmoniously with autoboxing:

```
// housekeeping/AutoboxingVarargs.java

public class AutoboxingVarargs {
  public static void f(Integer... args) {
    for(Integer i : args)
      System.out.print(i + " ");
    System.out.println();
  }
  public static void main(String[] args) {
    f(1, 2);
    f(4, 5, 6, 7, 8, 9);
    f(10, 11, 12);
  }
}
/* Output:
1 2
4 5 6 7 8 9
10 11 12
*/
```

Notice you can mix the types together in a single argument list, and autoboxing selectively promotes the `int` arguments to `Integer`.

Varargs complicate the process of overloading, although it seems safe enough at first:

```
// housekeeping/OverloadingVarargs.java

public class OverloadingVarargs {
  static void f(Character... args) {
    System.out.print("first");
    for(Character c : args)
      System.out.print(" " + c);
    System.out.println();
  }
  static void f(Integer... args) {
    System.out.print("second");
    for(Integer i : args)
      System.out.print(" " + i);
    System.out.println();
  }
  static void f(Long... args) {
    System.out.println("third");
  }
  public static void main(String[] args) {
    f('a', 'b', 'c');
    f(1);
    f(2, 1);
    f(0);
    f(0L);
    //- f(); // Won't compile -- ambiguous
  }
}
/* Output:
first a b c
second 1
second 2 1
second 0
third
*/
```

In each case, the compiler uses autoboxing to match the overloaded method, and calls the most specifically matching method.

But when you call `f()` without arguments, it has no way of knowing which one to call. Although this error is understandable, it will probably surprise the client programmer.

You might try solving the problem by adding a non-vararg argument to one of the methods:

```
// housekeeping/OverloadingVarargs2.java
// {WillNotCompile}

public class OverloadingVarargs2 {
  static void f(float i, Character... args) {
    System.out.println("first");
  }
  static void f(Character... args) {
    System.out.print("second");
  }
  public static void main(String[] args) {
    f(1, 'a');
    f('a', 'b');
  }
}
```

The `{WillNotCompile}` comment tag excludes the file from this book's Gradle build.

If you compile it by hand you'll see the error message:

```
OverloadingVarargs2.java:14: error: reference to f is ambiguous
 f('a', 'b');
 \^
 both method f(float,Character...) in OverloadingVarargs2 and method
f(Character...) in OverloadingVarargs2 match
1 error
```

If you give *both* methods a non-vararg argument, it works:

```
// housekeeping/OverloadingVarargs3.java

public class OverloadingVarargs3 {
  static void f(float i, Character... args) {
    System.out.println("first");
  }
  static void f(char c, Character... args) {
    System.out.println("second");
  }
  public static void main(String[] args) {
    f(1, 'a');
    f('a', 'b');
  }
}
/* Output:
first
second
*/
```

As a rule of thumb, only use a variable argument list on one version of an overloaded method. Or consider not doing it at all.

Enumerated Types

A seemingly small addition in Java 5 is the `enum` keyword, which makes your life much easier when you group together and use a set of *enumerated types*. In the past you created a set of constant integral values, but these do not naturally restrict themselves to your set and thus are riskier and more difficult to use. Enumerated types are a common enough need that C, C++, and a number of other languages have always had them. Before Java 5, programmers were forced to know a lot and be quite careful when they wanted to properly produce the `enum` effect. Now Java has `enum`, too, and it's much more full-featured than what you find in C/C++. Here's a simple example:

```
// housekeeping/Spiciness.java

public enum Spiciness {
  NOT, MILD, MEDIUM, HOT, FLAMING
}
```

This creates an enumerated type called `Spiciness` with five named values. Because the instances of enumerated types are constants, they are in all capital letters by convention (if there are multiple words in a name, they are separated by underscores).

To use an `enum`, you create a reference of that type and assign it to an instance:

```
// housekeeping/SimpleEnumUse.java

public class SimpleEnumUse {
  public static void main(String[] args) {
    Spiciness howHot = Spiciness.MEDIUM;
    System.out.println(howHot);
  }
}
/* Output:
MEDIUM
*/
```

The compiler automatically adds useful features when you create an `enum`. For example, it creates a `toString()` to easily display the name of an `enum` instance, which is how the print statement above produced its output. The compiler also creates an `ordinal()` method to indicate the declaration order of a particular `enum` constant, and a `static values()` method that produces an array of values of the `enum` constants in the order they were declared:

```
// housekeeping/EnumOrder.java

public class EnumOrder {
  public static void main(String[] args) {
    for(Spiciness s : Spiciness.values())
      System.out.println(
        s + ", ordinal " + s.ordinal());
  }
}
/* Output:
NOT, ordinal 0
MILD, ordinal 1
MEDIUM, ordinal 2
HOT, ordinal 3
FLAMING, ordinal 4
*/
```

Although `enum`s appear to be a new data type, the keyword only produces some compiler behavior while generating a class for the `enum`, so in many ways you can treat an `enum` as if it were any other class. In fact, `enum`s *are* classes and have their own methods.

An especially nice feature is the way that `enum`s can be used inside `switch` statements:

```
// housekeeping/Burrito.java

public class Burrito {
  Spiciness degree;
  public Burrito(Spiciness degree) {
    this.degree = degree;
  }
  public void describe() {
    System.out.print("This burrito is ");
    switch(degree) {
      case NOT:    System.out.println(
                     "not spicy at all.");
                   break;
      case MILD:
      case MEDIUM: System.out.println("a little hot.");
                   break;
      case HOT:
```

```
      case FLAMING:
      default:     System.out.println("maybe too hot.");
    }
  }
  public static void main(String[] args) {
    Burrito
      plain = new Burrito(Spiciness.NOT),
      greenChile = new Burrito(Spiciness.MEDIUM),
      jalapeno = new Burrito(Spiciness.HOT);
    plain.describe();
    greenChile.describe();
    jalapeno.describe();
  }
}
/* Output:
This burrito is not spicy at all.
This burrito is a little hot.
This burrito is maybe too hot.
*/
```

A `switch` is intended to select from a limited set of possibilities so it's an ideal match for an `enum`. Notice how `enum` names can produce a much clearer expression of intent.

In general you can use an `enum` as if it were another way to create a data type, then just put the results to work. That's the point, so you don't have to think too hard about them. Before the introduction of `enum`, you went to a lot of effort to make an equivalent enumerated type that was safe to use.

This is enough for you to understand and use basic `enum`s, but we'll look more deeply at them later in the book—they have their own chapter: Enumerations.

Summary

This seemingly elaborate mechanism for initialization, the constructor, should give you a strong hint about the critical importance placed on initialization in the language. As Bjarne Stroustrup, the inventor of C++, was designing that language, one of the first observations he made about productivity in C was that improper initialization of variables causes a significant portion of programming problems. These kinds of bugs are hard to find, and similar issues apply to improper cleanup.

Because constructors allow you to *guarantee* proper initialization and cleanup (the compiler will not allow an object to be created without the proper constructor calls), you get complete control and safety.

In C++, destruction is quite important because objects created with `new` must be explicitly destroyed. In Java, the garbage collector automatically releases the memory for all objects, so the equivalent cleanup method in Java isn't necessary much of the time (but when it is, you must do it yourself). In cases where you don't need destructor-like behavior, Java's garbage collector greatly simplifies programming and adds much-needed safety in managing memory. Some garbage collectors can even clean up other resources like graphics and file handles. However, the garbage collector does add a runtime cost, the expense of which is difficult to put into perspective because of the historical slowness of Java interpreters. Although Java has had significant performance increases over time, the speed problem has taken its toll on the adoption of the language for certain types of programming problems.

Because of the guarantee that all objects are constructed, there's actually more to the constructor than what is shown here. In particular, when you create new classes using either *composition* or *inheritance*, the guarantee of construction also holds, and some additional syntax is necessary to support this. You'll learn about composition, inheritance, and how they affect constructors in future chapters.

Implementation Hiding

> *Access control* (or *implementation hiding*) is about "not getting it right the first time."

All good writers—including those who write software—know that a piece of work isn't good until it's been rewritten, often many times. If you leave a piece of code in a drawer for a while and come back to it, you might see a much better way to do it. This is one of the prime motivations for *refactoring*, which rewrites working code to make it more readable, understandable, and thus maintainable.[69]

There is a tension, however, in this desire to change and improve your code. There are often consumers (*client programmers*) who rely on some aspect of your code staying the same. So you want to change it; they want it to stay the same. Thus a primary consideration in object-oriented design is to "separate the things that change from the things that stay the same."

This is particularly important for libraries. Consumers of that library must rely on the part they use, and know they won't have to rewrite code if a new version of the library comes out. On the flip side, the library creator must have the freedom to make modifications and improvements with the certainty that the client code won't be affected by those changes.

This can be achieved through convention. For example, the library programmer must agree not to remove existing methods when modifying a class in the library. That would break the client programmer's code. The reverse situation is more complex. In the case of a field, how can the library creator know which fields were accessed by client programmers? This is also true with methods used only to implement a class, but not meant for direct use by client programmers. What if the library creator wants to rip out an old implementation and put in a new one? Changing any of

[69] See *Refactoring: Improving the Design of Existing Code*, by Martin Fowler, et al. (Addison-Wesley, 1999). Occasionally someone will argue against refactoring, suggesting that code which works is perfectly good and it's a waste of time to refactor it. The problem with this way of thinking is that the lion's share of a project's time and money is not in the initial writing of the code, but in maintaining it. Making code easier to understand translates into very significant dollars.

those members might break a client programmer's code. Thus the library creator is in a strait jacket and can't change anything.

To solve this problem, Java provides *access specifiers* to allow the library creator to say what is available to the client programmer and what is not. The levels of access control span from "most access" to "least access": `public`, `protected`, *package access* (which has no keyword), and `private`. From the previous paragraph you might think, as a library designer, you'll keep everything as "private" as possible, and expose only the methods you want the client programmer to use. This is generally what you'll do, even though it's often counterintuitive for people who program in other languages (especially C) and who are used to accessing everything without restriction.

The concept of a library of components and the control over who can access the components of that library is not complete, however. There's still the question of how the components are bundled together into a cohesive library unit. This is controlled with the `package` keyword in Java, and the access specifiers are affected by whether a class is in the same package or in a separate package. So to begin this chapter, you'll learn how library components are placed into packages. Then you can understand the complete meaning of the access specifiers.

`package`: the Library Unit

A package contains a group of classes, organized together under a single *namespace*.

For example, there's a utility library that's part of the standard Java distribution, organized under the namespace `java.util`. One of the classes in `java.util` is called `ArrayList`. One way to use an `ArrayList` is to specify the full name `java.util.ArrayList`.

```
// hiding/FullQualification.java

public class FullQualification {
  public static void main(String[] args) {
    java.util.ArrayList list =
      new java.util.ArrayList();
  }
}
```

This rapidly becomes tedious, so you can use the `import` keyword instead. To import a single class, you name that class in the `import` statement:

```
// hiding/SingleImport.java
import java.util.ArrayList;

public class SingleImport {
  public static void main(String[] args) {
    ArrayList list = new ArrayList();
  }
}
```

Now you can use `ArrayList` with no qualification. However, none of the other classes in `java.util` are available. To import everything, you use the `*` as you've been seeing in the rest of the examples in this book:

```
import java.util.*;
```

The reason for all this importing is to provide a mechanism to manage namespaces. The names of all your class members are insulated from each other. A method `f()` inside a class `A` will not clash with an `f()` that has the same signature in class `B`. But what about the class names? Suppose you create a `Stack` class that is installed on a machine that already has a `Stack` class that's written by someone else? This potential clashing of names is why we need complete control over namespaces in Java. To achieve this, we create a unique identifier combination for each class.

Most of the examples thus far have existed in a single file and are designed for local use, so they haven't bothered with package names. However, an example without a package name is still in a package: the "unnamed" or *default package*. This is certainly an option, and for simplicity's sake this approach is used whenever possible throughout the rest of this book. However, if you're planning to create libraries or

programs friendly to other Java programs on the same machine, you must think about preventing class name clashes.

A Java source-code file is called a *compilation unit* (sometimes a *translation unit*). Each compilation unit must have a file name ending in `.java`. Inside the compilation unit there *can* be a `public` class that must have the same name as the file (including capitalization, but excluding the `.java` file name extension). There can be only *one* `public` class in each compilation unit; otherwise, the compiler will complain. If there are additional classes in that compilation unit, they are hidden from the world outside that package because they're *not* `public`, and they comprise "support" classes for the main `public` class.

Code Organization

When you compile a `.java` file, you get an output file *for each class* in the `.java` file. Each output file has the name of its corresponding class in the `.java` file, but with an extension of `.class`. Thus you can end up with quite a few `.class` files from a small number of `.java` files. If you've programmed with a compiled language, you might be used to the compiler spitting out an intermediate form (usually an "obj" file) that is then packaged together with others of its kind using a linker (to create an executable file) or a librarian (to create a library). That's not how Java works. A working program is a bunch of `.class` files, which can be packaged and compressed into a *Java ARchive* (JAR) file (using the `jar` archiver). The Java interpreter is responsible for finding, loading, and interpreting these files.

A library is a group of these class files. Each source file usually has a `public` class and any number of non-`public` classes, so there's one `public` component for each source file. To say that all these components belong together, use the `package` keyword.

If you use a `package` statement, it *must* appear as the first non-comment in the file. When you say:

```
package hiding;
```

you're stating this compilation unit is part of a library named `hiding`. Put another way, you're saying that the `public` class name within this compilation unit is under the umbrella of the name `hiding`, and anyone who wants to use that name must either fully specify the name or use the `import` keyword in combination with `hiding`,

using the choices given previously. (Note that the convention for Java package names is to use all lowercase letters, even for intermediate words.)

For example, suppose the name of a file is `MyClass.java`. This means there can only be one `public` class in that file, and the name of that class must be `MyClass` (including the capitalization):

```
// hiding/mypackage/MyClass.java
package hiding.mypackage;

public class MyClass {
  // ...
}
```

Now, if someone wants to use `MyClass` or, for that matter, any of the other `public` classes in `hiding.mypackage`, they must use the `import` keyword to make the name or names in `hiding.mypackage` available. The alternative is to give the fully qualified name:

```
// hiding/QualifiedMyClass.java

public class QualifiedMyClass {
  public static void main(String[] args) {
    hiding.mypackage.MyClass m =
      new hiding.mypackage.MyClass();
  }
}
```

The `import` keyword makes it cleaner:

```
// hiding/ImportedMyClass.java
import hiding.mypackage.*;

public class ImportedMyClass {
  public static void main(String[] args) {
    MyClass m = new MyClass();
  }
}
```

The `package` and `import` keywords divide up the single global namespace so names don't clash.

Creating Unique Package Names

You might observe that, because a package never really gets "packaged" into a single file, a package can be made up of many `.class` files, and things could get a bit cluttered. To prevent this, a logical thing to do is to place all the `.class` files for a particular package into a single directory; that is, use the hierarchical file structure of the operating system to your advantage. This is one way that Java references the problem of clutter; you'll see the other way later when the `jar` utility is introduced.

Collecting the package files into a single subdirectory solves two other problems: creating unique package names, and finding those classes that might be buried in a directory structure someplace. This is accomplished by encoding the path of the location of the `.class` file into the name of the `package`. By convention, the first part of the `package` name is the reversed Internet domain name of the creator of the class. Because Internet domain names are unique, *if* you follow this convention, your `package` name is also unique and you won't have a name clash. If you don't have your own domain name, you must fabricate an unlikely combination (such as your first and last name) to create unique package names. If you've decided to start publishing Java code, it's worth the relatively small effort to get a domain name.

The second part of this trick is resolving the `package` name into a directory on your machine, so when the Java interpreter must load a `.class` file, it can locate the directory where that `.class` file resides. First, it finds the environment variable CLASSPATH[70] (set via the operating system, and sometimes by the installation program that installs Java or a Java-based tool on your machine). CLASSPATH contains one or more directories that are roots in a search for `.class` files. Starting at that root, the interpreter takes the package name and replaces each dot with a slash to generate a path name off of the CLASSPATH root (so `package foo.bar.baz` becomes `foo\bar\baz` or `foo/bar/baz` or possibly something else, depending on your operating system). This is then concatenated with the various entries in the CLASSPATH. That's where it looks for the `.class` file with the name corresponding to the class you're trying to create. (It also searches some standard directories relative to where the Java interpreter resides.)

To understand this, consider my domain name, `MindviewInc.com`. By reversing this and making it all lowercase, `com.mindviewinc` establishes my unique global name

[70] When referring to the environment variable, capital letters are used (e.g. CLASSPATH).

for my classes. (The com, edu, org, etc., extensions were formerly capitalized in Java packages, but this was changed in Java 2 so the entire package name is lowercase.) I subdivide this by deciding to create a library named simple, yielding:

```
package com.mindviewinc.simple;
```

This package name can be used as an umbrella namespace for the following two files:

```
// com/mindviewinc/simple/Vector.java
// Creating a package
package com.mindviewinc.simple;

public class Vector {
  public Vector() {
    System.out.println("com.mindviewinc.simple.Vector");
  }
}
```

As mentioned before, the package statement must be the first non-comment code in the file. The second file looks much the same:

```
// com/mindviewinc/simple/List.java
// Creating a package
package com.mindviewinc.simple;

public class List {
  public List() {
    System.out.println("com.mindviewinc.simple.List");
  }
}
```

Both of these files are placed in the following subdirectory on my machine:

```
C:\DOC\Java\com\mindviewinc\simple
```

(The first comment line in every file in this book establishes the directory location of that file in the source-code tree—this is used by the automatic code-extraction tool for this book.)

If you walk back through this path, you see the package name com.mindviewinc.simple, but what about the first portion of the path? That's taken care of by the CLASSPATH environment variable. On my machine, part of the CLASSPATH looks like this:

```
CLASSPATH=.;D:\JAVA\LIB;C:\DOC\Java
```

The CLASSPATH can contain many alternative search paths.

There's a variation when using JAR files, however. You must put the actual name of the JAR file in the classpath, not just the path where it's located. So for a JAR named `grape.jar` your classpath would include:

```
CLASSPATH=.;D:\JAVA\LIB;C:\flavors\grape.jar
```

Once the classpath is set up properly, the following file can be placed in any directory:

```
// hiding/LibTest.java
// Uses the library
import com.mindviewinc.simple.*;

public class LibTest {
  public static void main(String[] args) {
    Vector v = new Vector();
    List l = new List();
  }
}
/* Output:
com.mindviewinc.simple.Vector
com.mindviewinc.simple.List
*/
```

When the compiler encounters the `import` statement for the `simple` library, it begins searching at the directories specified by CLASSPATH, looking for subdirectory `com/mindviewinc/simple`, then seeking the compiled files of the appropriate names (`Vector.class` for `Vector`, and `List.class` for `List`). Note that both the classes and the desired methods in `Vector` and `List` must be `public`.

Setting the CLASSPATH is such a trial for beginning Java users (it was for me, when I started) that the JDK for later versions of Java got a bit smarter. You'll find that when you install it, even if you don't set the CLASSPATH, you can compile and run basic Java programs. However, to compile and run the individual examples for this book (available at https://github.com/BruceEckel/OnJava8-examples[71]), you must add the base directory of the book's unpacked code tree to your CLASSPATH (the `gradlew` command manages its own CLASSPATH, so you only need to set the CLASSPATH if you want to use `javac` and `java` directly, without Gradle).

[71] https://github.com/BruceEckel/Onjava8-examples

Collisions

What happens if two libraries are imported via * and they include the same names? For example, suppose a program does this:

```
import com.mindviewinc.simple.*;
import java.util.*;
```

`java.util.*` also contains a `Vector` class so this causes a potential collision. However, as long as you don't write the code that actually causes the collision, everything is OK—this is good, because otherwise you might do a lot of typing to prevent collisions that never happen.

The collision *does* occur if you now try to make a `Vector`:

```
Vector v = new Vector();
```

Which `Vector` class does this refer to? The compiler can't know, and the reader can't know either. So the compiler complains and forces you to be explicit. For the standard Java `Vector`, you say:

```
java.util.Vector v = new java.util.Vector();
```

This, along with the CLASSPATH, completely specifies the location of that `Vector` so there's no need for the `import java.util.*` statement unless I use something else from `java.util`.

Alternatively, you can use the single-class import form to prevent clashes—as long as you don't use both colliding names in the same program (in which case you must fall back to fully specifying the names).

A Custom Tool Library

With this knowledge, you can now create your own libraries of tools to reduce or eliminate duplicate code.

Ordinarily, I would package such a utility using my reversed domain name, in something like `com.mindviewinc.util`, but to simplify and reduce some visual noise, I'll reduce this book's utility package name to just `onjava`.

For example, here are the `range()` methods introduced in the Control Flow chapter. These allow *for-in* syntax for simple integer sequences:

```
// onjava/Range.java
// Create arrays initialized with integer values.
package onjava;

public class Range {
  // Produce sequence [start..end) incrementing by step
  public static
  int[] range(int start, int end, int step) {
    if (step == 0)
      throw new
        IllegalArgumentException("Step cannot be zero");
    int sz = Math.max(0, step >= 0 ?
        (end + step - 1 - start) / step
      : (end + step + 1 - start) / step);
    int[] result = new int[sz];
    for(int i = 0; i < sz; i++)
      result[i] = start + (i * step);
    return result;
  }  // Produce a sequence [start..end)
  public static int[] range(int start, int end) {
    return range(start, end, 1);
  }
  // Produce a sequence [0..n)
  public static int[] range(int n) {
    return range(0, n);
  }
}
```

The location of `Range.java` must be in a directory that starts at one of the CLASSPATH locations, then continues into `onjava`. After compiling, the methods can be used anywhere in your system using an `import static` statement. Here are a few basic tests to verify that it's working correctly:

```
// onjava/TestRange.java
// Basic test of Range.java
import static onjava.Range.*;
import java.util.Arrays;

public class TestRange {
  private static void show(int[] rng) {
    System.out.println(Arrays.toString(rng));
  }
  public static void main(String[] args) {
    show(range(10, 21, 3));
    show(range(21, 10, -3));
    show(range(-5, 5, -3));
    show(range(-5, 5, 3));
    show(range(10, 21));
    show(range(10));
  }
}
/* Output:
[10, 13, 16, 19]
[21, 18, 15, 12]
[]
[-5, -2, 1, 4]
[10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20]
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
*/
```

From now on, whenever you come up with a useful new utility, you can add it to your own library. You'll see more components added to the onjava library throughout the book.

Using Imports to Change Behavior

A missing feature in Java is C's *conditional compilation*, where you change a switch and get different behavior without changing any other code. The reason such a feature was left out of Java is probably because it is most often used in C to solve cross-platform issues: Different portions of the code are compiled depending on the target platform. Java is intended to be automatically cross-platform so such a feature should not be necessary.

However, there are other valuable uses for conditional compilation. A very common use is for debugging code. The debugging features are enabled during development and disabled in the shipping product. You can accomplish this by changing the `package` that's imported to change the code used in your program from the debug version to the production version. This technique can be used for any kind of conditional code.

Package Caveat

When you create a package, you implicitly specify a directory structure when you give the package a name. The package *must* live in the directory indicated by its name, and that directory must be searchable starting from the CLASSPATH. Experimenting with the `package` keyword can be a bit frustrating at first, because unless you adhere to the package-name to directory-path rule, you'll get many mysterious runtime messages about not being able to find a particular class, even if that class is sitting there in the same directory. If you get a message like this, try commenting out the `package` statement, and if it runs, you'll know where the problem lies.

Note that compiled code is often placed in a different directory than source code. This is the standard for many projects, and integrated development environments will usually do it automatically. The path to the compiled code must still be found by the JVM through the CLASSPATH.

Java Access Specifiers

The Java access specifiers `public`, `protected`, and `private` are placed in front of definitions for members in your class, both fields and methods. Each access specifier only controls the access for that particular definition.

If you don't provide an access specifier, it means "package access." So one way or another, everything has some kind of access control. In the following sections, you'll learn about various kinds of access.

Package Access

All the examples before this chapter have only used the `public` access specifier, or no specifier (*default access*). Default access has no keyword, and is commonly called *package access* (sometimes "friendly"). It means that all the other classes in the current package have access to that member. To all the classes outside of this package, the member appears as `private`. A compilation unit—a file—can belong to only a single package so all the classes within a single compilation unit are automatically available to each other via package access.

Package access groups related classes into a package so they can easily interact with each other. Classes in a package grant mutual access to their package-access members, so you "own" the code in that package. It makes sense that only code you own should have package access to other code you own. Package access is one reason for grouping classes together in a package. In many languages, the way you organize your definitions in files can be arbitrary, but in Java you're compelled to organize them in a sensible fashion. In addition, you'll probably exclude classes that shouldn't have access to the classes defined in the current package.

The class controls the code that has access to its members. Code from another package can't just come around and say, "Hi, I'm a friend of `Bob`s!" and expect to be shown the `protected`, package-access, and `private` members of `Bob`. The only way to grant access to a member is to:

1. Make the member `public`. Then everybody, everywhere, can access it.
2. Give the member package access by leaving off any access specifier, and put the other classes in the same package. Then the other classes in that package can access the member.
3. As you'll see in the Reuse chapter, when inheritance is introduced, an inherited class can access a `protected` member as well as a `public` member (but not `private` members). It can access package-access members only if the two classes are in the same package. But don't worry about inheritance and `protected` right now.
4. Provide "accessor/mutator" methods (also known as "get/set" methods) that read and change the value.

`public`: Interface Access

When you use the `public` keyword, it means the member declaration that immediately follows `public` is available to everyone, in particular to the client programmer who uses the library. Suppose you define a package `dessert` containing the following compilation unit:

```
// hiding/dessert/Cookie.java
// Creates a library
package hiding.dessert;

public class Cookie {
  public Cookie() {
   System.out.println("Cookie constructor");
  }
  void bite() { System.out.println("bite"); }
}
```

Remember, the class file produced by `Cookie.java` must reside in a subdirectory called `dessert`, in a directory under `hiding` (indicating the Implementation Hiding chapter of this book) that must be under one of the CLASSPATH directories. Don't make the mistake of thinking that Java will always look at the current directory as one of the starting points for searching. If you don't have a `.` as one of the paths in your CLASSPATH, Java won't look there.

Now we can create a program that uses `Cookie`:

```
// hiding/Dinner.java
// Uses the library
import hiding.dessert.*;

public class Dinner {
  public static void main(String[] args) {
    Cookie x = new Cookie();
    //- x.bite(); // Can't access
  }
}
/* Output:
Cookie constructor
*/
```

You can create a `Cookie` object because its constructor is `public` and the class is `public`. (We'll look more at the concept of a `public` class later.) However, the `bite()` member is inaccessible inside `Dinner.java` because `bite()` provides access only within package `dessert`, so the compiler prevents you from using it.

The Default Package

You might be surprised to discover that the following code compiles, even though it appears to break the rules:

```
// hiding/Cake.java
// Accesses a class in a separate compilation unit

class Cake {
  public static void main(String[] args) {
    Pie x = new Pie();
    x.f();
  }
}
/* Output:
Pie.f()
*/
```

In a second file in the same directory:

```
// hiding/Pie.java
// The other class

class Pie {
  void f() { System.out.println("Pie.f()"); }
}
```

Initially, these seem like completely foreign files, and yet `Cake` is able to create a `Pie` object and call its `f()` method. (Note you must have `.` in your CLASSPATH for the files to compile.) You'd typically think `Pie` and `f()` have package access and are therefore not available to `Cake`. They *do* have package access—that part is correct. The reason they are available in `Cake.java` is because they are in the same directory and have no explicit package name. Java treats files like this as implicitly part of the "default package" for that directory, and thus they provide package access to all the other files in that directory.

`private`: You Can't Touch That!

The `private` keyword means no one can access that member except the class that contains that member, inside methods of that class. Other classes in the same package cannot access `private` members, so it's as if you're even insulating the class against yourself. On the other hand, it's not unlikely that a package might be created by several people collaborating together. With `private`, you can freely change that member without worrying whether it affects another class in the same package.

Default package access often provides an adequate amount of hiding; remember, a package-access member is inaccessible to the client programmer using the class. This is nice because default access is the one you normally use (and the one that you'll get if you forget to add any access control). Thus, you'll typically think about access for the members you make `public` for the client programmer. As a result, you might initially think you won't use the `private` keyword very often because it's tolerable to get away without it. However, the consistent use of `private` is important, especially where multithreading is concerned. (As you'll see in the Concurrent Programming chapter.)

Here's an example of `private`:

```
// hiding/IceCream.java
// Demonstrates "private" keyword

class Sundae {
  private Sundae() {}
  static Sundae makeASundae() {
    return new Sundae();
  }
}

public class IceCream {
  public static void main(String[] args) {
    //- Sundae x = new Sundae();
    Sundae x = Sundae.makeASundae();
  }
}
```

This shows an example where `private` comes in handy: To control how an object is created and prevent someone from directly accessing a particular constructor (or

all of them). In the preceding example, you cannot create a `Sundae` object via its constructor; instead, you must call the `makeASundae()` method to do it for you.[72]

Any method that you're certain is only a "helper" method for that class can be made `private`, to ensure you don't accidentally use it elsewhere in the package and thus prohibit yourself from changing or removing the method. Making a method `private` retains these options.

The same is true for a `private` field inside a class. Unless you must expose the underlying implementation (less likely than you might think), make fields `private`. However, just because a reference to an object is `private` inside a class doesn't mean that some other object can't have a `public` reference to the same object. (See Appendix: Passing and Returning Objects to learn about aliasing issues.)

`protected`: Inheritance Access

Understanding the `protected` access specifier requires a jump ahead. First, be aware you don't have to understand this section to continue through this book up through inheritance (the Reuse chapter). But for completeness, here is a brief description and example using `protected`.

The `protected` keyword deals with a concept called *inheritance*, that takes an existing class—which we refer to as the *base class*—and adds new members to that class without touching the existing class. You can also change the behavior of existing members of the class. To inherit from a class, you say that your new class `extends` an existing class, like this:

```
class Foo extends Bar {
```

The rest of the class definition looks the same.

If you create a new package and inherit from a class in another package, the only members you can access are the `public` members of the original class. (If you perform the inheritance in the *same* package, you can manipulate all the members with package access.) Sometimes the creator of the base class would like to take a particular member and grant access to derived classes but not the world in general.

[72]There's another effect here: The zero-argument constructor is the only one defined and it's `private` so it prevents *inheritance* of this class (A subject to be introduced later).

That's what `protected` does. `protected` also gives package access—that is, other classes in the same package can access `protected` elements.

If you refer back to the file `Cookie.java`, the following class *cannot* call the package-access member `bite()`:

```
// hiding/ChocolateChip.java
// Can't use package-access member from another package
import hiding.dessert.*;

public class ChocolateChip extends Cookie {
  public ChocolateChip() {
    System.out.println("ChocolateChip constructor");
  }
  public void chomp() {
    //- bite(); // Can't access bite
  }
  public static void main(String[] args) {
    ChocolateChip x = new ChocolateChip();
    x.chomp();
  }
}
/* Output:
Cookie constructor
ChocolateChip constructor
*/
```

If a method `bite()` exists in class `Cookie`, it also exists in any class inherited from `Cookie`. But `bite()` has package access and is in a foreign package so it's unavailable to us in this package. You can make it `public`, but then everyone has access, and maybe that's not what you want. If you change the class `Cookie` as follows:

```
// hiding/cookie2/Cookie.java
package hiding.cookie2;

public class Cookie {
  public Cookie() {
    System.out.println("Cookie constructor");
  }
  protected void bite() {
    System.out.println("bite");
  }
}
```

`bite()` becomes accessible to anyone inheriting from `Cookie`:

```
// hiding/ChocolateChip2.java
import hiding.cookie2.*;

public class ChocolateChip2 extends Cookie {
  public ChocolateChip2() {
   System.out.println("ChocolateChip2 constructor");
  }
  public void chomp() { bite(); } // Protected method
  public static void main(String[] args) {
    ChocolateChip2 x = new ChocolateChip2();
    x.chomp();
  }
}
/* Output:
Cookie constructor
ChocolateChip2 constructor
bite
*/
```

Although `bite()` also has package access, it is *not* `public`.

Package Access vs. Public Constructors

When you define a class with package access, you can give it a `public` constructor without any complaints from the compiler:

```
// hiding/packageaccess/PublicConstructor.java
package hiding.packageaccess;

class PublicConstructor {
  public PublicConstructor() {}
}
```

A tool like Checkstyle, which you can run with the command `gradlew hiding:checkstyleMain`, points out that this is false advertising, and technically an error. You can't actually access this so-called `public` constructor from outside the package:

```
// hiding/CreatePackageAccessObject.java
// {WillNotCompile}
import hiding.packageaccess.*;

public class CreatePackageAccessObject {
  public static void main(String[] args) {
    new PublicConstructor();
  }
}
```

If you compile this file by hand, you'll get a compiler error message:

```
CreatePackageAccessObject.java:6: error:
PublicConstructor is not public in hiding.packageaccess;
cannot be accessed from outside package
    new PublicConstructor();
        ^
1 error
```

Thus, declaring a constructor `public` inside a package-access class doesn't actually make it `public`, and it should probably be flagged with a compiler error at the point of declaration.

Interface and Implementation

Access control is often called *implementation hiding*. Wrapping data and methods within classes in combination with implementation hiding is called *encapsulation*.[73] The result is a data type with characteristics and behaviors.

[73] However, people often refer to implementation hiding alone as encapsulation.

Access control puts boundaries within a data type for two important reasons. The first is to establish what client programmers can and cannot use. You build your internal mechanisms into the structure without worrying that the client programmers will accidentally treat the internals as part of the interface they should be using.

This feeds directly into the second reason: to separate interface from implementation. If the structure is used in a set of programs, but client programmers can't do anything but send messages to the `public` interface, you are free to change anything that's *not* `public` (e.g., package access, `protected`, or `private`) without breaking client code.

For clarity, you might prefer a style of creating classes that puts the `public` members at the beginning, followed by the `protected`, package-access, and `private` members. The advantage is that the user of the class can then read down from the top and see first what's important to them (the `public` members, because they can be accessed outside the file), and stop reading when they encounter the non-`public` members, which are part of the internal implementation:

```
// hiding/OrganizedByAccess.java

public class OrganizedByAccess {
  public void pub1() { /* ... */ }
  public void pub2() { /* ... */ }
  public void pub3() { /* ... */ }
  private void priv1() { /* ... */ }
  private void priv2() { /* ... */ }
  private void priv3() { /* ... */ }
  private int i;
  // ...
}
```

This makes it only partially easier to read, because the interface and implementation are still mixed together. That is, you still see the source code—the implementation—because it's right there in the class. In addition, the comment documentation supported by Javadoc lessens the importance of code readability by the client programmer. Displaying the interface to the consumer of a class is really the job of the *class browser*, a tool that shows all available classes and what you can do with them (i.e., what members are available). In Java, the JDK documentation gives the same effect as a class browser.

Class Access

Access specifiers also determine which classes *within* a library are available to the users of that library. If you want a class to be available to a client programmer, you use the `public` keyword on the entire class definition. This controls whether the client programmer can even create an object of the class.

To control access for a class, the specifier must appear before the keyword `class`:

```
public class Widget {
```

If the name of your library is `hiding`, any client programmer can access `Widget` by saying:

```
import hiding.Widget;
```

or

```
import hiding.*;
```

There are additional constraints:

1. There can be only one `public` class per compilation unit (file). The idea is that each compilation unit has a single public interface represented by that `public` class. It can have as many supporting package-access classes as you want. More than one `public` class inside a compilation unit produces a compile-time error message.
2. The name of the `public` class must exactly match the name of the file containing the compilation unit, including capitalization. So for `Widget`, the name of the file must be `Widget.java`, not `widget.java` or `WIDGET.java`. Again, you'll get a compile-time error if they don't agree.
3. It is possible, though not typical, for a compilation unit to lack a `public` class. Here, you can name the file whatever you like (although naming it arbitrarily is confusing to people reading and maintaining the code).

What if you've got a class inside `hiding` that you're only using to accomplish the tasks performed by `Widget` or some other `public` class in `hiding`? You don't want to bother creating documentation for the client programmer, and you think

sometime later you might completely change things and rip out your class altogether, substituting a different one. To give you this flexibility, ensure that no client programmers become dependent on your particular implementation details hidden inside `hiding` by leaving the `public` keyword off the class, to give it package access.

When you create a package-access class, it still makes sense to make the fields of the class `private`—you should always make fields as `private` as possible—but it's generally reasonable to give the methods the same access as the class (package access). A package-access class is usually used only within the package so you only make the methods of such a class `public` if you're forced, and in those cases, the compiler will tell you.

Note that a class cannot be `private` (that would make it inaccessible to anyone but the class) or `protected`.[74] So you have only two choices for class access: package access or `public`. To prevent access to that class, make all the constructors `private`, thereby prohibiting anyone but you, inside a `static` member of the class, from creating an object of that class:

```
// hiding/Lunch.java
// Demonstrates class access specifiers. Make a class
// effectively private with private constructors:

class Soup1 {
  private Soup1() {}
  public static Soup1 makeSoup() {          // [1]
    return new Soup1();
  }
}

class Soup2 {
  private Soup2() {}
  private static Soup2 ps1 = new Soup2();   // [2]
  public static Soup2 access() {
    return ps1;
  }
  public void f() {}
}

// Only one public class allowed per file:
```

[74] Actually, an *inner class* can be private or protected, but that's a special case. These are introduced in the Inner Classes chapter.

```
public class Lunch {
  void testPrivate() {
    // Can't do this! Private constructor:
    //- Soup1 soup = new Soup1();
  }
  void testStatic() {
    Soup1 soup = Soup1.makeSoup();
  }
  void testSingleton() {
    Soup2.access().f();
  }
}
```

You can create an object via a `static` method using **[1]**. You can also create a static object and return a reference when the user requests it, as in **[2]**.

Up to now, most of the methods return either `void` or a primitive type, so the definition in **[1]** might look a little confusing at first. The word `Soup1` before the method name (`makeSoup`) tells what the method returns. So far, this has usually been `void`, which means it returns nothing. But you can also return a reference to an object, which happens here. This method returns a reference to an object of class `Soup1`.

The classes `Soup1` and `Soup2` show how to prevent direct creation of a class by making all the constructors `private`. Remember that if you don't explicitly create at least one constructor, the zero-argument constructor (constructor with no arguments) is created for you. By writing the zero-argument constructor, it won't be created automatically. By making it `private`, no one can create an object of that class. But now how does anyone use this class? The preceding example shows two options. In `Soup1`, a `static` method is created that creates a new `Soup1` and returns a reference to it. This can be useful to perform extra operations on the `Soup1` before returning it, or to keep count of how many `Soup1` objects to create (perhaps to restrict their population).

`Soup2` uses what's called a *design pattern*. This particular pattern is called a *Singleton*, because it allows only a single object to ever be created. The object of class `Soup2` is created as a `static private` member of `Soup2`, so there's one and only one, and you can't get at it except through the `public` method `access()`.

Summary

Boundaries are important in any relationship, respected by all parties involved. When you create a library, you establish a relationship with the user of that library—the client programmer—who is another programmer, but one using your library to build an application or a bigger library.

Without rules, client programmers can do anything they want with all the members of a class, even if you might prefer they don't directly manipulate some of the members. Everything's naked to the world.

This chapter looked at how classes are built to form libraries: first, the way a group of classes is packaged within a library, and second, the way the class controls access to its members.

It is estimated that a C programming project begins to break down somewhere between 50K and 100K lines of code because C has a single namespace, and names begin to collide, causing extra management overhead. In Java, the `package` keyword, the package naming scheme, and the `import` keyword give you complete control over names, so the issue of name collision is easily avoided.

There are two reasons for controlling access to members. The first is to keep users' hands off portions they shouldn't touch. These pieces are necessary for the internal operations of the class, but not part of the interface that the client programmer needs. So making methods and fields `private` is a service to client programmers, because they can easily see what's important to them and what they can ignore. It simplifies their understanding of the class.

The second and most important reason for access control is to allow the library designer to change the internal workings of the class without worrying it will affect the client programmer. You might, for example, build a class one way at first, then discover that restructuring your code will provide much greater speed. If the interface and implementation are clearly separated and protected, you can accomplish this without forcing client programmers to rewrite their code. Access control ensures that no client programmer becomes dependent on any part of the underlying implementation of a class.

When you have the ability to change the underlying implementation, you not only have the freedom to improve your design, you also have the freedom to make mis-

takes. No matter how carefully you plan and design, you'll make mistakes. Knowing it's relatively safe to make these mistakes means you'll be more experimental, you'll learn more quickly, and you'll finish your project sooner.

The `public` interface to a class is what the user *does* see, so that is the most important part of the class to get "right" during analysis and design. Even that allows you some leeway for change. If you don't get the interface right the first time, you can *add* more methods, as long as you don't remove any that client programmers have already used in their code.

Notice that access control focuses on a relationship—and a kind of communication—between a library creator and the external clients of that library. There are many situations where this is not the case. For example, if you write all the code yourself, or you work in close quarters with a small team and everything goes into the same package. These situations have a different kind of communication, and rigid adherence to access rules might not be optimal. Default (package) access might be just fine.

Reuse

> One of the most compelling reasons for object-oriented programming is code reuse.

In procedural languages like C, "reuse" often means "copying code," and this is equally easy to do in any language. But it doesn't work very well. Like everything in Java, the solution revolves around the class. You reuse code by creating new classes, but instead of creating them from scratch, you use existing classes that someone has already built and debugged.

The trick is to use the classes without soiling the existing code. In this chapter you'll see two ways to accomplish this. The first is straightforward: You create objects of your existing class inside the new class. This is called *composition*, because the new class is composed of objects of existing classes. You're reusing the functionality of the code, not its form.

The second approach is more subtle. It creates a new class as a *type of* an existing class. You literally take the form of the existing class and add code to it without modifying the existing class. This technique is called *inheritance*, and the compiler does most of the work. Inheritance is one of the cornerstones of object-oriented programming, and has additional implications explored in the Polymorphism chapter.

Much syntax and behavior are similar for both composition and inheritance (which makes sense because they are both ways of making new types from existing types). In this chapter, you'll learn about these code reuse mechanisms.

Composition Syntax

Composition has been used frequently in the examples you've already seen. You simply place object references inside new classes. For example, suppose you'd like an object that holds several `String` objects, a couple of primitives, and an object of another class. For the non-primitive objects, you put references inside your new class, but you define the primitives directly:

```
// reuse/SprinklerSystem.java
// Composition for code reuse

class WaterSource {
  private String s;
  WaterSource() {
    System.out.println("WaterSource()");
    s = "Constructed";
  }
  @Override public String toString() { return s; }
}

public class SprinklerSystem {
  private String valve1, valve2, valve3, valve4;
  private WaterSource source = new WaterSource();
  private int i;
  private float f;
  @Override public String toString() {
    return
      "valve1 = " + valve1 + " " +
      "valve2 = " + valve2 + " " +
      "valve3 = " + valve3 + " " +
      "valve4 = " + valve4 + "\n" +
      "i = " + i + " " + "f = " + f + " " +
      "source = " + source;                    // [1]
  }
  public static void main(String[] args) {
    SprinklerSystem sprinklers = new SprinklerSystem();
    System.out.println(sprinklers);
  }
}
/* Output:
WaterSource()
valve1 = null valve2 = null valve3 = null valve4 = null
i = 0 f = 0.0 source = Constructed
*/
```

One of the methods defined in both classes is special: `toString()`. Every non-primitive object has a `toString()` method, and it's called in special situations when the compiler wants a `String` but it has an object. So in **[1]**,the compiler sees you trying to "add" a `String` object ("`source =` ") to a `WaterSource`. Because you can only add a `String` to another `String`, it says, "I'll turn `source` into a `String` by

calling `toString()`." Then it can combine the two `Strings` and pass the resulting `String` to `System.out.println()`. To allow this behavior with any class you create, you need only write a `toString()` method.

The `@Override` annotation is used on `toString()` to tell the compiler to make sure we are overriding properly. `@Override` is optional, but it helps verify you don't misspell (or, more subtly, mis-type uppercase or lowercase letters), or make other common mistakes.

Primitive fields in a class are automatically initialized to zero, as noted in the Objects Everywhere chapter. But the object references are initialized to `null`, and if you try to call methods for any of them, you'll get an exception—a runtime error. Conveniently, you can still print a `null` reference without throwing an exception.

It makes sense that the compiler doesn't just create a default object for every reference, because that would incur unnecessary overhead in many cases. There are four ways to initialize references:

1. When the objects are defined. This means they'll always be initialized before the constructor is called.
2. In the constructor for that class.
3. Right before you actually use the object. This is often called *lazy initialization*. It can reduce overhead in situations where object creation is expensive and the object doesn't need to be created every time.
4. Using *instance initialization*.

All four approaches are shown here:

```
// reuse/Bath.java
// Constructor initialization with composition

class Soap {
  private String s;
  Soap() {
    System.out.println("Soap()");
    s = "Constructed";
  }
  @Override public String toString() { return s; }
}
```

```
public class Bath {
  private String // Initializing at point of definition:
    s1 = "Happy",
    s2 = "Happy",
    s3, s4;
  private Soap castile;
  private int i;
  private float toy;
  public Bath() {
    System.out.println("Inside Bath()");
    s3 = "Joy";
    toy = 3.14f;
    castile = new Soap();
  }
  // Instance initialization:
  { i = 47; }
  @Override public String toString() {
    if(s4 == null) // Delayed initialization:
      s4 = "Joy";
    return
      "s1 = " + s1 + "\n" +
      "s2 = " + s2 + "\n" +
      "s3 = " + s3 + "\n" +
      "s4 = " + s4 + "\n" +
      "i = " + i + "\n" +
      "toy = " + toy + "\n" +
      "castile = " + castile;
  }
  public static void main(String[] args) {
    Bath b = new Bath();
    System.out.println(b);
  }
}
/* Output:
Inside Bath()
Soap()
s1 = Happy
s2 = Happy
s3 = Joy
s4 = Joy
i = 47
toy = 3.14
```

```
castile = Constructed
*/
```

In the `Bath` constructor, a statement is executed before any of the initializations take place. When you don't initialize at the point of definition, there's still no guarantee that you'll perform any initialization before you send a message to an object reference—an un-initialized reference will produce a runtime exception if you try to call a method on it.

When `toString()` is called it fills in `s4` so all the fields are properly initialized by the time they are used.

Inheritance Syntax

Inheritance is an integral part of all object-oriented languages. It turns out you're always inheriting when you create a class, because unless you explicitly inherit from some other class, you implicitly inherit from Java's standard root class `Object`.

The syntax for composition is obvious, but inheritance uses a special syntax. When you inherit, you say, "This new class is like that old class." You state this in code before the opening brace of the class body, using the keyword `extends` followed by the name of the *base class*. When you do this, you automatically get all the fields and methods in the base class. Here's an example:

```
// reuse/Detergent.java
// Inheritance syntax & properties

class Cleanser {
  private String s = "Cleanser";
  public void append(String a) { s += a; }
  public void dilute() { append(" dilute()"); }
  public void apply() { append(" apply()"); }
  public void scrub() { append(" scrub()"); }
  @Override public String toString() { return s; }
  public static void main(String[] args) {
    Cleanser x = new Cleanser();
    x.dilute(); x.apply(); x.scrub();
    System.out.println(x);
  }
```

```
}

public class Detergent extends Cleanser {
  // Change a method:
  @Override public void scrub() {
    append(" Detergent.scrub()");
    super.scrub(); // Call base-class version
  }
  // Add methods to the interface:
  public void foam() { append(" foam()"); }
  // Test the new class:
  public static void main(String[] args) {
    Detergent x = new Detergent();
    x.dilute();
    x.apply();
    x.scrub();
    x.foam();
    System.out.println(x);
    System.out.println("Testing base class:");
    Cleanser.main(args);
  }
}
/* Output:
Cleanser dilute() apply() Detergent.scrub() scrub()
foam()
Testing base class:
Cleanser dilute() apply() scrub()
*/
```

This demonstrates a number of features. First, in the `Cleanser` `append()` method, `Strings` are concatenated to `s` using the `+=` operator, one of the operators (along with `+`) that the Java designers "overloaded" to work with `Strings`.

Second, both `Cleanser` and `Detergent` contain a `main()` method. You can create a `main()` for each of your classes; this allows easy testing for each class. You don't need to remove the `main()` when you're finished; you can leave it in for later testing. Even if you have many classes in a program, the only `main()` that runs is the one invoked on the command line. So here, when you say `java Detergent`, `Detergent.main()` is called. But you can also say `java Cleanser` to invoke `Cleanser.main()`, even though `Cleanser` is not a `public` class. Even if a class has package access, a `public` `main()` is accessible.

Here, `Detergent.main()` calls `Cleanser.main()` explicitly, passing it the same arguments from the command line (of course, you can pass it any `String` array).

All methods in `Cleanser` are `public`. Remember that if you leave off any access specifier, the member defaults to package access, which allows access only to package members. Thus, *within this package*, anyone could use those methods if there were no access specifier. `Detergent` would have no trouble, for example. However, if a class from some other package were to inherit from `Cleanser`, it could access only `public` members. So to allow for inheritance, as a general rule make all fields `private` and all methods `public`. (`protected` members also allow access by derived classes; you'll learn about this later.) In particular cases you must make adjustments, but this is a useful guideline.

`Cleanser` has a set of methods in its interface: `append()`, `dilute()`, `apply()`, `scrub()`, and `toString()`. Because `Detergent` is *derived from* `Cleanser` (via the `extends` keyword), it automatically gets all these methods in its interface, even though you don't see them all explicitly defined in `Detergent`. You can think of inheritance, then, as reusing the class.

As seen in `scrub()`, it's possible to take a method that's been defined in the base class and modify it. Here, you might call the method from the base class inside the new version. But inside `scrub()`, you cannot simply call `scrub()` because that would produce a recursive call. To solve this problem, Java's `super` keyword refers to the "superclass" (base class) that the current class inherits. Thus the expression `super.scrub()` calls the base-class version of the method `scrub()`.

When inheriting, you're not restricted to using the methods of the base class. You can also add new methods to the derived class exactly the way you add any method to a class: Just define it. The method `foam()` is an example.

In `Detergent.main()` you see that for a `Detergent` object, you can call all the methods available in `Cleanser` as well as in `Detergent` (such as `foam()`).

Initializing the Base Class

There are now two classes involved: the base class and the derived class. It can be confusing to imagine the resulting object produced by a derived class. From the outside, it looks like the new class has the same interface as the base class and maybe

some additional methods and fields. But inheritance doesn't just copy the interface of the base class. When you create an object of the derived class, it contains within it a *subobject* of the base class. This subobject is the same as if you had created an object of the base class by itself. It's just that from the outside, the subobject of the base class is wrapped within the derived-class object.

It's essential that the base-class subobject be initialized correctly, and there's only one way to guarantee this: Perform the initialization in the constructor by calling the base-class constructor, which has all the appropriate knowledge and privileges to perform the base-class initialization. Java automatically inserts calls to the base-class constructor in the derived-class constructor. The following example shows this with three levels of inheritance:

```
// reuse/Cartoon.java
// Constructor calls during inheritance

class Art {
  Art() {
    System.out.println("Art constructor");
  }
}

class Drawing extends Art {
  Drawing() {
    System.out.println("Drawing constructor");
  }
}

public class Cartoon extends Drawing {
  public Cartoon() {
    System.out.println("Cartoon constructor");
  }
  public static void main(String[] args) {
    Cartoon x = new Cartoon();
  }
}
/* Output:
Art constructor
Drawing constructor
Cartoon constructor
*/
```

The construction happens from the base "outward," so the base class is initialized before the derived-class constructors can access it. Even if you don't create a constructor for `Cartoon`, the compiler will synthesize a zero-argument constructor for you that calls the base-class constructor. Try removing the `Cartoon` constructor to see this.

Constructors with Arguments

The preceding example has all zero-argument constructors; that is, they have no parameters. It's easy for the compiler to call these because there's no question about what arguments to pass. If there isn't a zero-argument base-class constructor, or if you must call a base-class constructor that has arguments, you must explicitly write a call to the base-class constructor using the `super` keyword and the appropriate argument list:

```
// reuse/Chess.java
// Inheritance, constructors and arguments

class Game {
  Game(int i) {
    System.out.println("Game constructor");
  }
}

class BoardGame extends Game {
  BoardGame(int i) {
    super(i);
    System.out.println("BoardGame constructor");
  }
}

public class Chess extends BoardGame {
  Chess() {
    super(11);
    System.out.println("Chess constructor");
  }
  public static void main(String[] args) {
    Chess x = new Chess();
  }
}
```

```
/* Output:
Game constructor
BoardGame constructor
Chess constructor
*/
```

If you don't call the base-class constructor inside the `BoardGame` constructor, the compiler will complain it can't find a constructor of the form `Game()`. In addition, the call to the base-class constructor *must* be the first action inside the derived-class constructor. (The compiler reminds you if you get it wrong.)

Delegation

A third relationship, not directly supported by Java, is called *delegation*. This is midway between inheritance and composition, because you place a member object in the class you're building (like composition), but at the same time you expose all the methods from the member object in your new class (like inheritance). For example, a spaceship needs a control module:

```
// reuse/SpaceShipControls.java

public class SpaceShipControls {
  void up(int velocity) {}
  void down(int velocity) {}
  void left(int velocity) {}
  void right(int velocity) {}
  void forward(int velocity) {}
  void back(int velocity) {}
  void turboBoost() {}
}
```

One way to build a space ship is to use inheritance:

```
// reuse/DerivedSpaceShip.java

public class
DerivedSpaceShip extends SpaceShipControls {
  private String name;
  public DerivedSpaceShip(String name) {
    this.name = name;
  }
  @Override public String toString() {
    return name;
  }
  public static void main(String[] args) {
    DerivedSpaceShip protector =
        new DerivedSpaceShip("NSEA Protector");
    protector.forward(100);
  }
}
```

However, a `DerivedSpaceShip` isn't really "a type of" `SpaceShipControls`, even if, for example, you "tell" a `DerivedSpaceShip` to go `forward()`. It's more accurate to say that a space ship *contains* `SpaceShipControls`, and at the same time all the methods in `SpaceShipControls` are exposed in a space ship. Delegation solves the dilemma:

```
// reuse/SpaceShipDelegation.java

public class SpaceShipDelegation {
  private String name;
  private SpaceShipControls controls =
    new SpaceShipControls();
  public SpaceShipDelegation(String name) {
    this.name = name;
  }
  // Delegated methods:
  public void back(int velocity) {
    controls.back(velocity);
  }
  public void down(int velocity) {
    controls.down(velocity);
  }
  public void forward(int velocity) {
    controls.forward(velocity);
```

```
  }
  public void left(int velocity) {
    controls.left(velocity);
  }
  public void right(int velocity) {
    controls.right(velocity);
  }
  public void turboBoost() {
    controls.turboBoost();
  }
  public void up(int velocity) {
    controls.up(velocity);
  }
  public static void main(String[] args) {
    SpaceShipDelegation protector =
      new SpaceShipDelegation("NSEA Protector");
    protector.forward(100);
  }
}
```

The methods are forwarded to the underlying `controls` object, and the interface is thus the same as it is with inheritance. However, you have more control with delegation because you can choose to provide only a subset of the methods in the member object.

Although the Java language doesn't support delegation, development tools often do. The above example, for instance, was automatically generated using the JetBrains Idea IDE.

Combining Composition and Inheritance

You'll often use composition and inheritance together. The following example shows the creation of class using both inheritance and composition, along with the necessary constructor initialization:

```
// reuse/PlaceSetting.java
// Combining composition & inheritance

class Plate {
  Plate(int i) {
    System.out.println("Plate constructor");
  }
}

class DinnerPlate extends Plate {
  DinnerPlate(int i) {
    super(i);
    System.out.println("DinnerPlate constructor");
  }
}

class Utensil {
  Utensil(int i) {
    System.out.println("Utensil constructor");
  }
}

class Spoon extends Utensil {
  Spoon(int i) {
    super(i);
    System.out.println("Spoon constructor");
  }
}

class Fork extends Utensil {
  Fork(int i) {
    super(i);
    System.out.println("Fork constructor");
  }
}

class Knife extends Utensil {
  Knife(int i) {
    super(i);
    System.out.println("Knife constructor");
  }
}
```

```
// A cultural way of doing something:
class Custom {
  Custom(int i) {
    System.out.println("Custom constructor");
  }
}

public class PlaceSetting extends Custom {
  private Spoon sp;
  private Fork frk;
  private Knife kn;
  private DinnerPlate pl;
  public PlaceSetting(int i) {
    super(i + 1);
    sp = new Spoon(i + 2);
    frk = new Fork(i + 3);
    kn = new Knife(i + 4);
    pl = new DinnerPlate(i + 5);
    System.out.println("PlaceSetting constructor");
  }
  public static void main(String[] args) {
    PlaceSetting x = new PlaceSetting(9);
  }
}
/* Output:
Custom constructor
Utensil constructor
Spoon constructor
Utensil constructor
Fork constructor
Utensil constructor
Knife constructor
Plate constructor
DinnerPlate constructor
PlaceSetting constructor
*/
```

Although the compiler forces you to initialize the base classes, and requires you do it right at the beginning of the constructor, it doesn't watch over you to make sure you initialize the member objects.

Notice how cleanly the classes are separated. You don't even need the source code for the methods to reuse the code. At most, you just import a package. (This is true for both inheritance and composition.)

Guaranteeing Proper Cleanup

Java doesn't have the C++ concept of a *destructor*, a method that is automatically called when an object is destroyed. The reason is probably that in Java, the practice is simply to forget about objects rather than to destroy them, allowing the garbage collector to reclaim memory as necessary.

Often this is fine, but there are times when your class might perform some activities during its lifetime that require cleanup. The Housekeeping chapter explained that you can't know when the garbage collector is called, or even *if* it is called. So if you want something cleaned up for a class, you must explicitly write a special method to do it, and make sure the client programmer knows they must call this method. On top of this—as described in the Exceptions chapter—you must guard against an exception by putting such cleanup in a `finally` clause.

Consider an example of a computer-aided design system that draws pictures on the screen:

```
// reuse/CADSystem.java
// Ensuring proper cleanup
// {java reuse.CADSystem}
package reuse;

class Shape {
  Shape(int i) {
    System.out.println("Shape constructor");
  }
  void dispose() {
    System.out.println("Shape dispose");
  }
}

class Circle extends Shape {
  Circle(int i) {
    super(i);
    System.out.println("Drawing Circle");
```

```
  }
  @Override void dispose() {
    System.out.println("Erasing Circle");
    super.dispose();
  }
}

class Triangle extends Shape {
  Triangle(int i) {
    super(i);
    System.out.println("Drawing Triangle");
  }
  @Override void dispose() {
    System.out.println("Erasing Triangle");
    super.dispose();
  }
}

class Line extends Shape {
  private int start, end;
  Line(int start, int end) {
    super(start);
    this.start = start;
    this.end = end;
    System.out.println(
      "Drawing Line: " + start + ", " + end);
  }
  @Override void dispose() {
    System.out.println(
      "Erasing Line: " + start + ", " + end);
    super.dispose();
  }
}

public class CADSystem extends Shape {
  private Circle c;
  private Triangle t;
  private Line[] lines = new Line[3];
  public CADSystem(int i) {
    super(i + 1);
    for(int j = 0; j < lines.length; j++)
      lines[j] = new Line(j, j*j);
```

```
    c = new Circle(1);
    t = new Triangle(1);
    System.out.println("Combined constructor");
  }
  @Override public void dispose() {
    System.out.println("CADSystem.dispose()");
    // The order of cleanup is the reverse
    // of the order of initialization:
    t.dispose();
    c.dispose();
    for(int i = lines.length - 1; i >= 0; i--)
      lines[i].dispose();
    super.dispose();
  }
  public static void main(String[] args) {
    CADSystem x = new CADSystem(47);
    try {
      // Code and exception handling...
    } finally {
      x.dispose();
    }
  }
}
/* Output:
Shape constructor
Shape constructor
Drawing Line: 0, 0
Shape constructor
Drawing Line: 1, 1
Shape constructor
Drawing Line: 2, 4
Shape constructor
Drawing Circle
Shape constructor
Drawing Triangle
Combined constructor
CADSystem.dispose()
Erasing Triangle
Shape dispose
Erasing Circle
Shape dispose
Erasing Line: 2, 4
```

```
Shape dispose
Erasing Line: 1, 1
Shape dispose
Erasing Line: 0, 0
Shape dispose
Shape dispose
*/
```

Everything in this system is some kind of `Shape` (itself a kind of `Object`, because it's implicitly inherited from the root class). Each class overrides `Shape`'s `dispose()` method in addition to calling the base-class version of that method using `super`. The specific `Shape` classes—`Circle`, `Triangle`, and `Line`—all have constructors that "draw," although any method called during the lifetime of the object can be responsible for doing something that needs cleanup. Each class has its own `dispose()` method to restore non-memory things back to the way they were before the object existed.

In `main()`, there are two keywords you haven't seen before, and won't be explained in detail until the Exceptions chapter: `try` and `finally`. The `try` keyword indicates that the block that follows (delimited by curly braces) is a *guarded region*, which means it is given special treatment. One of these special treatments is that the code in the `finally` clause following this guarded region is *always* executed, no matter how the `try` block exits. (With exception handling, it's possible to leave a `try` block in a number of non-ordinary ways.) Here, the `finally` clause is saying, "Always call `dispose()` for `x`, no matter what happens."

In your cleanup method (`dispose()`, in this case), you must also pay attention to the calling order for the base-class and member-object cleanup methods in case one subobject depends on another. First perform all cleanup work specific to your class, in the reverse order of creation. (In general, this requires that base-class elements still be viable.) Then call the base-class cleanup method, as demonstrated here.

There are many cases where the cleanup issue is not a problem; you just let the garbage collector do the work. But when you must perform explicit cleanup, diligence and attention are required, because there's not much you can rely on when it comes to garbage collection. The garbage collector might never be called. If it is, it can reclaim objects in any order it wants. You can't rely on garbage collection for anything but memory reclamation. If you want cleanup to take place, make your own cleanup methods and don't use `finalize()`.

Name Hiding

If a Java base class has a method name that's overloaded several times, redefining that method name in the derived class will *not* hide any of the base-class versions. Overloading works regardless of whether the method was defined at this level or in a base class:

```
// reuse/Hide.java
// Overloading a base-class method name in a derived
// class does not hide the base-class versions

class Homer {
  char doh(char c) {
    System.out.println("doh(char)");
    return 'd';
  }
  float doh(float f) {
    System.out.println("doh(float)");
    return 1.0f;
  }
}

class Milhouse {}

class Bart extends Homer {
  void doh(Milhouse m) {
    System.out.println("doh(Milhouse)");
  }
}

public class Hide {
  public static void main(String[] args) {
    Bart b = new Bart();
    b.doh(1);
    b.doh('x');
    b.doh(1.0f);
    b.doh(new Milhouse());
  }
}
/* Output:
doh(float)
```

```
doh(char)
doh(float)
doh(Milhouse)
*/
```

All the overloaded methods of `Homer` are available in `Bart`, even though `Bart` introduces a new overloaded method. As you'll see in the next chapter, it's far more common to override methods of the same name, using exactly the same signature and return type as in the base class. Otherwise it can be confusing.

You've been seeing the Java 5 `@Override` annotation, which is not a keyword but can be used as if it were. When you mean to override a method, you can choose to add this annotation and the compiler will produce an error message if you accidentally overload instead of overriding:

```
// reuse/Lisa.java
// {WillNotCompile}

class Lisa extends Homer {
  @Override void doh(Milhouse m) {
    System.out.println("doh(Milhouse)");
  }
}
```

The `{WillNotCompile}` tag excludes the file from this book's Gradle build, but if you compile it by hand you'll see:

```
method does not override a method from its superclass
```

The `@Override` annotation prevents you from accidentally overloading.

Choosing Composition vs. Inheritance

Both composition and inheritance allow you to place subobjects inside your new class (composition explicitly does this—with inheritance it's implicit). You might wonder about the difference between the two, and when to choose one over the other.

Use composition when you want the functionality of an existing class inside your new class, but not its interface. That is, embed a (usually `private`) object to

implement features in your new class. The user of your new class sees the interface you've defined for the new class rather than the interface from the embedded object.

Sometimes it makes sense to allow the class user to directly access the composition of your new class. For this, make the member objects `public` (you can think of this as a kind of "semi-delegation"). The member objects use implementation hiding themselves, so this is safe. When the user knows you're assembling a bunch of parts, it makes the interface easier to understand. A `car` object is a good example:

```
// reuse/Car.java
// Composition with public objects

class Engine {
  public void start() {}
  public void rev() {}
  public void stop() {}
}

class Wheel {
  public void inflate(int psi) {}
}

class Window {
  public void rollup() {}
  public void rolldown() {}
}

class Door {
  public Window window = new Window();
  public void open() {}
  public void close() {}
}

public class Car {
  public Engine engine = new Engine();
  public Wheel[] wheel = new Wheel[4];
  public Door
    left = new Door(),
    right = new Door(); // 2-door
  public Car() {
    for(int i = 0; i < 4; i++)
      wheel[i] = new Wheel();
```

```
  }
  public static void main(String[] args) {
    Car car = new Car();
    car.left.window.rollup();
    car.wheel[0].inflate(72);
  }
}
```

The composition of a car is part of the analysis of the problem (and not part of the underlying design). Making the members `public` assists the client programmer's understanding of how to use the class and requires less code complexity for the creator of the class. However, keep in mind this is a special case. In general, make fields `private`.

When you inherit, you take an existing class and make a special version of it. This usually means taking a general-purpose class and specializing it for a particular need. With a little thought, you'll see it would make no sense to compose a car using a vehicle object—a car doesn't contain a vehicle, it *is* a vehicle. The *is-a* relationship is expressed with inheritance, and the *has-a* relationship is expressed with composition.

`protected`

Now that you've been introduced to inheritance, the keyword `protected` becomes meaningful. In an ideal world, the `private` keyword would be enough. In real projects, there are times when you hide something from the world at large and yet allow access for members of derived classes.

The `protected` keyword is a nod to pragmatism. It says, "This is `private` as far as the class user is concerned, but available to anyone who inherits from this class or anyone else in the same package." (`protected` also provides package access.)

Although it's possible to create `protected` fields, the best approach is to leave the fields `private` and always preserve your right to change the underlying implementation. You can then allow controlled access to inheritors of your class through `protected` methods:

```
// reuse/Orc.java
// The protected keyword

class Villain {
  private String name;
  protected void set(String nm) { name = nm; }
  Villain(String name) { this.name = name; }
  @Override public String toString() {
    return "I'm a Villain and my name is " + name;
  }
}

public class Orc extends Villain {
  private int orcNumber;
  public Orc(String name, int orcNumber) {
    super(name);
    this.orcNumber = orcNumber;
  }
  public void change(String name, int orcNumber) {
    set(name); // Available because it's protected
    this.orcNumber = orcNumber;
  }
  @Override public String toString() {
    return "Orc " + orcNumber + ": " + super.toString();
  }
  public static void main(String[] args) {
    Orc orc = new Orc("Limburger", 12);
    System.out.println(orc);
    orc.change("Bob", 19);
    System.out.println(orc);
  }
}
/* Output:
Orc 12: I'm a Villain and my name is Limburger
Orc 19: I'm a Villain and my name is Bob
*/
```

`change()` has access to `set()` because it's `protected`. Notice the way `Orc`'s `toString()` method is defined in terms of the base-class version of `toString()`.

Upcasting

The most important aspect of inheritance is not that it provides methods for the new class. It's the relationship expressed between the new class and the base class. This relationship can be summarized by saying, "The new class *is a type of* the existing class."

This description is not just a fanciful way of explaining inheritance—it's supported directly by the language. As an example, consider a base class called `Instrument` that represents musical instruments, and a derived class called `Wind`. Because inheritance guarantees that all methods in the base class are also available in the derived class, any message you can send to the base class can also be sent to the derived class. If the `Instrument` class has a `play()` method, so will `Wind` instruments. This means you can accurately say that a `Wind` object is also a type of `Instrument`. The following example shows how the compiler supports this notion:

```
// reuse/Wind.java
// Inheritance & upcasting

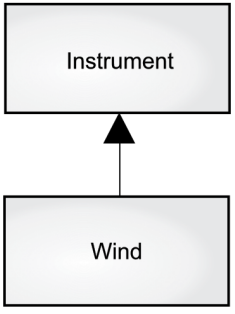
class Instrument {
  public void play() {}
  static void tune(Instrument i) {
    // ...
    i.play();
  }
}

// Wind objects are instruments
// because they have the same interface:
public class Wind extends Instrument {
  public static void main(String[] args) {
    Wind flute = new Wind();
    Instrument.tune(flute); // Upcasting
  }
}
```

The `tune()` method accepts an `Instrument` reference. However, in `Wind.main()` the `tune()` method is handed a `Wind` reference. Given that Java is particular about type checking, it seems strange that a method that accepts one type will readily

accept another type, until you realize that a `Wind` object is also an `Instrument` object, and there's no method that `tune()` could call for an `Instrument` that isn't also in `Wind`. Inside `tune()`, the code works for `Instrument` and anything derived from `Instrument`, and the act of converting a `Wind` reference into an `Instrument` reference is called *upcasting*.

The term is based on the way that class inheritance diagrams have traditionally been drawn: with the root at the top of the page, growing downward. (Of course, you can draw your diagrams any way you find helpful.) The inheritance diagram for `Wind.java` is then:



Casting from a derived type to a base type moves *up* on the inheritance diagram, so it's commonly called *upcasting*. Upcasting is always safe because you're going from a more specific type to a more general type. That is, the derived class is a superset of the base class. It might contain more methods than the base class, but it must contain *at least* the methods in the base class. During the upcast, the class interface can only lose methods, not gain them. This is why the compiler allows upcasting without any explicit casts or other special notation.

You can also perform the reverse of upcasting, called *downcasting*, but this involves a dilemma that is examined further in the next chapter, and in the Reflection chapter.

Composition vs. Inheritance Revisited

In object-oriented programming, the most likely way to create and use code is by packaging data and methods together into a class, and using objects of that class. You'll also use existing classes to build new classes with composition. Less frequently, you'll use inheritance. So although inheritance gets a lot of emphasis when teaching

OOP, it doesn't mean you should use it everywhere you possibly can. On the contrary, use it sparingly, only when it's clear that inheritance is useful. One of the clearest ways to determine whether to use composition or inheritance is to ask whether you'll ever upcast from your new class to the base class. If you must upcast, inheritance is necessary, but if you don't upcast, look closely at whether you need inheritance. The Polymorphism chapter provides one of the most compelling reasons for upcasting, but if you remember to ask, "Do I need to upcast?" you'll have a good tool for deciding between composition and inheritance.

The `final` Keyword

Java's `final` keyword has slightly different meanings depending on context, but in general it says, "This cannot be changed." You can prevent changes for two reasons: design or efficiency. Because these two reasons are quite different, it's possible to misuse the `final` keyword.

The following sections discuss the three places where `final` can be used: for data, methods, and classes.

`final` Data

Many programming languages have a way to tell the compiler that a piece of data is "constant." A constant is useful for two reasons:

1. It can be a *compile-time constant* that won't ever change.
2. It can be a value initialized at run time that you don't want changed.

In the case of a compile-time constant, the compiler is allowed to "fold" the constant value into calculations; that is, the calculation can be performed at compile time, eliminating some runtime overhead. In Java, these sorts of constants must be primitives and are expressed with the `final` keyword. You must provide a value when the constant is defined.

A field that is both `static` and `final` has only one piece of storage that cannot be changed.

When `final` is used with object references rather than primitives, the meaning can be confusing. With a primitive, `final` makes the *value* a constant, but with an object

reference, `final` makes the *reference* a constant. Once the reference is initialized to an object, it can never be changed to point to another object. However, the object itself can be modified; Java does not provide a way to make any arbitrary object a constant. (You can, however, write your class so objects have the effect of being constant.) This restriction includes arrays, which are also objects.

Here's an example that demonstrates `final` fields:

```
// reuse/FinalData.java
// The effect of final on fields
import java.util.*;

class Value {
  int i; // Package access
  Value(int i) { this.i = i; }
}

public class FinalData {
  private static Random rand = new Random(47);
  private String id;
  public FinalData(String id) { this.id = id; }
  // Can be compile-time constants:
  private final int valueOne = 9;
  private static final int VALUE_TWO = 99;
  // Typical public constant:
  public static final int VALUE_THREE = 39;
  // Cannot be compile-time constants:
  private final int i4 = rand.nextInt(20);
  static final int INT_5 = rand.nextInt(20);
  private Value v1 = new Value(11);
  private final Value v2 = new Value(22);
  private static final Value VAL_3 = new Value(33);
  // Arrays:
  private final int[] a = { 1, 2, 3, 4, 5, 6 };
  @Override public String toString() {
    return id + ": " + "i4 = " + i4
      + ", INT_5 = " + INT_5;
  }
  public static void main(String[] args) {
    FinalData fd1 = new FinalData("fd1");
    //- fd1.valueOne++; // Error: can't change value
    fd1.v2.i++; // Object isn't constant!
```

```
    fd1.v1 = new Value(9); // OK -- not final
    for(int i = 0; i < fd1.a.length; i++)
      fd1.a[i]++; // Object isn't constant!
    //- fd1.v2 = new Value(0); // Error: Can't
    //- fd1.VAL_3 = new Value(1); // change reference
    //- fd1.a = new int[3];
    System.out.println(fd1);
    System.out.println("Creating new FinalData");
    FinalData fd2 = new FinalData("fd2");
    System.out.println(fd1);
    System.out.println(fd2);
  }
}
/* Output:
fd1: i4 = 15, INT_5 = 18
Creating new FinalData
fd1: i4 = 15, INT_5 = 18
fd2: i4 = 13, INT_5 = 18
*/
```

`valueOne` and `VALUE_TWO` are `final` primitives with compile-time values so they can both be used as compile-time constants and are not different in any important way. `VALUE_THREE` is the more typical way you'll see such constants defined: `public` so they're usable outside the package, `static` to emphasize there's only one, and `final` to say it's a constant.

By convention, `final static` primitives with constant initial values (that is, compile-time constants) are named with all capitals, with words separated by underscores. (This is just like C constants, which is where that style originated.)

Just because something is `final` doesn't mean its value is known at compile time. This is demonstrated by initializing `i4` and `INT_5` at run time using randomly generated numbers. This portion of the example also shows the difference between making a `final` value `static` or non-`static`. This difference shows up only when the values are initialized at run time because the compile-time values are treated the same by the compiler. (And presumably optimized out of existence.) The difference is shown when you run the program. Note that the values of `i4` for `fd1` and `fd2` are unique, but the value for `INT_5` is not changed by creating the second `FinalData` object. That's because it's `static` and is initialized once upon loading and not each time you create a new object.

The variables `v1` through `VAL_3` demonstrate the meaning of a `final` reference. As you see in `main()`, just because `v2` is `final` doesn't mean you can't change its value. Because it's a reference, `final` just means you cannot rebind `v2` to a new object. The same meaning holds true for an array, which is just another kind of reference. (There is no way I know of to make the array references themselves `final`.) Making references `final` seems less useful than making primitives `final`.

Blank `final`s

Blank finals are `final` fields without initialization values. The compiler ensures that blank `finals` are initialized before use. A `final` field inside a class can now be different for each object while retaining its immutable quality:

```
// reuse/BlankFinal.java
// "Blank" final fields

class Poppet {
  private int i;
  Poppet(int ii) { i = ii; }
}

public class BlankFinal {
  private final int i = 0; // Initialized final
  private final int j; // Blank final
  private final Poppet p; // Blank final reference
  // Blank finals MUST be initialized in constructor:
  public BlankFinal() {
    j = 1; // Initialize blank final
    p = new Poppet(1); // Init blank final reference
  }
  public BlankFinal(int x) {
    j = x; // Initialize blank final
    p = new Poppet(x); // Init blank final reference
  }
  public static void main(String[] args) {
    new BlankFinal();
    new BlankFinal(47);
  }
}
```

You're forced to perform assignments to `final`s either with an expression at the point of definition of the field or in every constructor. This guarantees that the `final` field is always initialized before use.

`final` Arguments

You make arguments `final` by declaring them as such in the argument list. This means that inside the method you cannot change what the argument reference points to:

```
// reuse/FinalArguments.java
// Using "final" with method arguments

class Gizmo {
  public void spin() {}
}

public class FinalArguments {
  void with(final Gizmo g) {
    //- g = new Gizmo(); // Illegal -- g is final
  }
  void without(Gizmo g) {
    g = new Gizmo(); // OK -- g not final
    g.spin();
  }
  // void f(final int i) { i++; } // Can't change
  // You can only read from a final primitive:
  int g(final int i) { return i + 1; }
  public static void main(String[] args) {
    FinalArguments bf = new FinalArguments();
    bf.without(null);
    bf.with(null);
  }
}
```

The methods `f()` and `g()` show what happens when primitive arguments are `final`. You can read the argument, but you can't change it. This feature is primarily used to pass data to anonymous inner classes, which you'll learn about in the Inner Classes chapter.

`final` Methods

There are two reasons for `final` methods. The first is to put a "lock" on the method to prevent an inheriting class from changing that method's meaning by overriding it. This is done for design reasons when you want to make sure that a method's behavior is retained during inheritance.

The second reason `final` methods were suggested in the past is efficiency. In earlier implementations of Java, if you made a method `final`, you allowed the compiler to turn any calls to that method into *inline* calls. When the compiler saw a `final` method call, it could (at its discretion) skip the normal approach of inserting code to perform the method call mechanism (push arguments on the stack, hop over to the method code and execute it, hop back and clean off the stack arguments, and deal with the return value) and instead replace the method call with a copy of the actual code in the method body. This eliminates the overhead of the method call. However, if a method is big, your code begins to bloat, and you probably wouldn't see performance gains from inlining because any speedups in the call and return were dwarfed by the amount of time spent inside the method.

Relatively early in the history of Java, the virtual machine (in particular, the *hotspot* technologies) began detecting these situations and optimizing away the extra indirection. For a long time, using `final` to help the optimizer has been discouraged. You should let the compiler and JVM handle efficiency issues and make a method `final` only to explicitly prevent overriding.[75]

`final` and `private`

Any `private` methods in a class are implicitly `final`. Because you can't access a `private` method, you can't override it. You can add the `final` specifier to a `private` method, but it doesn't give that method any extra meaning.

This can be confusing, because if you try to override a `private` method (which is implicitly `final`), it seems to work, and the compiler doesn't give an error message:

[75] Don't listen to the sirens of premature optimization. If you get your system working and it's too slow, it's doubtful you can fix it with the `final` keyword. Profiling, however, *can* be helpful in speeding up your program.

```
// reuse/FinalOverridingIllusion.java
// It only looks like you can override
// a private or private final method

class WithFinals {
  // Identical to "private" alone:
  private final void f() {
    System.out.println("WithFinals.f()");
  }
  // Also automatically "final":
  private void g() {
    System.out.println("WithFinals.g()");
  }
}

class OverridingPrivate extends WithFinals {
  private final void f() {
    System.out.println("OverridingPrivate.f()");
  }
  private void g() {
    System.out.println("OverridingPrivate.g()");
  }
}

class OverridingPrivate2 extends OverridingPrivate {
  public final void f() {
    System.out.println("OverridingPrivate2.f()");
  }
  public void g() {
    System.out.println("OverridingPrivate2.g()");
  }
}

public class FinalOverridingIllusion {
  public static void main(String[] args) {
    OverridingPrivate2 op2 = new OverridingPrivate2();
    op2.f();
    op2.g();
    // You can upcast:
    OverridingPrivate op = op2;
    // But you can't call the methods:
    //- op.f();
```

```
    //- op.g();
    // Same here:
    WithFinals wf = op2;
    //- wf.f();
    //- wf.g();
  }
}
/* Output:
OverridingPrivate2.f()
OverridingPrivate2.g()
*/
```

"Overriding" can only occur if something is part of the base-class interface. That is, you must upcast an object to its base type and call the same method (the point of this becomes clear in the next chapter). If a method is `private`, it isn't part of the base-class interface. It is just code that's hidden away inside the class, and it just happens to have that name. But if you create a `public`, `protected`, or package-access method with the same name in the derived class, there's no connection to the method that might happen to have that name in the base class. You haven't overridden the method, you've just created a new method. A `private` method is unreachable and effectively invisible so it doesn't factor into anything except for the code organization of the class for which it was defined.

Note that using `@Override` in the above example will produce helpful messages pointing out the mistakes.

`final` Classes

When you say that an entire class is `final` (by preceding its definition with the `final` keyword), you're preventing all inheritance from this class. You do this because, for some reason, the design of your class is such that there is never a need to make any changes, or for safety or security reasons you don't want subclassing.

```
// reuse/Jurassic.java
// Making an entire class final

class SmallBrain {}

final class Dinosaur {
  int i = 7;
  int j = 1;
  SmallBrain x = new SmallBrain();
  void f() {}
}

//- class Further extends Dinosaur {}
// error: Cannot extend final class 'Dinosaur'

public class Jurassic {
  public static void main(String[] args) {
    Dinosaur n = new Dinosaur();
    n.f();
    n.i = 40;
    n.j++;
  }
}
```

The fields of a `final` class can be `final` or not, as you choose. The same rules apply to `final` for fields regardless of whether the class is defined as `final`. However, because it prevents inheritance, all *methods* in a `final` class are implicitly `final` because there's no way to override them. You can include the `final` specifier to a method in a `final` class, but it doesn't add any meaning.

`final` Caution

It can seem sensible to make a method `final` while you're designing a class. You might feel that no one could possibly want to override that method. Sometimes this is true.

But be careful with your assumptions. In general, it's difficult to anticipate how a class can be reused, especially a general-purpose class. If you define a method as `final`, you might prevent the possibility of reusing your class through inheritance

in some other programmer's project simply because you can't imagine it being used that way.

The standard Java library is a good example of this. In particular, the Java 1.0/1.1 `Vector` class was commonly used and might be even more useful if, in the name of efficiency (which was almost certainly an illusion), all the methods hadn't been made `final`. It's easily conceivable you might inherit and override with such a fundamentally useful class, but the designers somehow decided this wasn't appropriate. This is ironic for two reasons. First, `Stack` inherits `Vector`, which says that a `Stack` *is* a `Vector`, which isn't really true from a logical standpoint. Nonetheless, it's a case where the Java designers themselves inherited `Vector`. When they created `Stack` this way, they should have realized that `final` methods were too restrictive.

Second, many of the most important methods of `Vector`, such as `addElement()` and `elementAt()`, are `synchronized`. The Concurrent Programming chapter shows that this imposes a significant performance overhead that probably wipes out any gains provided by `final`. This lends credence to the theory that programmers are consistently bad at guessing where optimizations should occur. It's too bad that such a clumsy design made it into the standard library, where everyone had to cope with it. Fortunately, the modern Java collection library supersedes `Vector` with `ArrayList`, which behaves much more civilly. Unfortunately, there's still new code that uses the old collection library, including `Vector`.

`Hashtable`, another important Java 1.0/1.1 standard library class (later superseded by `HashMap`), does *not* have any `final` methods. As mentioned elsewhere in this book, it's obvious that different classes were designed by different people. The method names in `Hashtable` are much briefer compared to those in `Vector`, another piece of evidence. This is precisely the sort of thing that should *not* be obvious to consumers of a class library. When things are inconsistent, it just makes more work for the user—yet another paean to the value of design and code walkthroughs.

Initialization and Class Loading

In more traditional languages, programs are loaded all at once, as part of the startup process. This is followed by initialization, then the program begins. The process of initialization in these languages must be carefully controlled so the order of

initialization of `statics` doesn't cause trouble. C++, for example, has problems if one `static` expects another `static` to be valid before the second one is initialized.

Java doesn't have this problem because it takes a different approach to loading. This is one of the activities that become easier because everything in Java is an object. Remember that the compiled code for each class exists in its own separate file. That file isn't loaded until the code is needed. In general, you can say that "class code is loaded at the point of first use." This is usually when the first object of that class is constructed, but loading also occurs when a `static` field or `static` method is accessed. The constructor is also a `static` method even though the `static` keyword is not explicit. So to be precise, a class is first loaded when any one of its `static` members is accessed.

The point of first use is also when `static` initialization takes place. All `static` objects and `static` code blocks are initialized in textual order (the order you write them in the class definition), at load time. The `statics` are initialized only once.

Initialization with Inheritance

It's helpful to look at the whole initialization process, including inheritance, to get a full picture of what happens. Consider the following example:

```
// reuse/Beetle.java
// The full process of initialization

class Insect {
  private int i = 9;
  protected int j;
  Insect() {
    System.out.println("i = " + i + ", j = " + j);
    j = 39;
  }
  private static int x1 =
    printInit("static Insect.x1 initialized");
  static int printInit(String s) {
    System.out.println(s);
    return 47;
  }
}
```

```
public class Beetle extends Insect {
  private int k = printInit("Beetle.k initialized");
  public Beetle() {
    System.out.println("k = " + k);
    System.out.println("j = " + j);
  }
  private static int x2 =
    printInit("static Beetle.x2 initialized");
  public static void main(String[] args) {
    System.out.println("Beetle constructor");
    Beetle b = new Beetle();
  }
}
/* Output:
static Insect.x1 initialized
static Beetle.x2 initialized
Beetle constructor
i = 9, j = 0
Beetle.k initialized
k = 47
j = 39
*/
```

When you run `java Beetle`, you first try to access `Beetle.main()` (a `static` method), so the loader goes out and finds the compiled code for the `Beetle` class, in the file `Beetle.class`. While loading that, the loader notices there's a base class, which it also loads. This happens whether or not you make an object of that base class. (Try commenting out the object creation to prove it to yourself.)

If the base class has its own base class, that second base class will also be loaded, and so on. Next, the `static` initialization in the root base class (in this case, `Insect`) is performed, then the next derived class, and so on. This is important because the derived-class `static` initialization might depend on the base-class member being initialized properly.

Now the necessary classes have all been loaded, so the object can be created. First, all the primitives in this object are set to their default values and the object references are set to `null`—this happens in one fell swoop by setting the memory in the object to binary zero. Then the base-class constructor is called. Here the call is automatic, but you can also specify the base-class constructor call (as the first operation in the `Beetle` constructor) by using `super`. The base-class constructor goes through the

same process in the same order as the derived-class constructor. After the base-class constructor completes, the instance variables are initialized in textual order. Finally, the rest of the body of the constructor is executed.

Summary

Both inheritance and composition create new types from existing types. Composition reuses existing types as part of the underlying implementation of the new type, and inheritance reuses the interface.

With inheritance, the derived class has the base-class interface, so it can be *upcast* to the base, which is critical for polymorphism, as you'll see in the next chapter.

Despite the strong emphasis on inheritance in object-oriented programming, when you start a design, prefer composition (or possibly delegation) during the first cut and use inheritance only when it is clearly necessary. Composition tends to be more flexible. In addition, by using the added artifice of inheritance with your member type, you can change the exact type, and thus the behavior, of those member objects at run time. Therefore, you can change the behavior of the composed object at run time.

When designing a system, your goal is to find or create a set of classes where each class has a specific use and is neither too big (encompassing so much functionality it's unwieldy to reuse) nor annoyingly small (you can't use it by itself or without adding functionality). If your designs become too complex, it's often helpful to add more objects by breaking existing objects into smaller parts.

When you start designing a system, remember that program development is an incremental process, just like human learning. It relies on experimentation; you can do as much analysis as you want, but you still won't know all the answers when you set out on a project. You'll have much more success—and more immediate feedback—if you start out to "grow" your project as an organic, evolutionary creature, rather than constructing it all at once like a glass-box skyscraper. Inheritance and composition are two of the most fundamental tools in object-oriented programming that allow you to perform such experiments.

Polymorphism

> "I have been asked, 'Pray, Mr. Babbage, if you put into the machine wrong figures, will the right answers come out?' I am not able to rightly apprehend the kind of confusion of ideas that could provoke such a question."—*Charles Babbage (1791-1871)*
>
> Polymorphism is the third essential feature of an object-oriented programming language, after data abstraction and inheritance.

It provides another dimension of separation of interface from implementation, to decouple *what* from *how*. Polymorphism improves code organization and readability as well as the creation of *extensible* programs that can be "grown" not only during the original creation of the project, but also when new features are desired.

Encapsulation creates new data types by combining characteristics and behaviors. Implementation hiding separates the interface from the implementation by making the details `private`. This sort of mechanical organization makes ready sense to someone with a procedural programming background. But polymorphism deals with decoupling in terms of *types*. In the previous chapter, inheritance enabled you to treat an object as its own type *or* its base type. This treats many types (derived from the same base type) as if they were one type, and a single piece of code works on all those different types equally. The polymorphic method call allows one type to express its distinction from another, similar type, as long as they're both derived from the same base type. This distinction is expressed through differences in behavior of the methods you can call through the base class.

In this chapter, you'll learn about polymorphism (also called *dynamic binding* or *late binding* or *runtime binding*) starting from the basics, with simple examples that strip away everything but the polymorphic behavior of the program.

Upcasting Revisited

In the previous chapter you saw how an object can be used as its own type or as an object of its base type. Taking an object reference and treating it as a reference to its base type is called *upcasting* because of the way inheritance trees are drawn with the base class at the top.

You also saw a problem arise, embodied in the following example of musical instruments. Several of these examples play `Notes` so we first create a separate `Note` enumeration, in a package:

```
// polymorphism/music/Note.java
// Notes to play on musical instruments
package polymorphism.music;

public enum Note {
  MIDDLE_C, C_SHARP, B_FLAT; // Etc.
}
```

`enums` were introduced in the Housekeeping chapter.

Here, `Wind` is a type of `Instrument`:

```
// polymorphism/music/Instrument.java
package polymorphism.music;

class Instrument {
  public void play(Note n) {
    System.out.println("Instrument.play()");
  }
}
```

Therefore, `Wind` inherits `Instrument`:

```
// polymorphism/music/Wind.java
package polymorphism.music;

// Wind objects are instruments
// because they have the same interface:
public class Wind extends Instrument {
  // Redefine interface method:
  @Override public void play(Note n) {
    System.out.println("Wind.play() " + n);
  }
}
```

The method `Music.tune()` accepts an `Instrument` reference, but also anything derived from `Instrument`:

```
// polymorphism/music/Music.java
// Inheritance & upcasting
// {java polymorphism.music.Music}
package polymorphism.music;

public class Music {
  public static void tune(Instrument i) {
    // ...
    i.play(Note.MIDDLE_C);
  }
  public static void main(String[] args) {
    Wind flute = new Wind();
    tune(flute); // Upcasting
  }
}
/* Output:
Wind.play() MIDDLE_C
*/
```

In `main()` you see a `Wind` reference passed to `tune()`, with no cast necessary. This is acceptable—the interface in `Instrument` must exist in `Wind`, because `Wind` inherits `Instrument`. Upcasting from `Wind` to `Instrument` can "narrow" that interface, but it cannot make it anything less than the full interface to `Instrument`.

Forgetting the Object Type

`Music.java` might seem strange to you. Why should anyone intentionally *forget* the type of an object? This "forgetting" is exactly what happens when you upcast, and it seems like it might be much more straightforward if `tune()` takes a `Wind` reference as its argument. This brings up an essential point: If you did that, you'd have to write a new `tune()` for every type of `Instrument` in your system. Suppose you follow this reasoning and add `Stringed` and `Brass` instruments:

```
// polymorphism/music/Music2.java
// Overloading instead of upcasting
// {java polymorphism.music.Music2}
package polymorphism.music;

class Stringed extends Instrument {
  @Override public void play(Note n) {
    System.out.println("Stringed.play() " + n);
  }
}

class Brass extends Instrument {
  @Override public void play(Note n) {
    System.out.println("Brass.play() " + n);
  }
}

public class Music2 {
  public static void tune(Wind i) {
    i.play(Note.MIDDLE_C);
  }
  public static void tune(Stringed i) {
    i.play(Note.MIDDLE_C);
  }
  public static void tune(Brass i) {
    i.play(Note.MIDDLE_C);
  }
  public static void main(String[] args) {
    Wind flute = new Wind();
    Stringed violin = new Stringed();
    Brass frenchHorn = new Brass();
    tune(flute); // No upcasting
```

```
    tune(violin);
    tune(frenchHorn);
  }
}
/* Output:
Wind.play() MIDDLE_C
Stringed.play() MIDDLE_C
Brass.play() MIDDLE_C
*/
```

This works, but there's a major drawback: You must write type-specific methods for each new `Instrument` class you add. This means more programming in the first place, but it also means that to add a new method like `tune()` or a new type of `Instrument`, you've got a lot of work to do. Add the fact that the compiler won't tell you if you forget to overload one of your methods, and the whole process of working with types becomes unmanageable.

Wouldn't it be much nicer to write a single method that takes the base class as its argument, and not worry about any of the specific derived classes? That is, wouldn't it be nice to forget there are derived classes, and write your code to talk only to the base class?

That's exactly what polymorphism enables. However, most programmers who come from a procedural programming background have a bit of trouble with the way polymorphism works.

The Twist

The difficulty with `Music.java` can be seen by running the program. The output is `Wind.play()`. This is clearly the desired output, but it doesn't seem to make sense it would work that way. Look at the `tune()` method:

```
public static void tune(Instrument i) {
  // ...
  i.play(Note.MIDDLE_C);
}
```

It receives an `Instrument` reference. So how can the compiler possibly know this `Instrument` reference points to a `Wind` here and not a `Brass` or `Stringed`? The

compiler can't. To get a deeper understanding of the issue, it's helpful to examine the subject of *binding*.

Method-Call Binding

Connecting a method call to a method body is called *binding*. When binding is performed before the program runs(by the compiler and linker, if there is one), it's called *early binding*. You might not have heard the term before because it has never been an option with procedural languages. C, for example, has only one kind of method call, and that's early binding.

The confusing part of the preceding program revolves around early binding, because the compiler cannot know the correct method to call when it has only an `Instrument` reference.

The solution is called *late binding*, which means the binding occurs at run time, based on the type of object. Late binding is also called *dynamic binding* or *runtime binding*. When a language implements late binding, there must be some mechanism to determine the type of the object at run time and to call the appropriate method. That is, the compiler still doesn't know the object type, but the method-call mechanism finds out and calls the correct method body. The late-binding mechanism varies from language to language, but you can imagine that some sort of type information must be installed in the objects.

All method binding in Java uses late binding unless the method is `static` or `final` (`private` methods are implicitly `final`). This means that ordinarily you don't make decisions about whether late binding will occur—it happens automatically.

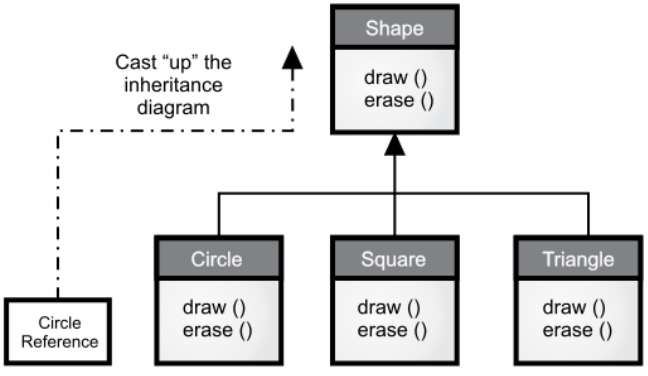
Why would you declare a method `final`? As noted in the last chapter, it prevents anyone from overriding that method. Perhaps more important, it effectively "turns off" dynamic binding, or rather it tells the compiler that dynamic binding isn't necessary. This allows the compiler to generate slightly more efficient code for `final` method calls. However, in most cases it won't make any overall performance difference in your program, so it's best to only use `final` as a design decision, and not as an attempt to improve performance.

Producing the Right Behavior

Once you know that all method binding in Java happens polymorphically via late binding, you can write your code to talk to the base class and know that all the derived-class cases will work correctly using the same code. Or to put it another way, you "send a message to an object and let the object figure out the right thing to do."

The classic object-oriented example uses shapes. This example is easy to visualize, but unfortunately it can confuse novice programmers into thinking that OOP is just for graphics programming, which is not the case.

The shape example has a base class called `Shape` and various derived types: `Circle`, `Square`, `Triangle`, etc. The reason the example works so well is it's easy to say, "A circle is a type of shape" and be understood. The inheritance diagram shows the relationships:



The upcast could occur in a statement as simple as:

```
Shape s = new Circle();
```

This creates a `Circle` object, and the resulting reference is immediately assigned to a `Shape`, which would seem to be an error (assigning one type to another); and yet it's fine because a `Circle` *is* a `Shape` by inheritance. So the compiler agrees with the statement and doesn't issue an error message.

Suppose you call one of the base-class methods (those overridden in the derived classes):

```
s.draw();
```

Again, you might expect that `Shape`'s `draw()` is called because this is, after all, a `Shape` reference—how could the compiler know to do anything else? And yet the proper `Circle.draw()` is called because of late binding (polymorphism).

The following example puts it a slightly different way. First, let's create a reusable library of `Shape` types. The base class `Shape` establishes the common interface to anything inherited from `Shape`—all shapes can be drawn and erased:

```
// polymorphism/shape/Shape.java
package polymorphism.shape;

public class Shape {
  public void draw() {}
  public void erase() {}
}
```

The derived classes override these definitions to provide unique behavior for each specific type of shape:

```
// polymorphism/shape/Circle.java
package polymorphism.shape;

public class Circle extends Shape {
  @Override public void draw() {
    System.out.println("Circle.draw()");
  }
  @Override public void erase() {
    System.out.println("Circle.erase()");
  }
}
```

```
// polymorphism/shape/Square.java
package polymorphism.shape;

public class Square extends Shape {
  @Override public void draw() {
    System.out.println("Square.draw()");
  }
  @Override public void erase() {
    System.out.println("Square.erase()");
  }
}
```

```
// polymorphism/shape/Triangle.java
package polymorphism.shape;

public class Triangle extends Shape {
  @Override public void draw() {
    System.out.println("Triangle.draw()");
  }
  @Override public void erase() {
    System.out.println("Triangle.erase()");
  }
}
```

RandomShapes is a kind of "factory" that produces a reference to a randomly created Shape object each time you call its `get()` method. Note that the upcasting happens in the `return` statements, each of which takes a reference to a `Circle`, `Square`, or `Triangle` and sends it out of `get()` as the return type, Shape. So whenever you call `get()`, you never get a chance to see what specific type it is because you always get back a plain Shape reference:

```
// polymorphism/shape/RandomShapes.java
// A "factory" that randomly creates shapes
package polymorphism.shape;
import java.util.*;

public class RandomShapes {
  private Random rand = new Random(47);
  public Shape get() {
    switch(rand.nextInt(3)) {
      default:
      case 0: return new Circle();
      case 1: return new Square();
      case 2: return new Triangle();
    }
  }
  public Shape[] array(int sz) {
    Shape[] shapes = new Shape[sz];
    // Fill up the array with shapes:
    for(int i = 0; i < shapes.length; i++)
      shapes[i] = get();
    return shapes;
  }
}
```

The `array()` method allocates and fills an array of `Shape` objects, and is used here in the *for-in* expression:

```
// polymorphism/Shapes.java
// Polymorphism in Java
import polymorphism.shape.*;

public class Shapes {
  public static void main(String[] args) {
    RandomShapes gen = new RandomShapes();
    // Make polymorphic method calls:
    for(Shape shape : gen.array(9))
      shape.draw();
  }
}
/* Output:
Triangle.draw()
Triangle.draw()
```

```
Square.draw()
Triangle.draw()
Square.draw()
Triangle.draw()
Square.draw()
Triangle.draw()
Circle.draw()
*/
```

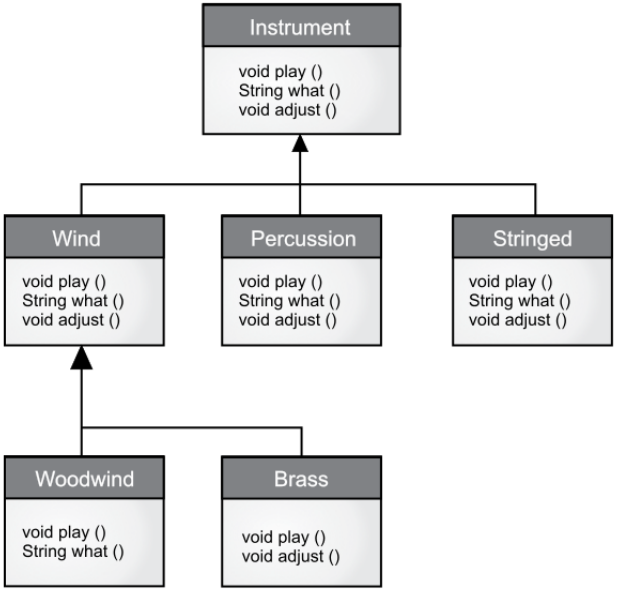
`main()` steps an array of `Shape` references produced (via the call to `array()`) by calls to `RandomShapes.get()`. Now you know you have `Shapes`, but you don't know anything more specific (and neither does the compiler). However, when you step through this array and call `draw()` for each one, the correct type-specific behavior magically occurs, as you see from the output when you run the program.

The point of creating the shapes randomly is to drive home the understanding that the compiler can have no special knowledge that allows it to make the correct calls at compile time. All the calls to `draw()` must be made through dynamic binding.

Extensibility

Now let's return to the musical instrument example. Because of polymorphism, you can add as many new types as you want to the system without changing the `tune()` method. In a well-designed OOP program, many of your methods will follow the model of `tune()` and communicate only with the base-class interface. Such a program is *extensible* because you can add new functionality by inheriting new data types from the common base class. The methods that manipulate the base-class interface will not change to accommodate the new classes.

Consider what happens if you take the instrument example and add more methods in the base class, along with several new classes:



All these new classes work correctly with the old, unchanged `tune()` method. Even if `tune()` is in a separate file and new methods are added to the interface of `Instrument`, `tune()` will still work correctly, even without recompiling it. Here is the implementation of the diagram:

```
// polymorphism/music3/Music3.java
// An extensible program
// {java polymorphism.music3.Music3}
package polymorphism.music3;
import polymorphism.music.Note;

class Instrument {
  void play(Note n) {
    System.out.println("Instrument.play() " + n);
  }
  String what() { return "Instrument"; }
  void adjust() {
    System.out.println("Adjusting Instrument");
  }
}

class Wind extends Instrument {
```

```
  @Override void play(Note n) {
    System.out.println("Wind.play() " + n);
  }
  @Override String what() { return "Wind"; }
  @Override void adjust() {
    System.out.println("Adjusting Wind");
  }
}

class Percussion extends Instrument {
  @Override void play(Note n) {
    System.out.println("Percussion.play() " + n);
  }
  @Override String what() { return "Percussion"; }
  @Override void adjust() {
    System.out.println("Adjusting Percussion");
  }
}

class Stringed extends Instrument {
  @Override void play(Note n) {
    System.out.println("Stringed.play() " + n);
  }
  @Override String what() { return "Stringed"; }
  @Override void adjust() {
    System.out.println("Adjusting Stringed");
  }
}

class Brass extends Wind {
  @Override void play(Note n) {
    System.out.println("Brass.play() " + n);
  }
  @Override void adjust() {
    System.out.println("Adjusting Brass");
  }
}

class Woodwind extends Wind {
  @Override void play(Note n) {
    System.out.println("Woodwind.play() " + n);
  }
```

```
  @Override String what() { return "Woodwind"; }
}

public class Music3 {
  // Doesn't care about type, so new types
  // added to the system still work right:
  public static void tune(Instrument i) {
    // ...
    i.play(Note.MIDDLE_C);
  }
  public static void tuneAll(Instrument[] e) {
    for(Instrument i : e)
      tune(i);
  }
  public static void main(String[] args) {
    // Upcasting during addition to the array:
    Instrument[] orchestra = {
      new Wind(),
      new Percussion(),
      new Stringed(),
      new Brass(),
      new Woodwind()
    };
    tuneAll(orchestra);
  }
}
/* Output:
Wind.play() MIDDLE_C
Percussion.play() MIDDLE_C
Stringed.play() MIDDLE_C
Brass.play() MIDDLE_C
Woodwind.play() MIDDLE_C
*/
```

The new methods are `what()`, which returns a `String` reference with a description of the class, and `adjust()`, which provides some way to adjust each instrument.

In `main()`, when you place something inside the `orchestra` array, you automatically upcast to `Instrument`.

The `tune()` method is blissfully ignorant of all the code changes that happen around it, and yet it works correctly. This is exactly what polymorphism is supposed to

provide. Changes in your code don't cause damage to parts of the program that should not be affected. Put another way, polymorphism is an important technique for the programmer to "separate the things that change from the things that stay the same."

Pitfall: "Overriding" `private` Methods

Here's something you might innocently try to do:

```
// polymorphism/PrivateOverride.java
// Trying to override a private method
// {java polymorphism.PrivateOverride}
package polymorphism;

public class PrivateOverride {
  private void f() {
    System.out.println("private f()");
  }
  public static void main(String[] args) {
    PrivateOverride po = new Derived();
    po.f();
  }
}

class Derived extends PrivateOverride {
  public void f() { System.out.println("public f()"); }
}
/* Output:
private f()
*/
```

You might reasonably expect the output to be "`public f()`", but a `private` method is automatically `final`, and is also hidden from the derived class. So `Derived`'s `f()` here is a brand new method; it's not even overloaded because the base-class version of `f()` isn't visible in `Derived`.

The result of this is that only non-`private` methods can be overridden, but watch out for the *appearance* of overriding `private` methods, which generates no compiler warnings, but doesn't do what you might expect. To be clear, use a different name from a `private` base-class method in your derived class.

If you use the `@Override` annotation, the problem is detected:

```
// polymorphism/PrivateOverride2.java
// Detecting a mistaken override using @Override
// {WillNotCompile}
package polymorphism;

public class PrivateOverride2 {
  private void f() {
    System.out.println("private f()");
  }
  public static void main(String[] args) {
    PrivateOverride2 po = new Derived2();
    po.f();
  }
}

class Derived2 extends PrivateOverride2 {
  @Override public void f() {
    System.out.println("public f()");
  }
}
```

The compiler message is:

```
error: method does not override or
implement a method from a supertype
```

Pitfall: Fields and `static` Methods

Once you learn about polymorphism, you can begin to think everything happens polymorphically. However, only ordinary method calls can be polymorphic. For example, if you access a field directly, that access is resolved at compile time:

```
// polymorphism/FieldAccess.java
// Direct field access is determined at compile time

class Super {
  public int field = 0;
  public int getField() { return field; }
}

class Sub extends Super {
  public int field = 1;
  @Override public int getField() { return field; }
  public int getSuperField() { return super.field; }
}

public class FieldAccess {
  public static void main(String[] args) {
    Super sup = new Sub(); // Upcast
    System.out.println("sup.field = " + sup.field +
      ", sup.getField() = " + sup.getField());
    Sub sub = new Sub();
    System.out.println("sub.field = " +
      sub.field + ", sub.getField() = " +
      sub.getField() +
      ", sub.getSuperField() = " +
      sub.getSuperField());
  }
}
/* Output:
sup.field = 0, sup.getField() = 1
sub.field = 1, sub.getField() = 1, sub.getSuperField()
= 0
*/
```

When a `Sub` object is upcast to a `Super` reference, any field accesses are resolved by the compiler, and are thus not polymorphic. In this example, different storage is allocated for `Super.field` and `Sub.field`. Thus, `Sub` actually contains two fields called `field`: its own and the one it gets from `Super`. However, the `Super` version is not the default that is produced when you refer to `field` in `Sub`. To get the `Super` `field` you must explicitly say `super.field`.

Although this seems like it could be a confusing issue, in practice it virtually never comes up. For one thing, you'll generally make all fields `private` and so you won't

access them directly, but only as side effects of calling methods. In addition, you probably won't give the same name to a base-class field and a derived-class field, because it is confusing.

If a method is `static`, it doesn't behave polymorphically:

```
// polymorphism/StaticPolymorphism.java
// Static methods are not polymorphic

class StaticSuper {
  public static String staticGet() {
    return "Base staticGet()";
  }
  public String dynamicGet() {
    return "Base dynamicGet()";
  }
}

class StaticSub extends StaticSuper {
  public static String staticGet() {
    return "Derived staticGet()";
  }
  @Override public String dynamicGet() {
    return "Derived dynamicGet()";
  }
}

public class StaticPolymorphism {
  public static void main(String[] args) {
    StaticSuper sup = new StaticSub(); // Upcast
    System.out.println(StaticSuper.staticGet());
    System.out.println(sup.dynamicGet());
  }
}
/* Output:
Base staticGet()
Derived dynamicGet()
*/
```

`static` methods are associated with the class, and not the individual objects.

Constructors and Polymorphism

As usual, constructors are different from other kinds of methods. This is also true when polymorphism is involved. Even though constructors are not polymorphic (they're actually `static` methods, but the `static` declaration is implicit), it's important to understand the way constructors work in complex hierarchies and with polymorphism. This understanding will help you avoid unpleasant entanglements.

Order of Constructor Calls

The order of constructor calls was briefly discussed in the Housekeeping chapter and again in the Reuse chapter, but that was before polymorphism was introduced.

A constructor for the base class is always called during the construction process for a derived class. The initialization automatically moves up the inheritance hierarchy so a constructor for every base class is called. This makes sense because the constructor has a special job: to see that the object is built properly. Fields are usually `private` so you must generally assume that a derived class has access to its own members only, and not to those of the base class. Only the base-class constructor has the proper knowledge and access to initialize its own elements. Therefore, it's essential that all constructors get called; otherwise, the entire object wouldn't be constructed. That's why the compiler enforces a constructor call for every portion of a derived class. It will silently call the zero-argument constructor if you don't explicitly call a base-class constructor in the derived-class constructor body. If there is no zero-argument constructor, the compiler will complain. (In the case where a class has no constructors, the compiler will automatically synthesize a zero-argument constructor.)

This example shows the effects of composition, inheritance, and polymorphism on the order of construction:

```
// polymorphism/Sandwich.java
// Order of constructor calls
// {java polymorphism.Sandwich}
package polymorphism;

class Meal {
  Meal() { System.out.println("Meal()"); }
}

class Bread {
  Bread() { System.out.println("Bread()"); }
}

class Cheese {
  Cheese() { System.out.println("Cheese()"); }
}

class Lettuce {
  Lettuce() { System.out.println("Lettuce()"); }
}

class Lunch extends Meal {
  Lunch() { System.out.println("Lunch()"); }
}

class PortableLunch extends Lunch {
  PortableLunch() {
    System.out.println("PortableLunch()");
  }
}

public class Sandwich extends PortableLunch {
  private Bread b = new Bread();
  private Cheese c = new Cheese();
  private Lettuce l = new Lettuce();
  public Sandwich() {
    System.out.println("Sandwich()");
  }
  public static void main(String[] args) {
    new Sandwich();
  }
}
```

```
/* Output:
Meal()
Lunch()
PortableLunch()
Bread()
Cheese()
Lettuce()
Sandwich()
*/
```

This example creates a complex class out of other classes. Each class has a constructor that announces itself. The important class is `Sandwich`, which reflects three levels of inheritance (four, if you count the implicit inheritance from `Object`) and three member objects.

The output for creating a `Sandwich` object shows that the order of constructor calls for a complex object is as follows:

1. The base-class constructor is called. This step is repeated recursively such that the root of the hierarchy is constructed first, followed by the next-derived class, etc., until the most-derived class is reached.
2. Member initializers are called in the order of declaration.
3. The body of the derived-class constructor is called.

The order of the constructor calls is important. When you inherit, you know all about the base class and can access any `public` and `protected` members of the base class. This means you can assume that all members of the base class are valid when you're in the derived class. In a normal method, construction has already taken place, so all the members of all parts of the object are also constructed.

Inside the constructor you must be certain all members are already built. The only way to guarantee this is for the base-class constructor to be called first. Then, when you're in the derived-class constructor, all the members you can access in the base class have been initialized. Knowing that all members are valid inside the constructor is also the reason that, whenever possible, you should initialize all member objects (that is, objects placed in the class using composition) at their point of definition in the class (e.g., `b`, `c`, and `l` in the preceding example). If you follow this practice, you help ensure that all base-class members *and* member objects of the current object have been initialized.

Unfortunately, this doesn't handle every case, as you will see in the next section.

Inheritance and Cleanup

When using composition and inheritance to create a new class, most of the time you won't worry about cleanup. Subobjects can usually be left to the garbage collector. If you do have cleanup issues, you must be diligent and create a `dispose()` method (the name I have chosen to use here; you might come up with something better) for your new class. And with inheritance, you must override `dispose()` in the derived class for any special cleanup that must happen as part of garbage collection. When you override `dispose()` in an inherited class, it's important to remember to call the base-class version of `dispose()` because otherwise the base-class cleanup will not happen:

```
// polymorphism/Frog.java
// Cleanup and inheritance
// {java polymorphism.Frog}
package polymorphism;

class Characteristic {
  private String s;
  Characteristic(String s) {
    this.s = s;
    System.out.println("Creating Characteristic " + s);
  }
  protected void dispose() {
    System.out.println("disposing Characteristic " + s);
  }
}

class Description {
  private String s;
  Description(String s) {
    this.s = s;
    System.out.println("Creating Description " + s);
  }
  protected void dispose() {
    System.out.println("disposing Description " + s);
  }
}

class LivingCreature {
```

```
  private Characteristic p =
    new Characteristic("is alive");
  private Description t =
    new Description("Basic Living Creature");
  LivingCreature() {
    System.out.println("LivingCreature()");
  }
  protected void dispose() {
    System.out.println("LivingCreature dispose");
    t.dispose();
    p.dispose();
  }
}

class Animal extends LivingCreature {
  private Characteristic p =
    new Characteristic("has heart");
  private Description t =
    new Description("Animal not Vegetable");
  Animal() { System.out.println("Animal()"); }
  @Override protected void dispose() {
    System.out.println("Animal dispose");
    t.dispose();
    p.dispose();
    super.dispose();
  }
}

class Amphibian extends Animal {
  private Characteristic p =
    new Characteristic("can live in water");
  private Description t =
    new Description("Both water and land");
  Amphibian() {
    System.out.println("Amphibian()");
  }
  @Override protected void dispose() {
    System.out.println("Amphibian dispose");
    t.dispose();
    p.dispose();
    super.dispose();
  }
```

```
}

public class Frog extends Amphibian {
  private Characteristic p =
    new Characteristic("Croaks");
  private Description t = new Description("Eats Bugs");
  public Frog() { System.out.println("Frog()"); }
  @Override protected void dispose() {
    System.out.println("Frog dispose");
    t.dispose();
    p.dispose();
    super.dispose();
  }
  public static void main(String[] args) {
    Frog frog = new Frog();
    System.out.println("Bye!");
    frog.dispose();
  }
}
/* Output:
Creating Characteristic is alive
Creating Description Basic Living Creature
LivingCreature()
Creating Characteristic has heart
Creating Description Animal not Vegetable
Animal()
Creating Characteristic can live in water
Creating Description Both water and land
Amphibian()
Creating Characteristic Croaks
Creating Description Eats Bugs
Frog()
Bye!
Frog dispose
disposing Description Eats Bugs
disposing Characteristic Croaks
Amphibian dispose
disposing Description Both water and land
disposing Characteristic can live in water
Animal dispose
disposing Description Animal not Vegetable
disposing Characteristic has heart
```

```
LivingCreature dispose
disposing Description Basic Living Creature
disposing Characteristic is alive
*/
```

Each class in the hierarchy contains member objects of types `Characteristic` and `Description`, which must also be disposed. The order of disposal should be the reverse of the order of initialization, in case one subobject is dependent on another. For fields, this means reverse order of declaration (because fields are initialized in declaration order). For base classes (following the form that's used in C++ for destructors), perform the derived-class cleanup first, then the base-class cleanup. That's because the derived-class cleanup could call some methods in the base class that require the base-class components to be alive, so you cannot destroy them prematurely. The output shows that all parts of the `Frog` object are disposed in reverse order of creation.

Although you don't always perform cleanup, when you do, the process requires care and awareness.

A `Frog` object "owns" its member objects. It creates them, and it knows they should live as long as the `Frog` does, so it knows when to `dispose()` of the member objects. However, if one of these member objects is shared with one or more other objects, the problem becomes more complex and you cannot simply call `dispose()`. Here, *reference counting* might be necessary to keep track of the number of objects still accessing a shared object. Here's what it looks like:

```
// polymorphism/ReferenceCounting.java
// Cleaning up shared member objects

class Shared {
  private int refcount = 0;
  private static long counter = 0;
  private final long id = counter++;
  Shared() {
    System.out.println("Creating " + this);
  }
  public void addRef() { refcount++; }
  protected void dispose() {
    if(--refcount == 0)
      System.out.println("Disposing " + this);
```

```
  }
  @Override public String toString() {
    return "Shared " + id;
  }
}

class Composing {
  private Shared shared;
  private static long counter = 0;
  private final long id = counter++;
  Composing(Shared shared) {
    System.out.println("Creating " + this);
    this.shared = shared;
    this.shared.addRef();
  }
  protected void dispose() {
    System.out.println("disposing " + this);
    shared.dispose();
  }
  @Override public String toString() {
    return "Composing " + id;
  }
}

public class ReferenceCounting {
  public static void main(String[] args) {
    Shared shared = new Shared();
    Composing[] composing = {
      new Composing(shared),
      new Composing(shared),
      new Composing(shared),
      new Composing(shared),
      new Composing(shared)
    };
    for(Composing c : composing)
      c.dispose();
  }
}
/* Output:
Creating Shared 0
Creating Composing 0
Creating Composing 1
```

```
Creating Composing 2
Creating Composing 3
Creating Composing 4
disposing Composing 0
disposing Composing 1
disposing Composing 2
disposing Composing 3
disposing Composing 4
Disposing Shared 0
*/
```

The `static long counter` keeps track of the number of instances of `Shared` that are created, and it also provides a value for `id`. The type of `counter` is `long` rather than `int`, to prevent overflow (this is just good practice; overflowing such a counter is not likely to happen in any of the examples in this book). The `id` is `final` because it should not change its value once initialized.

When you attach a shared object to your class, you must remember to call `addRef()`, but the `dispose()` method will keep track of the reference count and decide when to actually perform the cleanup. This technique requires extra diligence to use, but if you are sharing objects that require cleanup you don't have much choice.

Behavior of Polymorphic Methods Inside Constructors

The hierarchy of constructor calls brings up a dilemma. What happens if you're inside a constructor and you call a dynamically bound method of the object that's being constructed?

Inside an ordinary method, the dynamically bound call is resolved at run time, because the object cannot know whether it belongs to the class that the method is in or some class derived from it.

If you call a dynamically bound method inside a constructor, the overridden definition for that method is also used. However, the effect of this call can be rather unexpected because the overridden method is called before the object is fully constructed. This can conceal some difficult-to-find bugs.

Conceptually, the constructor's job is to bring the object into existence (hardly an ordinary feat). Inside any constructor, the entire object might be only partially

formed—you can only know that the base-class objects are initialized. If the constructor is only one step in building an object of a class that's been derived from that constructor's class, the derived parts have not yet been initialized at the time the current constructor is called. A dynamically bound method call, however, reaches "outward" into the inheritance hierarchy. It calls a method in a derived class. If you do this inside a constructor, you can call a method that might manipulate members that haven't been initialized yet—a sure recipe for disaster.

Here's the problem:

```
// polymorphism/PolyConstructors.java
// Constructors and polymorphism
// don't produce what you might expect

class Glyph {
  void draw() { System.out.println("Glyph.draw()"); }
  Glyph() {
    System.out.println("Glyph() before draw()");
    draw();
    System.out.println("Glyph() after draw()");
  }
}

class RoundGlyph extends Glyph {
  private int radius = 1;
  RoundGlyph(int r) {
    radius = r;
    System.out.println(
      "RoundGlyph.RoundGlyph(), radius = " + radius);
  }
  @Override void draw() {
    System.out.println(
      "RoundGlyph.draw(), radius = " + radius);
  }
}

public class PolyConstructors {
  public static void main(String[] args) {
    new RoundGlyph(5);
  }
}
/* Output:
```

```
Glyph() before draw()
RoundGlyph.draw(), radius = 0
Glyph() after draw()
RoundGlyph.RoundGlyph(), radius = 5
*/
```

`Glyph.draw()` is designed for overriding, which happens in `RoundGlyph`. But the `Glyph` constructor calls this method, and the call ends up in `RoundGlyph.draw()`, which would seem to be the intent. The output shows that when `Glyph`'s constructor calls `draw()`, the value of `radius` isn't even the default initial value `1`. It's `0`. This would probably result in either a dot or nothing at all drawn on the screen, and you'd be left staring, trying to figure out why the program won't work.

The order of initialization described in the earlier section isn't quite complete, and that's the key to solving the mystery. The actual process of initialization is:

1. The storage allocated for the object is initialized to binary zero before anything else happens.
2. The base-class constructors are called as described previously. Now the overridden `draw()` method is called (yes, *before* the `RoundGlyph` constructor is called), which discovers a `radius` value of zero, due to Step 1.
3. Member initializers are called in the order of declaration.
4. The body of the derived-class constructor is called.

There's an upside to this: Everything is at least initialized to zero (or whatever zero means for that particular data type) and not just left as garbage. This includes object references embedded inside a class via composition, which become `null`. So if you forget to initialize that reference, you'll get an exception at run time. Everything else gets zero, usually a telltale value when you are looking at output.

On the other hand, you should be horrified at the outcome of this program. You've done a perfectly logical thing, and yet the behavior is mysteriously wrong, with no complaints from the compiler. (C++ produces more rational behavior in this situation.) Bugs like this can easily be buried and take a long time to discover.

As a result, a good guideline for constructors is "Do as little as possible to set the object into a good state, and if you can possibly avoid it, don't call any other methods in this class." The only safe methods to call inside a constructor are those that are `final` in the base class. (This also applies to `private` methods, which are

automatically `final`.) These cannot be overridden and thus cannot produce this kind of surprise. You might not always follow this guideline, but it's something to strive towards.

Covariant Return Types

Java 5 added *covariant return types*, which means an overridden method in a derived class can return a type *derived from* the type returned by the base-class method:

```
// polymorphism/CovariantReturn.java

class Grain {
  @Override public String toString() {
    return "Grain";
  }
}

class Wheat extends Grain {
  @Override public String toString() {
    return "Wheat";
  }
}

class Mill {
  Grain process() { return new Grain(); }
}

class WheatMill extends Mill {
  @Override Wheat process() {
    return new Wheat();
  }
}

public class CovariantReturn {
  public static void main(String[] args) {
    Mill m = new Mill();
    Grain g = m.process();
    System.out.println(g);
    m = new WheatMill();
    g = m.process();
```

```
    System.out.println(g);
  }
}
/* Output:
Grain
Wheat
*/
```

The key difference here is that pre-Java-5 versions forced the overridden version of `process()` to return `Grain`, rather than `Wheat`, even though `Wheat` is derived from `Grain` and thus is still a legitimate return type. Covariant return types allow the more specific `Wheat` return type.

Designing with Inheritance

Once you learn polymorphism, it can seem that everything ought to be inherited, because polymorphism is such a clever tool. This can burden your designs. In fact, if you choose inheritance first when you're using an existing class to make a new class, things can become needlessly complicated.

A better approach is to choose composition first, especially when it's not obvious which approach to use. Composition does not force a design into an inheritance hierarchy. Composition is also more flexible because it's possible to dynamically choose a type (and thus behavior) when using composition, whereas inheritance requires that an exact type be known at compile time. The following example illustrates this:

```
// polymorphism/Transmogrify.java
// Dynamically changing the behavior of an object
// via composition (the "State" design pattern)

class Actor {
  public void act() {}
}

class HappyActor extends Actor {
  @Override public void act() {
    System.out.println("HappyActor");
```

```
  }
}

class SadActor extends Actor {
  @Override public void act() {
    System.out.println("SadActor");
  }
}

class Stage {
  private Actor actor = new HappyActor();
  public void change() { actor = new SadActor(); }
  public void performPlay() { actor.act(); }
}

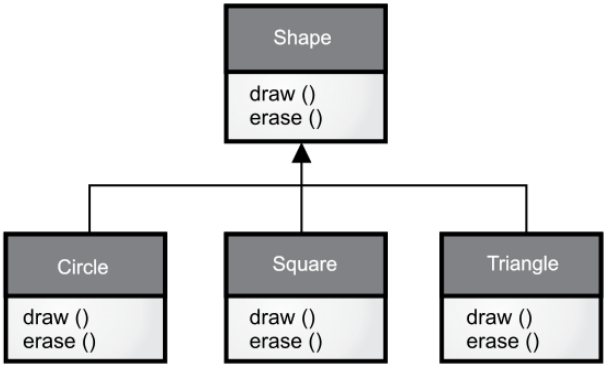
public class Transmogrify {
  public static void main(String[] args) {
    Stage stage = new Stage();
    stage.performPlay();
    stage.change();
    stage.performPlay();
  }
}
/* Output:
HappyActor
SadActor
*/
```

A `Stage` object contains a reference to an `Actor`, which is initialized to a `HappyActor` object. This means `performPlay()` produces a particular behavior. Because a reference can be re-bound to a different object at run time, a reference for a `SadActor` object can be substituted in `actor`, then the behavior produced by `performPlay()` changes. Thus you gain dynamic flexibility at run time. (This is also called the *State pattern*). In contrast, you can't decide to inherit differently at run time; that must be completely determined at compile time.

A general guideline is "Use inheritance to express differences in behavior, and fields to express variations in state." In the preceding example, both are used; two different classes are inherited to express the difference in the `act()` method, and `Stage` uses composition to allow its state to be changed. Here, that change in state happens to produce a change in behavior.

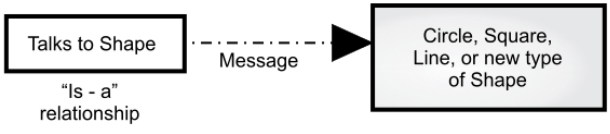
Substitution vs. Extension

It would seem that the cleanest way to create an inheritance hierarchy is to take the "pure" approach. That is, only methods from the base class are overridden in the derived class, as seen in this diagram:



This can be called a pure "is-a" relationship because the interface of a class establishes what it is. Inheritance guarantees that any derived class has the interface of the base class and nothing less. If you follow this diagram, derived classes will also have *no more* than the base-class interface.

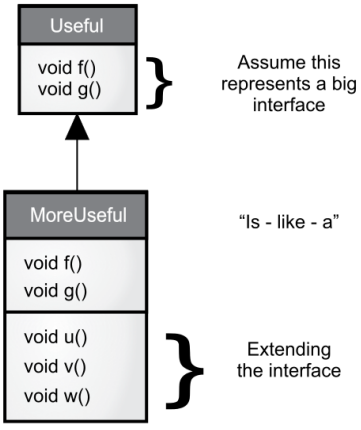
This *pure substitution* means derived class objects can be perfectly substituted for the base class, and you don't know any extra information about the subclasses when you're using them:



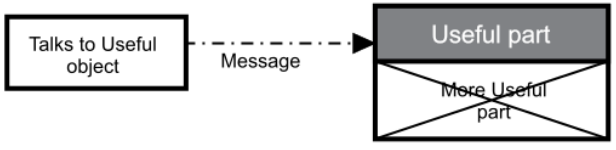
That is, the base class can receive any message you can send to the derived class because the two have exactly the same interface. All you do is upcast from the derived class and never look back to see what exact type of object you're dealing with. Everything is handled through polymorphism.

When you see it this way, it seems like a pure is-a relationship is the only sensible way to do things, and any other design indicates muddled thinking and is by definition

broken. This too is a trap. As soon as you start thinking this way, you'll turn around and discover that extending the interface (which, unfortunately, the keyword `extends` seems to encourage) is the perfect solution to a particular problem. This can be termed an "is-like-a" relationship, because the derived class is *like* the base class—it has the same fundamental interface—but it has other features that require additional methods to implement:



While this is also a useful and sensible approach (depending on the situation), it has a drawback. The extended part of the interface in the derived class is not available from the base class, so once you upcast, you can't call the new methods:



If you're not upcasting, it won't bother you, but often you'll get into a situation where you must rediscover the exact type of the object so you can access the extended methods of that type. The following section shows how this is done.

Downcasting and Reflection

Because you lose the specific type information via an *upcast* (moving up the inheritance hierarchy), it makes sense that to retrieve the type information—that is,

to move back down the inheritance hierarchy—you use a *downcast*.

You know an upcast is always safe because the base class cannot have a bigger interface than the derived class. Therefore, every message you send through the base-class interface is guaranteed to be accepted.

With a downcast, however, you don't really know that a shape (for example) is actually a circle. It could also be a triangle or square or some other type.

To solve this problem, there must be some way to guarantee that a downcast is correct, so you won't accidentally cast to the wrong type then send a message that the object can't accept. That would be unsafe.

In some languages (like C++) you must perform a special operation to get a type-safe downcast, but in Java, *every* cast is checked! So even though it looks like you're just performing an ordinary parenthesized cast, at run time this cast is checked to ensure it is in fact the type you think it is. If it isn't, you get a `ClassCastException`. This act of checking types at run time is part of Java *reflection*. Here we see basic reflection behavior:

```
// polymorphism/Reflect.java
// {ThrowsException}

class Useful {
  public void f() {}
  public void g() {}
}

class MoreUseful extends Useful {
  @Override public void f() {}
  @Override public void g() {}
  public void u() {}
  public void v() {}
  public void w() {}
}

public class Reflect {
  public static void main(String[] args) {
    Useful[] x = {
      new Useful(),
      new MoreUseful()
    };
```

```
    x[0].f();
    x[1].g();
    // Compile time: method not found in Useful:
    //- x[1].u();
    ((MoreUseful)x[1]).u(); // Downcast/Reflect
    ((MoreUseful)x[0]).u(); // Exception thrown
  }
}
/* Output:
___[ Error Output ]___
Exception in thread "main"
java.lang.ClassCastException: Useful cannot be cast to
MoreUseful
        at Reflect.main(Reflect.java:28)
*/
```

As in the previous diagram, `MoreUseful` extends the interface of `Useful`. Because it's inherited, it can also be upcast to a `Useful`. You see this happening in the initialization of the array `x` in `main()`. Both objects in the array are of class `Useful` so you can send the `f()` and `g()` methods to both, and if you try to call `u()` (which exists only in `MoreUseful`), you'll get a compile-time error message.

To access the extended interface of a `MoreUseful` object, you can try downcasting. If it's the correct type, it is successful. Otherwise, you'll get a `ClassCastException`. You don't write any special code for this exception because it indicates a programmer error that could happen anywhere in a program. The `{ThrowsException}` comment tag tells this book's build system to expect this program to throw an exception when it executes.

There's more to reflection than a simple cast. For example, there's a way to see what type you're dealing with *before* you try to downcast it. The Reflection chapter is devoted to the study of different aspects of Java reflection.

Summary

Polymorphism means "different forms." In object-oriented programming, you have the same interface from the base class, and different forms using that interface: the different versions of the dynamically bound methods.

You've seen in this chapter that it's impossible to understand, or even create, an example of polymorphism without using data abstraction and inheritance. Polymorphism is a feature that cannot be viewed in isolation (like a `switch` statement can, for example), but instead works only in concert, as part of the larger picture of class relationships.

To use polymorphism—and thus object-oriented techniques—effectively in your programs, you must expand your view of programming to include not just members and messages of an individual class, but also the commonality among classes and their relationships with each other. Although this requires significant effort, it's a worthy struggle. The results are faster program development, better code organization, extensible programs, and easier code maintenance.

Keep in mind, however, that polymorphism can be overused. Analyze your code and make sure it's actually providing a benefit.

Interfaces

> Interfaces and abstract classes provide a more structured way to separate interface from implementation.

Such mechanisms are not that common in programming languages. C++, for example, only has indirect support for these concepts. The fact that language keywords exist in Java indicates that these ideas were considered important enough to provide direct support.

First, we'll look at the *abstract class*, a kind of midway step between an ordinary class and an interface. Although your first impulse is to create an interface, the abstract class is an important and necessary tool for building classes that have fields and unimplemented methods. You can't always use a pure interface.

Abstract Classes and Methods

In all the "instrument" examples in the previous chapter, the methods in the base class `Instrument` were always "dummy" methods. If these methods are ever called, you've done something wrong. That's because the intent of `Instrument` is to create a common interface for all the classes derived from it.

In those examples, the only reason to create this common interface is so it can be expressed differently for each different subtype. It establishes a basic form, so you can say what's common for all the derived classes. Another way of saying this is to call `Instrument` an *abstract base class*, or simply an *abstract class*.

For an abstract class like `Instrument`, objects of that specific class almost always have no meaning. You create an abstract class when you want to manipulate a set of classes through its common interface. Thus, `Instrument` is meant to express only the interface, and not a particular implementation, so creating an object that is *only* an `Instrument` makes no sense, and you'll probably want to prevent the user from doing it. This can be accomplished by making all the methods in `Instrument`

generate errors, but that delays the information until run time and requires reliable exhaustive testing on the user's part. It's usually better to catch problems at compile time.

Java provides a mechanism for doing this called the *abstract method*.[76] This is a method that is incomplete; it has only a declaration and no method body. Here is the syntax for an abstract method declaration:

```
abstract void f();
```

A class containing abstract methods is called an *abstract class*. If a class contains one or more abstract methods, the class itself must be qualified as `abstract`, otherwise, the compiler produces an error message.

```
// interfaces/Basic.java

abstract class Basic {
  abstract void unimplemented();
}
```

If an abstract class is incomplete, what is Java supposed to do when someone tries to make an object of that class? It cannot safely create an object of an abstract class, so you get an error message from the compiler. This ensures the purity of the abstract class, and you don't worry about misusing it.

```
// interfaces/AttemptToUseBasic.java
// {WillNotCompile}

public class AttemptToUseBasic {
  Basic b = new Basic();
  // error: Basic is abstract; cannot be instantiated
}
```

If you inherit from an abstract class and you want to make objects of the new type, you must provide method definitions for all the abstract methods in the base class. If you don't (and you might choose not to), then the derived class is also abstract, and the compiler will force you to qualify *that* class with the `abstract` keyword.

[76] For C++ programmers, this is the analogue of C++'s *pure virtual function*.

```
// interfaces/Basic2.java

abstract class Basic2 extends Basic {
  int f() { return 111; }
  abstract void g();
  // unimplemented() still not implemented
}
```

It's possible to make a class `abstract` without including any `abstract` methods. This is useful when you've got a class where `abstract` methods don't make sense, and yet you want to prevent any instances of that class.

```
// interfaces/AbstractWithoutAbstracts.java

abstract class Basic3 {
  int f() { return 111; }
  // No abstract methods
}

public class AbstractWithoutAbstracts {
  // Basic3 b3 = new Basic3();
  // error: Basic3 is abstract; cannot be instantiated
}
```

To create an instantiable class, inherit from your `abstract` class and provide definitions for all the `abstract` methods:

```
// interfaces/Instantiable.java

abstract class Uninstantiable {
  abstract void f();
  abstract int g();
}

public class Instantiable extends Uninstantiable {
  @Override void f() { System.out.println("f()"); }
  @Override int g() { return 22; }
  public static void main(String[] args) {
    Uninstantiable ui = new Instantiable();
  }
}
```

Note the use of `@Override`. Without this annotation, if you don't provide the exact method name or signature, the `abstract` mechanism sees you haven't implemented the `abstract` method and produces a compile-time error. Thus, you could effectively argue that `@Override` is redundant here. However, `@Override` also gives the reader a signal that this method is overridden—I consider this useful, and so will use `@Override` even when the compiler would inform me of mistakes without it.

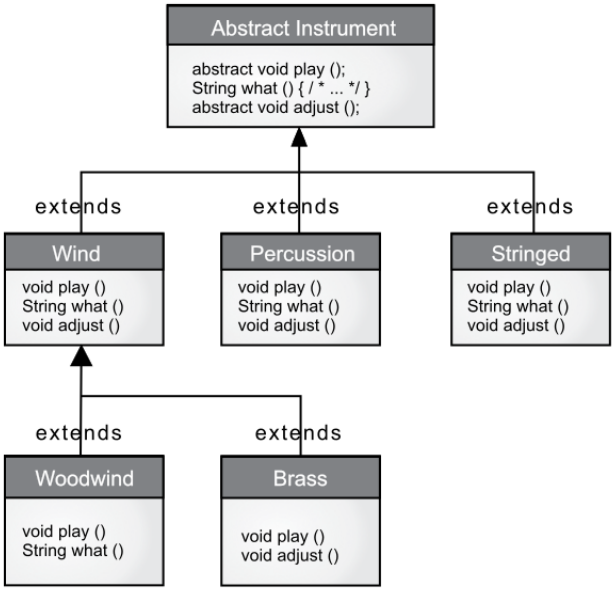
Remember that the defacto access is "friendly." You'll see shortly that an interface automatically makes its methods `public`; in fact, interfaces *only* allow `public` methods and if you don't provide an access specifier, the resulting method is *not* "friendly," but `public`. Whereas `abstract` classes allow almost everything:

```
// interfaces/AbstractAccess.java

abstract class AbstractAccess {
  private void m1() {}
  // private abstract void m1a(); // illegal
  protected void m2() {}
  protected abstract void m2a();
  void m3() {}
  abstract void m3a();
  public void m4() {}
  public abstract void m4a();
}
```

It makes sense that `private abstract` is not allowed, because you could never legally provide a definition in any subclass of `AbstractAccess`.

The `Instrument` class from the previous chapter can easily be turned into an `abstract` class. Only some of the methods are `abstract`; making a class `abstract` doesn't force you to make all the methods `abstract`. Here's what it looks like:



Here's the orchestra example modified to use `abstract` classes and methods:

```
// interfaces/music4/Music4.java
// Abstract classes and methods
// {java interfaces.music4.Music4}
package interfaces.music4;
import polymorphism.music.Note;

abstract class Instrument {
  private int i; // Storage allocated for each
  public abstract void play(Note n);
  public String what() { return "Instrument"; }
  public abstract void adjust();
}

class Wind extends Instrument {
  @Override public void play(Note n) {
    System.out.println("Wind.play() " + n);
  }
  @Override public String what() { return "Wind"; }
  @Override public void adjust() {
    System.out.println("Adjusting Wind");
  }
```

```
}

class Percussion extends Instrument {
  @Override public void play(Note n) {
    System.out.println("Percussion.play() " + n);
  }
  @Override public String what() {
    return "Percussion";
  }
  @Override public void adjust() {
    System.out.println("Adjusting Percussion");
  }
}

class Stringed extends Instrument {
  @Override public void play(Note n) {
    System.out.println("Stringed.play() " + n);
  }
  @Override public String what() { return "Stringed"; }
  @Override public void adjust() {
    System.out.println("Adjusting Stringed");
  }
}

class Brass extends Wind {
  @Override public void play(Note n) {
    System.out.println("Brass.play() " + n);
  }
  @Override public void adjust() {
    System.out.println("Adjusting Brass");
  }
}

class Woodwind extends Wind {
  @Override public void play(Note n) {
    System.out.println("Woodwind.play() " + n);
  }
  @Override public String what() { return "Woodwind"; }
}

public class Music4 {
  // Doesn't care about type, so new types
```

```
  // added to the system still work right:
  static void tune(Instrument i) {
    // ...
    i.play(Note.MIDDLE_C);
  }
  static void tuneAll(Instrument[] e) {
    for(Instrument i : e)
      tune(i);
  }
  public static void main(String[] args) {
    // Upcasting during addition to the array:
    Instrument[] orchestra = {
      new Wind(),
      new Percussion(),
      new Stringed(),
      new Brass(),
      new Woodwind()
    };
    tuneAll(orchestra);
  }
}
/* Output:
Wind.play() MIDDLE_C
Percussion.play() MIDDLE_C
Stringed.play() MIDDLE_C
Brass.play() MIDDLE_C
Woodwind.play() MIDDLE_C
*/
```

There's really no change except in `Instrument`.

It's helpful to create `abstract` classes and methods because they make the abstractness of a class explicit, and tell both the user and the compiler its intended use. Abstract classes are also useful refactoring tools because they allow you to easily move common methods up the inheritance hierarchy.

Defining Interfaces

To define an interface, use the `interface` keyword. The name "interface" is so commonly used throughout this book that, like "class," I set it in normal body font

unless I am specifically referring to the keyword `interface`.

Interfaces were easier to describe before Java 8, because they only allowed `abstract` methods. They looked like this:

```
// interfaces/PureInterface.java
// Interface only looked like this before Java 8

public interface PureInterface {
  int m1();
  void m2();
  double m3();
}
```

You don't even have to use the `abstract` keyword on the methods—because they are in an interface, Java knows they can't have method bodies (you can still add the `abstract` keyword, but then it just looks like you don't understand interfaces and you'll embarrass yourself).

So, before Java 8, we could say things like:

> The `interface` keyword creates a completely abstract class, one that doesn't represent any implementation. With an interface, you describe what a class should look like, and do, but not how it should do it. You determine method names, argument lists, and return types, but no method bodies. An interface provides only a form, but generally no implementation, although in certain restricted cases, it can.
>
> An interface says, "All classes that *implement* this particular interface will look like this." Thus, any code that uses a particular interface knows what methods might be called for that interface, and that's all. So the interface is used to establish a "protocol" between classes. (Some object-oriented programming languages have a keyword called *protocol* to do the same thing.)

With Java 8 the interface waters have been muddied a bit, because Java 8 allows both `default` methods and `static` methods—for important reasons that you'll understand as we continue through the book. The basic concept of the interface still holds, which is that they are more of a *concept* of a type, and less of an implementation.

Perhaps the most notable difference between an interface and an abstract class is the idiomatic ways the two are used. An interface typically suggests a "type of class" or an adjective, like *Runnable*, or *Serializable*, whereas an abstract class is usually part of your class hierarchy and is a "type of thing," like *String* or *ActionHero*.

To create an interface, use the `interface` keyword instead of the `class` keyword. As with a class, you can add the `public` keyword before the `interface` keyword (but only if that interface is defined in a file of the same name). If you leave off the `public` keyword, you get package access, so the interface is only usable within the same package.

An interface can also contain fields, but these are implicitly `static` and `final`.

To make a class that conforms to a particular interface (or group of interfaces), use the `implements` keyword, which says, "The interface is what it looks like, but now I'm going to say how it *works*." Other than that, it looks like inheritance.

```
// interfaces/ImplementingAnInterface.java

interface Concept { // Package access
  void idea1();
  void idea2();
}

class Implementation implements Concept {
  @Override public void idea1() {
    System.out.println("idea1");
  }
  @Override public void idea2() {
    System.out.println("idea2");
  }
}
```

You can choose to explicitly declare the methods in an interface as `public`, but they are `public` even if you don't say it. So when you `implement` an `interface`, the methods from the interface *must* be defined as `public`. Otherwise, they would default to package access, and you'd be reducing the accessibility of a method during inheritance, which is not allowed by the Java compiler.

As with `abstract` classes, `@Override` can be used when implementing an interface. It prevents accidental overloading and also shows the reader of the code which `interface` methods are being implemented.

Default Methods

Java 8 creates an additional use for the `default` keyword (formerly used only in `switch` statements and annotations). When used within an interface, `default` creates a method body that is substituted whenever the interface is implemented without defining that method.

Default methods are a bit more limited than methods on `abstract` classes, but can be very useful, as we will see in the Streams chapter.

Let's see how this works by starting with an interface:

```
// interfaces/AnInterface.java

interface AnInterface {
  void firstMethod();
  void secondMethod();
}
```

We can implement this in the usual way:

```
// interfaces/AnImplementation.java

public class AnImplementation implements AnInterface {
  @Override public void firstMethod() {
    System.out.println("firstMethod");
  }
  @Override public void secondMethod() {
    System.out.println("secondMethod");
  }
  public static void main(String[] args) {
    AnInterface i = new AnImplementation();
    i.firstMethod();
    i.secondMethod();
  }
}
/* Output:
firstMethod
secondMethod
*/
```

If we add another method `newMethod()` to `AnInterface` without an associated implementation for that method within `AnImplementation`, the compiler will issue an error:

```
AnImplementation.java:3: error: AnImplementation is
not abstract and does not override abstract method
newMethod() in AnInterface
public class AnImplementation implements AnInterface {
       ^
1 error
```

If we use the `default` keyword and provide a default definition for `newMethod()`, all existing uses of the interface can continue to work, untouched, while new code can call `newMethod()`:

```
// interfaces/InterfaceWithDefault.java

interface InterfaceWithDefault {
  void firstMethod();
  void secondMethod();
  default void newMethod() {
    System.out.println("newMethod");
  }
}
```

The `default` keyword allows method implementations within interfaces—before Java 8 this was not permitted.

```
// interfaces/Implementation2.java

public class Implementation2
implements InterfaceWithDefault {
  @Override public void firstMethod() {
    System.out.println("firstMethod");
  }
  @Override public void secondMethod() {
    System.out.println("secondMethod");
  }
  public static void main(String[] args) {
    InterfaceWithDefault i =
      new Implementation2();
```

```
    i.firstMethod();
    i.secondMethod();
    i.newMethod();
  }
}
/* Output:
firstMethod
secondMethod
newMethod
*/
```

Although `newMethod()` has no definition within `Implementation2`, it's now available.

The compelling reason to add `default` methods is that they allow you to add methods to an existing interface without breaking all the code that already uses that interface. `default` methods are sometimes also called *defender methods* or *virtual extension methods*.

Multiple Inheritance

Multiple inheritance means a class may inherit characteristics and features from more than one parent type.

When Java was first designed, multiple inheritance in C++ was roundly denigrated. Java was strictly a single-inheritance language: You could only inherit from one class (or `abstract` class). You could also implement as many interfaces as you like, but before Java 8 an interface carried no baggage—it was only a description of what its methods looked like.

Now, many years later, via `default` methods, Java has *some* multiple inheritance. Combining interfaces with `default` methods means you can combine *behaviors* from multiple base types. Because interfaces still don't allow fields (only `static` fields, which don't apply), fields can still only come from the single base class or `abstract` class; that is, you cannot have multiple inheritance of *state*. Here's what it looks like:

```
// interfaces/MultipleInheritance.java
import java.util.*;

interface One {
  default void first() { System.out.println("first"); }
}

interface Two {
  default void second() {
    System.out.println("second");
  }
}

interface Three {
  default void third() { System.out.println("third"); }
}

class MI implements One, Two, Three {}

public class MultipleInheritance {
  public static void main(String[] args) {
    MI mi = new MI();
    mi.first();
    mi.second();
    mi.third();
  }
}
/* Output:
first
second
third
*/
```

Now we can do something we never could prior to Java 8: combine implementations from multiple sources. This works fine as long as all the base-class methods have distinct names and argument lists. If not, you get compile-time errors:

```
// interfaces/MICollision.java
import java.util.*;

interface Bob1 {
  default void bob() {
    System.out.println("Bob1::bob");
  }
}

interface Bob2 {
  default void bob() {
    System.out.println("Bob2::bob");
  }
}

// class Bob implements Bob1, Bob2 {}
/* Produces:
error: class Bob inherits unrelated defaults
for bob() from types Bob1 and Bob2
class Bob implements Bob1, Bob2 {}
^
1 error
*/

interface Sam1 {
  default void sam() {
    System.out.println("Sam1::sam");
  }
}

interface Sam2 {
  default void sam(int i) {
    System.out.println(i * 2);
  }
}

// This works because the argument lists are distinct:
class Sam implements Sam1, Sam2 {}

interface Max1 {
  default void max() {
    System.out.println("Max1::max");
```

```
  }
}

interface Max2 {
  default int max() { return 47; }
}

// class Max implements Max1, Max2 {}
/* Produces:
error: types Max2 and Max1 are incompatible;
both define max(), but with unrelated return types
class Max implements Max1, Max2 {}
^
1 error
*/
```

In `Sam`, the two `sam()` methods have the same name but their *signatures* are unique—the signature includes the name and argument types, and it's what the compiler uses to distinguish one method from another. However, as `Max` shows, the return type is not part of the signature and thus cannot be used to differentiate two methods.

To fix the problem, you must override the conflicting method:

```
// interfaces/Jim.java
import java.util.*;

interface Jim1 {
  default void jim() {
    System.out.println("Jim1::jim");
  }
}

interface Jim2 {
  default void jim() {
    System.out.println("Jim2::jim");
  }
}

public class Jim implements Jim1, Jim2 {
  @Override public void jim() {
    Jim2.super.jim();
  }
```

```
  public static void main(String[] args) {
    new Jim().jim();
  }
}
/* Output:
Jim2::jim
*/
```

Of course, you can redefine `jim()` to anything you want, but you'll typically choose one of the base-class implementations using the `super` keyword, as shown.

`static` Methods in Interfaces

Java 8 also adds the ability to include `static` methods inside interfaces. This allows utilities that rightly belong in the interface, which are typically things that manipulate that interface, or are general-purpose tools:

```
// onjava/Operations.java
package onjava;
import java.util.*;

public interface Operations {
  void execute();
  static void runOps(Operations... ops) {
    for(Operations op : ops)
      op.execute();
  }
  static void show(String msg) {
    System.out.println(msg);
  }
}
```

This is a version of the *Template Method* design pattern (described in the Patterns chapter), where `runOps()` is the template method. `runOps()` uses a variable argument list so we can pass as many `Operation` arguments as we want and run them in order:

```
// interfaces/Machine.java
import java.util.*;
import onjava.Operations;

class Bing implements Operations {
  @Override public void execute() {
    Operations.show("Bing");
  }
}

class Crack implements Operations {
  @Override public void execute() {
    Operations.show("Crack");
  }
}

class Twist implements Operations {
  @Override public void execute() {
    Operations.show("Twist");
  }
}

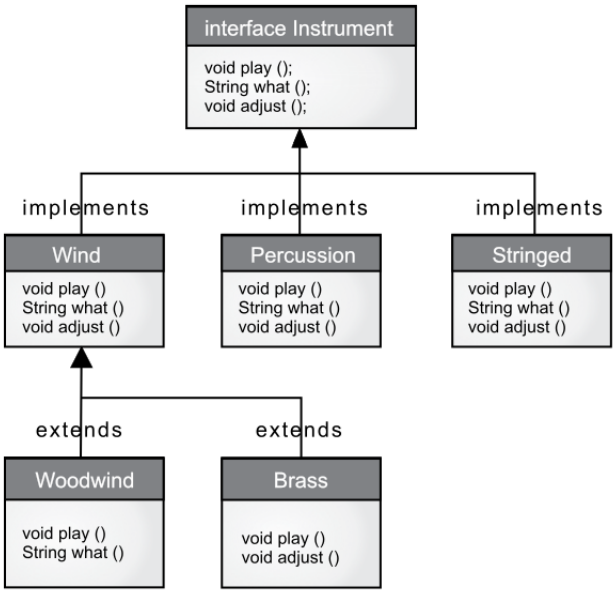
public class Machine {
  public static void main(String[] args) {
    Operations.runOps(
      new Bing(), new Crack(), new Twist());
  }
}
/* Output:
Bing
Crack
Twist
*/
```

Here you see the different ways to create `Operations`: an external class (`Bing`), an anonymous class, a method reference, and lambda expressions—which certainly appear to be the nicest solution here.

This feature is an improvement, because it allows you to keep `static` methods in more appropriate places.

Instrument as an Interface

Let's revisit the instrument example, using interfaces:



The `Woodwind` and `Brass` classes show that once you've implemented an interface, that implementation becomes an ordinary class that can be extended in the regular way.

Because of the way interfaces work, none of the methods in `Instrument` are explicitly `public`, but they're automatically `public` anyway. Both `play()` and `adjust()` have definitions using the `default` keyword. Before Java 8, both definitions had to be duplicated in each implementation, which was redundant and annoying:

```
// interfaces/music5/Music5.java
// {java interfaces.music5.Music5}
package interfaces.music5;
import polymorphism.music.Note;

interface Instrument {
  // Compile-time constant:
  int VALUE = 5; // static & final
  default void play(Note n) {  // Automatically public
    System.out.println(this + ".play() " + n);
  }
  default void adjust() {
    System.out.println("Adjusting " + this);
  }
}

class Wind implements Instrument {
  @Override public String toString() {
    return "Wind";
  }
}

class Percussion implements Instrument {
  @Override public String toString() {
    return "Percussion";
  }
}

class Stringed implements Instrument {
  @Override public String toString() {
    return "Stringed";
  }
}

class Brass extends Wind {
  @Override public String toString() {
    return "Brass";
  }
}

class Woodwind extends Wind {
  @Override public String toString() {
```

```
    return "Woodwind";
  }
}

public class Music5 {
  // Doesn't care about type, so new types
  // added to the system still work right:
  static void tune(Instrument i) {
    // ...
    i.play(Note.MIDDLE_C);
  }
  static void tuneAll(Instrument[] e) {
    for(Instrument i : e)
      tune(i);
  }
  public static void main(String[] args) {
    // Upcasting during addition to the array:
    Instrument[] orchestra = {
      new Wind(),
      new Percussion(),
      new Stringed(),
      new Brass(),
      new Woodwind()
    };
    tuneAll(orchestra);
  }
}
/* Output:
Wind.play() MIDDLE_C
Percussion.play() MIDDLE_C
Stringed.play() MIDDLE_C
Brass.play() MIDDLE_C
Woodwind.play() MIDDLE_C
*/
```

One other change was made to this version of the example: The `what()` method is changed to `toString()` because that is how the method is used. `toString()` is part of the root class `Object` so it doesn't have to appear in the interface.

Notice it doesn't matter if you are upcasting to a "regular" class called `Instrument`, an `abstract` class called `Instrument`, or to an interface called `Instrument`. The behavior is the same. In fact, the `tune()` method shows that there isn't any evidence

about whether `Instrument` is a "regular" class, an `abstract` class, or an interface.

Abstract Classes vs. Interfaces

Especially with the addition of `default` methods in Java 8, it can become somewhat confusing to know when an abstract class is the best choice, and when you should instead use an interface. This table should make the distinction clear:

Feature	Interfaces	Abstract Classes
Combinations	Can combine multiple interfaces in a new class.	Can only inherit from a single abstract class.
State	Cannot contain fields (except `static` fields, which do not support object state).	Can contain fields. Non-`abstract` methods may refer to these fields.
`default` methods & `abstract` methods	`default` methods need not be implemented in subtypes. `default` methods can only refer to other interface methods (not fields).	`abstract` methods must be implemented in subtypes.
Constructor	Cannot have a constructor.	Can have a constructor.
Visibility	Implicitly `public`.	Can be `protected` or "friendly."

An `abstract` class is still a class, and can thus be the only class inherited when creating a new class. Multiple interfaces can be implemented in the process of creating a new class.

A rule of thumb is to "be as abstract as possible—within reason." Thus, prefer interfaces over abstract classes. You'll know when you must use an abstract class. And don't use either one unless you must. Most of the time, a regular class will do the trick, and when it doesn't, you can move to an interface or abstract class.

Complete Decoupling

Whenever a method works with a class instead of an interface, you are limited to using that class or its subclasses. If you would like to apply the method to a class that isn't in that hierarchy, you're out of luck. An interface relaxes this constraint considerably. As a result, you can write more reusable code.

For example, suppose you have a `Processor` class with `name()` and `process()` methods. `process()` takes input, modifies it and produces output. This class is inherited to create different types of `Processor`. Here, the `Processor` types modify `String` objects (note that the return types can be covariant, but not the argument types):

```
// interfaces/Applicator.java
import java.util.*;

class Processor {
  public String name() {
    return getClass().getSimpleName();
  }
  public Object process(Object input) {
    return input;
  }
}

class Upcase extends Processor {
  @Override // Covariant return:
  public String process(Object input) {
    return ((String)input).toUpperCase();
  }
}

class Downcase extends Processor {
  @Override
  public String process(Object input) {
    return ((String)input).toLowerCase();
  }
}

class Splitter extends Processor {
```

```
  @Override
  public String process(Object input) {
    // split() divides a String into pieces:
    return Arrays.toString(((String)input).split(" "));
  }
}

public class Applicator {
  public static void apply(Processor p, Object s) {
    System.out.println("Using Processor " + p.name());
    System.out.println(p.process(s));
  }
  public static void main(String[] args) {
    String s =
      "We are such stuff as dreams are made on";
    apply(new Upcase(), s);
    apply(new Downcase(), s);
    apply(new Splitter(), s);
  }
}
/* Output:
Using Processor Upcase
WE ARE SUCH STUFF AS DREAMS ARE MADE ON
Using Processor Downcase
we are such stuff as dreams are made on
Using Processor Splitter
[We, are, such, stuff, as, dreams, are, made, on]
*/
```

The `Applicator.apply()` method takes any kind of `Processor`, applies it to an `Object` and prints the results. Creating a method that behaves differently depending on the argument object you pass it is called the *Strategy* design pattern. The method contains the fixed part of the algorithm, and the Strategy contains the part that varies. The Strategy is the object you pass in, containing code. Here, the `Processor` object is the Strategy, and `main()` shows three different Strategies applied to the `String` `s`.

The `split()` method is part of the `String` class. It takes the `String` object and splits it using the argument as a boundary, and returns a `String[]`. It is used here as a shorter way of creating an array of `String`.

Now suppose you discover a set of electronic filters that seem like they might fit into

your `Applicator.apply()` method:

```
// interfaces/filters/Waveform.java
package interfaces.filters;

public class Waveform {
  private static long counter;
  private final long id = counter++;
  @Override public String toString() {
    return "Waveform " + id;
  }
}


// interfaces/filters/Filter.java
package interfaces.filters;

public class Filter {
  public String name() {
    return getClass().getSimpleName();
  }
  public Waveform process(Waveform input) {
    return input;
  }
}


// interfaces/filters/LowPass.java
package interfaces.filters;

public class LowPass extends Filter {
  double cutoff;
  public LowPass(double cutoff) {
    this.cutoff = cutoff;
  }
  @Override
  public Waveform process(Waveform input) {
    return input; // Dummy processing
  }
}
```

```
// interfaces/filters/HighPass.java
package interfaces.filters;

public class HighPass extends Filter {
  double cutoff;
  public HighPass(double cutoff) {
    this.cutoff = cutoff;
  }
  @Override
  public Waveform process(Waveform input) {
    return input;
  }
}

// interfaces/filters/BandPass.java
package interfaces.filters;

public class BandPass extends Filter {
  double lowCutoff, highCutoff;
  public BandPass(double lowCut, double highCut) {
    lowCutoff = lowCut;
    highCutoff = highCut;
  }
  @Override
  public Waveform process(Waveform input) {
    return input;
  }
}
```

`Filter` has the same interface elements as `Processor`, but because it isn't inherited from `Processor`—because the creator of the `Filter` class had no clue you might want to use it as a `Processor`—you can't use a `Filter` with the `Applicator.apply()` method, even though it would work fine. Basically, the coupling between `Applicator.apply()` and `Processor` is stronger than it needs to be, and this prevents the `Applicator.apply()` code from being reused when it ought to be. Also notice that the inputs and outputs are both `Waveforms`.

If `Processor` is an interface, however, the constraints are loosened enough you can reuse an `Applicator.apply()` that takes that interface. Here are the modified versions of `Processor` and `Applicator`:

```
// interfaces/interfaceprocessor/Processor.java
package interfaces.interfaceprocessor;

public interface Processor {
  default String name() {
    return getClass().getSimpleName();
  }
  Object process(Object input);
}

// interfaces/interfaceprocessor/Applicator.java
package interfaces.interfaceprocessor;

public class Applicator {
  public static void apply(Processor p, Object s) {
    System.out.println("Using Processor " + p.name());
    System.out.println(p.process(s));
  }
}
```

The first way you can reuse code is if client programmers can write their classes to conform to the interface, like this:

```
// interfaces/interfaceprocessor/StringProcessor.java
// {java interfaces.interfaceprocessor.StringProcessor}
package interfaces.interfaceprocessor;
import java.util.*;

interface StringProcessor extends Processor {
  @Override
  String process(Object input);        // [1]
  String S =                           // [2]
  "If she weighs the same as a duck, " +
  "she's made of wood";
  static void main(String[] args) {    // [3]
    Applicator.apply(new Upcase(), S);
    Applicator.apply(new Downcase(), S);
    Applicator.apply(new Splitter(), S);
  }
}

class Upcase implements StringProcessor {
```

```
  @Override // Covariant return:
  public String process(Object input) {
    return ((String)input).toUpperCase();
  }
}

class Downcase implements StringProcessor {
  @Override
  public String process(Object input) {
    return ((String)input).toLowerCase();
  }
}

class Splitter implements StringProcessor {
  @Override
  public String process(Object input) {
    return Arrays.toString(((String)input).split(" "));
  }
}
/* Output:
Using Processor Upcase
IF SHE WEIGHS THE SAME AS A DUCK, SHE'S MADE OF WOOD
Using Processor Downcase
if she weighs the same as a duck, she's made of wood
Using Processor Splitter
[If, she, weighs, the, same, as, a, duck,, she's, made,
of, wood]
*/
```

- **[1]** This declaration is unnecessary; the compiler will not complain if you remove it. However it does notate the covariant return change from `Object` to `String`.
- **[2]** `s` is automatically `static` and `final` because it's defined inside an interface.
- **[3]** You can even define a `main()` inside an interface.

here, the approach works. However, you are often in the situation of being unable to modify the classes. In the case of the electronic filters, for example, the library was discovered rather than created. In these cases, you can use the *Adapter* design pattern. In Adapter, you write code to take the interface you have and produce the interface you need, like this:

```
// interfaces/interfaceprocessor/FilterProcessor.java
// {java interfaces.interfaceprocessor.FilterProcessor}
package interfaces.interfaceprocessor;
import interfaces.filters.*;

class FilterAdapter implements Processor {
  Filter filter;
  FilterAdapter(Filter filter) {
    this.filter = filter;
  }
  @Override
  public String name() { return filter.name(); }
  @Override
  public Waveform process(Object input) {
    return filter.process((Waveform)input);
  }
}

public class FilterProcessor {
  public static void main(String[] args) {
    Waveform w = new Waveform();
    Applicator.apply(
      new FilterAdapter(new LowPass(1.0)), w);
    Applicator.apply(
      new FilterAdapter(new HighPass(2.0)), w);
    Applicator.apply(
      new FilterAdapter(new BandPass(3.0, 4.0)), w);
  }
}
/* Output:
Using Processor LowPass
Waveform 0
Using Processor HighPass
Waveform 0
Using Processor BandPass
Waveform 0
*/
```

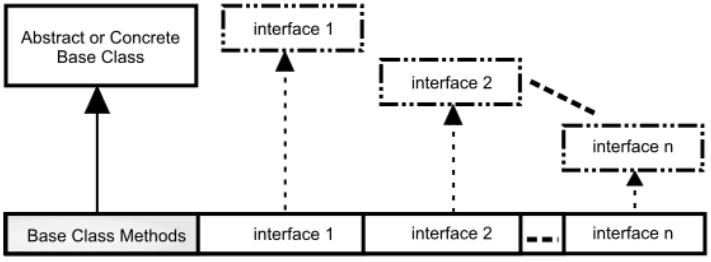
In this approach to Adapter, the `FilterAdapter` constructor takes the interface you have—`Filter`—and produces an object that has the `Processor` interface you need. You might also notice delegation in the `FilterAdapter` class.

Covariance allows us to produce a `Waveform` from `process()` rather than just an `Object`.

Decoupling interface from implementation allows an interface to be applied to multiple different implementations, and thus your code is more reusable.

Combining Multiple Interfaces

Because an interface has no implementation at all—that is, there is no storage associated with an interface—there's nothing to prevent many interfaces from being combined. This is valuable because there are times when you want to say, "An `x` is an `a` *and* a `b` *and* a `c`."



In a derived class, you aren't forced to have a base class that is either `abstract` or "concrete" (that is, one with no `abstract` methods). If you *do* inherit from a non-interface, you can inherit from only one. All the rest of the base elements must be interfaces. You place all the interface names after the `implements` keyword and separate them with commas. You can have as many interfaces as you want. You can upcast to each interface, because each interface is an independent type. The following example shows a concrete class combined with several interfaces to produce a new class:

```
// interfaces/Adventure.java
// Multiple interfaces

interface CanFight {
  void fight();
}

interface CanSwim {
  void swim();
}

interface CanFly {
  void fly();
}

class ActionCharacter {
  public void fight() {}
}

class Hero extends ActionCharacter
    implements CanFight, CanSwim, CanFly {
  @Override public void swim() {}
  @Override public void fly() {}
}

public class Adventure {
  public static void t(CanFight x) { x.fight(); }
  public static void u(CanSwim x) { x.swim(); }
  public static void v(CanFly x) { x.fly(); }
  public static void w(ActionCharacter x) { x.fight(); }
  public static void main(String[] args) {
    Hero h = new Hero();
    t(h); // Treat it as a CanFight
    u(h); // Treat it as a CanSwim
    v(h); // Treat it as a CanFly
    w(h); // Treat it as an ActionCharacter
  }
}
```

`Hero` combines the concrete class `ActionCharacter` with the interfaces `CanFight`, `CanSwim`, and `CanFly`. When you combine a concrete class with interfaces this way, the concrete class must come first, then the interfaces. (The compiler gives an error

otherwise.)

The signature for `fight()` is the same in the interface `CanFight` and the class `ActionCharacter`, and `fight()` is *not* provided with a definition in `Hero`. You can extend an interface, but then you've got another interface. When you want to create an object, all the definitions must first exist. Even though `Hero` does not explicitly provide a definition for `fight()`, the definition comes along with `ActionCharacter`; thus, it's possible to create `Hero` objects.

`Adventure` shows four methods that take arguments of the various interfaces and of the concrete class. When you create a `Hero` object, it can be passed to any of these methods, which means it is upcast to each interface in turn. Because of the way interfaces are designed in Java, this works without any particular effort on the part of the programmer.

Keep in mind that one of the core reasons for interfaces is shown in the preceding example: to upcast to more than one base type (and the flexibility this provides). However, a second reason for using interfaces is the same as using an `abstract` base class: to prevent the client programmer from making an object of this class and to establish it is only an interface.

This brings up a question: Should you use an interface or an `abstract` class? If it's possible to create your base class without any method definitions or member variables, prefer interfaces to `abstract` classes. In fact, if you know something is a base class, you can consider making it an interface (this subject is revisited in the chapter summary).

Extending an Interface with Inheritance

You can easily add new method declarations to an interface by using inheritance, and you can also combine several interfaces into a new interface with inheritance. In both cases you get a new interface, as seen in this example:

```
// interfaces/HorrorShow.java
// Extending an interface with inheritance

interface Monster {
  void menace();
}

interface DangerousMonster extends Monster {
  void destroy();
}

interface Lethal {
  void kill();
}

class DragonZilla implements DangerousMonster {
  @Override public void menace() {}
  @Override public void destroy() {}
}

interface Vampire extends DangerousMonster, Lethal {
  void drinkBlood();
}

class VeryBadVampire implements Vampire {
  @Override public void menace() {}
  @Override public void destroy() {}
  @Override public void kill() {}
  @Override public void drinkBlood() {}
}

public class HorrorShow {
  static void u(Monster b) { b.menace(); }
  static void v(DangerousMonster d) {
    d.menace();
    d.destroy();
  }
  static void w(Lethal l) { l.kill(); }
  public static void main(String[] args) {
    DangerousMonster barney = new DragonZilla();
    u(barney);
    v(barney);
```

```
    Vampire vlad = new VeryBadVampire();
    u(vlad);
    v(vlad);
    w(vlad);
  }
}
```

`DangerousMonster` is a simple extension to `Monster` that produces a new interface. This is implemented in `DragonZilla`.

The syntax used in `Vampire` works *only* when inheriting interfaces. Normally, you can use `extends` with only a single class, but `extends` can refer to multiple base interfaces when building a new interface. Notice that interface names are separated with commas.

Name Collisions When Combining Interfaces

There's a small pitfall when implementing multiple interfaces. In the preceding example, both `CanFight` and `ActionCharacter` have identical `void fight()` methods. An identical method is not a problem, but what if the method differs by signature or return type? Here's an example:

```
// interfaces/InterfaceCollision.java

interface I1 { void f(); }
interface I2 { int f(int i); }
interface I3 { int f(); }
class C { public int f() { return 1; } }

class C2 implements I1, I2 {
  @Override
  public void f() {}
  @Override
  public int f(int i) { return 1; } // Overloaded
}

class C3 extends C implements I2 {
  @Override
  public int f(int i) { return 1; } // Overloaded
}
```

```
class C4 extends C implements I3 {
  // Identical, no problem:
  @Override public int f() { return 1; }
}

// Methods differ only by return type:
//- class C5 extends C implements I1 {}
//- interface I4 extends I1, I3 {}
```

The difficulty occurs because overriding, implementation, and overloading get unpleasantly mixed together. Also, overloaded methods cannot differ only by return type. When the last two lines are uncommented, the error messages say it all:

```
error: C5 is not abstract and does not override abstract
  method f() in I1
  class C5 extends C implements I1 {}

error: types I3 and I1 are incompatible; both define f(),
 but with unrelated return types
 interface I4 extends I1, I3 {}
```

Using the same method names in different interfaces when you expect those interface to be combined generally causes confusion in the readability of the code. Strive to avoid it.

Adapting to an Interface

One of the most compelling reasons for interfaces is to allow multiple implementations for the same interface. In simple cases this is in the form of a method that accepts an interface, leaving it up to you to implement that interface and pass your object to the method.

Thus, a common use for interfaces is the aforementioned *Strategy* design pattern. You write a method that performs certain operations, and that method takes an interface you also specify. You're basically saying, "You can use my method with any object you like, as long as your object conforms to my interface." This makes your method more flexible, general and reusable.

For example, the constructor for the `Scanner` class (which you'll learn more about in the Strings chapter) takes a `Readable` interface. You'll find that `Readable` is not an argument for any other method in the Java standard library—it was created solely for `Scanner`, so `Scanner` doesn't constrain its argument to a particular class. This way, `Scanner` can be made to work with more types. If you create a new class and you want it used with `Scanner`, you make it `Readable`, like this:

```
// interfaces/RandomStrings.java
// Implementing an interface to conform to a method
import java.nio.*;
import java.util.*;

public class RandomStrings implements Readable {
  private static Random rand = new Random(47);
  private static final char[] CAPITALS =
    "ABCDEFGHIJKLMNOPQRSTUVWXYZ".toCharArray();
  private static final char[] LOWERS =
    "abcdefghijklmnopqrstuvwxyz".toCharArray();
  private static final char[] VOWELS =
    "aeiou".toCharArray();
  private int count;
  public RandomStrings(int count) {
    this.count = count;
  }
  @Override public int read(CharBuffer cb) {
    if(count-- == 0)
      return -1; // Indicates end of input
    cb.append(CAPITALS[rand.nextInt(CAPITALS.length)]);
    for(int i = 0; i < 4; i++) {
      cb.append(VOWELS[rand.nextInt(VOWELS.length)]);
      cb.append(LOWERS[rand.nextInt(LOWERS.length)]);
    }
    cb.append(" ");
    return 10; // Number of characters appended
  }
  public static void main(String[] args) {
    Scanner s = new Scanner(new RandomStrings(10));
    while(s.hasNext())
      System.out.println(s.next());
  }
}
```

```
/* Output:
Yazeruyac
Fowenucor
Goeazimom
Raeuuacio
Nuoadesiw
Hageaikux
Ruqicibui
Numasetih
Kuuuuozog
Waqizeyoy
*/
```

The `Readable` interface only requires the implementation of a `read()` method (notice how the `@Override` points out the salient method). Inside `read()`, you add to the `CharBuffer` argument (there are several ways to do this; see the `CharBuffer` documentation), or return `-1` when you have no more input.

Suppose you have a type that does not already implement `Readable`—how do you make it work with `Scanner`? Here's an example that produces random floating point numbers:

```
// interfaces/RandomDoubles.java
import java.util.*;

public interface RandomDoubles {
  Random RAND = new Random(47);
  default double next() { return RAND.nextDouble(); }
  static void main(String[] args) {
    RandomDoubles rd = new RandomDoubles() {};
    for(int i = 0; i < 7; i ++)
      System.out.print(rd.next() + " ");
  }
}
/* Output:
0.7271157860730044 0.5309454508634242
0.16020656493302599 0.18847866977771732
0.5166020801268457 0.2678662084200585
0.2613610344283964
*/
```

Again, we can use the Adapter pattern, but here the adapted class can be created by implementing both interfaces. So, using the multiple inheritance provided by the

`interface` keyword, we produce a new class which is both `RandomDoubles` and `Readable`:

```
// interfaces/AdaptedRandomDoubles.java
// Creating an adapter with inheritance
import java.nio.*;
import java.util.*;

public class AdaptedRandomDoubles
implements RandomDoubles, Readable {
  private int count;
  public AdaptedRandomDoubles(int count) {
    this.count = count;
  }
  @Override public int read(CharBuffer cb) {
    if(count-- == 0)
      return -1;
    String result = Double.toString(next()) + " ";
    cb.append(result);
    return result.length();
  }
  public static void main(String[] args) {
    Scanner s =
      new Scanner(new AdaptedRandomDoubles(7));
    while(s.hasNextDouble())
      System.out.print(s.nextDouble() + " ");
  }
}
/* Output:
0.7271157860730044 0.5309454508634242
0.16020656493302599 0.18847866977771732
0.5166020801268457 0.2678662084200585
0.2613610344283964
*/
```

Because you can add an interface onto any existing class in this way, it means that a method that takes an interface provides a way to adapt any class to work with that method. This is the power of using interfaces instead of classes.

Fields in Interfaces

Because any fields you put into an interface are automatically `static` and `final`, the interface is a convenient tool for creating groups of constant values. Before Java 5, this was the only way to produce the same effect as an `enum` in C or C++. So you will see pre-Java 5 code like this:

```
// interfaces/Months.java
// Using interfaces to create groups of constants

public interface Months {
  int
    JANUARY = 1, FEBRUARY = 2, MARCH = 3,
    APRIL = 4, MAY = 5, JUNE = 6, JULY = 7,
    AUGUST = 8, SEPTEMBER = 9, OCTOBER = 10,
    NOVEMBER = 11, DECEMBER = 12;
}
```

Notice the Java style of using all uppercase letters (with underscores to separate multiple words in a single identifier) for `static finals` that have constant initializers. The fields in an interface are automatically `public`, so that is not explicitly specified.

Since Java 5, we have the much more powerful and flexible `enum` keyword, so there are times you'll want to use that instead of interfaces to hold groups of constants. You can find more details on `enums` in the Enumerations chapter.

Initializing Fields in Interfaces

Fields defined in interfaces cannot be "blank `finals`," but they can be initialized with non-constant expressions. For example:

```
// interfaces/RandVals.java
// Initializing interface fields with
// non-constant initializers
import java.util.*;

public interface RandVals {
  Random RAND = new Random(47);
  int RANDOM_INT = RAND.nextInt(10);
  long RANDOM_LONG = RAND.nextLong() * 10;
  float RANDOM_FLOAT = RAND.nextLong() * 10;
  double RANDOM_DOUBLE = RAND.nextDouble() * 10;
}
```

The fields are `static` so they are initialized when the class is first loaded, which happens when any of the fields are accessed for the first time. Here's a simple test:

```
// interfaces/TestRandVals.java

public class TestRandVals {
  public static void main(String[] args) {
    System.out.println(RandVals.RANDOM_INT);
    System.out.println(RandVals.RANDOM_LONG);
    System.out.println(RandVals.RANDOM_FLOAT);
    System.out.println(RandVals.RANDOM_DOUBLE);
  }
}
/* Output:
8
-32032247016559954
-8.5939291E18
5.779976127815049
*/
```

The fields are not part of the interface. The values are stored in the static storage area for that interface.

Nesting Interfaces

Interfaces can be nested within classes and within other interfaces. This reveals several interesting features:

```
// interfaces/nesting/NestingInterfaces.java
// {java interfaces.nesting.NestingInterfaces}
package interfaces.nesting;

class A {
  interface B {
    void f();
  }
  public class BImp implements B {
    @Override public void f() {}
  }
  private class BImp2 implements B {
    @Override public void f() {}
  }
  public interface C {
    void f();
  }
  class CImp implements C {
    @Override public void f() {}
  }
  private class CImp2 implements C {
    @Override public void f() {}
  }
  private interface D {
    void f();
  }
  private class DImp implements D {
    @Override public void f() {}
  }
  public class DImp2 implements D {
    @Override public void f() {}
  }
  public D getD() { return new DImp2(); }
  private D dRef;
  public void receiveD(D d) {
    dRef = d;
    dRef.f();
  }
}

interface E {
  interface G {
```

```
    void f();
  }
  // Redundant "public":
  public interface H {
    void f();
  }
  void g();
  // Cannot be private within an interface:
  //- private interface I {}
}

public class NestingInterfaces {
  public class BImp implements A.B {
    @Override public void f() {}
  }
  class CImp implements A.C {
    @Override public void f() {}
  }
  // Cannot implement a private interface except
  // within that interface's defining class:
  //- class DImp implements A.D {
  //-  public void f() {}
  //- }
  class EImp implements E {
    @Override public void g() {}
  }
  class EGImp implements E.G {
    @Override public void f() {}
  }
  class EImp2 implements E {
    @Override public void g() {}
    class EG implements E.G {
      @Override public void f() {}
    }
  }
  public static void main(String[] args) {
    A a = new A();
    // Can't access A.D:
    //- A.D ad = a.getD();
    // Doesn't return anything but A.D:
    //- A.DImp2 di2 = a.getD();
    // Cannot access a member of the interface:
```

```
    //- a.getD().f();
    // Only another A can do anything with getD():
    A a2 = new A();
    a2.receiveD(a.getD());
  }
}
```

The syntax for nesting an interface within a class is reasonably obvious. Just like non-nested interfaces, these can have `public` or package-access visibility.

As an added twist, interfaces can also be `private`, as seen in `A.D` (the same qualification syntax is used for nested interfaces as for nested classes). What good is a `private` nested interface? You might guess it can only be implemented as a `private` inner class as in `DImp`, but `A.DImp2` shows it can also be implemented as a `public` class. However, `A.DImp2` can only be used as itself. You are not allowed to mention the fact that it implements the `private` interface `D`, so implementing a `private` interface is a way to force the definition of the methods in that interface without adding any type information (that is, without allowing any upcasting).

The method `getD()` produces a further quandary concerning the `private` interface: It's a `public` method that returns a reference to a `private` interface. What can you do with the return value of this method? `main()` shows several attempts to use the return value, all of which fail. The return value must be handed to an object that has permission to use it—here, another `A`, via the `receiveD()` method.

Interface `E` shows that interfaces can be nested within each other. However, the rules about interfaces—in particular, that all interface elements must be `public`—are strictly enforced here, so an interface nested within another interface is automatically `public` and cannot be made `private`.

`NestingInterfaces` shows the various ways that nested interfaces can be implemented. In particular, notice that when you implement an interface, you are not required to implement any interfaces nested within. Also, `private` interfaces cannot be implemented outside of their defining classes.

Initially, these features can seem like they are added strictly for syntactic consistency, but I generally find that once you know about a feature, you often discover places where it is useful.

Interfaces and Factories

An interface is a gateway to multiple implementations, and a typical way to produce objects that fit the interface is the *Factory Method* design pattern. Instead of calling a constructor directly, you call a creation method on a factory object which produces an implementation of the interface—this way, in theory, your code is completely isolated from the implementation of the interface, thus making it possible to transparently swap one implementation for another. Here's a demonstration showing the structure of the Factory Method:

```
// interfaces/Factories.java

interface Service {
  void method1();
  void method2();
}

interface ServiceFactory {
  Service getService();
}

class Service1 implements Service {
  Service1() {} // Package access
  @Override public void method1() {
    System.out.println("Service1 method1");
  }
  @Override public void method2() {
    System.out.println("Service1 method2");
  }
}

class Service1Factory implements ServiceFactory {
  @Override public Service getService() {
    return new Service1();
  }
}

class Service2 implements Service {
  Service2() {} // Package access
  @Override public void method1() {
```

```
    System.out.println("Service2 method1");
  }
  @Override public void method2() {
    System.out.println("Service2 method2");
  }
}

class Service2Factory implements ServiceFactory {
  @Override public Service getService() {
    return new Service2();
  }
}

public class Factories {
  public static void
  serviceConsumer(ServiceFactory fact) {
    Service s = fact.getService();
    s.method1();
    s.method2();
  }
  public static void main(String[] args) {
    serviceConsumer(new Service1Factory());
    // Services are completely interchangeable:
    serviceConsumer(new Service2Factory());
  }
}
/* Output:
Service1 method1
Service1 method2
Service2 method1
Service2 method2
*/
```

Without the Factory Method, your code must somewhere specify the exact type of `Service` created, to call the appropriate constructor.

Why would you add this extra level of indirection? One common reason is to create a framework. Suppose you are creating a system to play games; for example, to play both chess and checkers on the same board:

```
// interfaces/Games.java
// A Game framework using Factory Methods

interface Game { boolean move(); }
interface GameFactory { Game getGame(); }

class Checkers implements Game {
  private int moves = 0;
  private static final int MOVES = 3;
  @Override public boolean move() {
    System.out.println("Checkers move " + moves);
    return ++moves != MOVES;
  }
}

class CheckersFactory implements GameFactory {
  @Override
  public Game getGame() { return new Checkers(); }
}

class Chess implements Game {
  private int moves = 0;
  private static final int MOVES = 4;
  @Override public boolean move() {
    System.out.println("Chess move " + moves);
    return ++moves != MOVES;
  }
}

class ChessFactory implements GameFactory {
  @Override
  public Game getGame() { return new Chess(); }
}

public class Games {
  public static void playGame(GameFactory factory) {
    Game s = factory.getGame();
    while(s.move())
      ;
  }
  public static void main(String[] args) {
    playGame(new CheckersFactory());
```

```
    playGame(new ChessFactory());
  }
}
/* Output:
Checkers move 0
Checkers move 1
Checkers move 2
Chess move 0
Chess move 1
Chess move 2
Chess move 3
*/
```

If the `Games` class represents a complex piece of code, this approach means you can reuse that code with different types of games. You can imagine more elaborate games that can benefit from this pattern.

In the next chapter, you'll see a more elegant way to implement factories using anonymous inner classes.

Summary

It is tempting to decide that interfaces are good, and therefore always choose interfaces over concrete classes. Almost anytime you create a class, you can instead create an interface and a factory.

Many people have fallen to this temptation, creating interfaces and factories whenever it's possible. The logic seems to be that you might use a different implementation, so always add that abstraction. It has become a kind of premature design optimization.

Any abstraction should be motivated by a real need. Interfaces should be something you refactor to when necessary, rather than installing the extra level of indirection everywhere, along with the extra complexity. That extra complexity is significant, and if you make someone work through that complexity only to realize that you've added interfaces "just in case" and for no compelling reason—well, if I see such a design I begin questioning all the other designs this particular person has done.

An appropriate guideline is to *prefer classes to interfaces*. Start with classes, and if it becomes clear that interfaces are necessary, refactor. Interfaces are a great tool, but they can easily be overused.

Inner Classes

A class defined within another class is called an *inner class*.

With inner classes you can group classes that logically belong together and control the visibility of one within the other. However, it's important to understand that inner classes are distinctly different from composition.

At first, inner classes look like a simple code-hiding mechanism: You place classes inside other classes. You'll learn, however, that the inner class does more than that—it knows about and can communicate with the surrounding class—and the kind of code you can write with inner classes is more elegant and clear, although there's certainly no guarantee of this (also, Java 8 Lambdas and Method References reduce some of the need for inner classes).

Initially, inner classes can seem odd, and it will take some time to become comfortable using them in your designs. The need for inner classes isn't always obvious, but after the basic syntax and semantics of inner classes are described, the section "Why inner classes?" should begin to make clear their benefits.

The remainder of the chapter contains more detailed explorations of the syntax of inner classes. These features are provided for language completeness, but you might not use them, at least not at first. So the initial parts of this chapter might be all you need for now, and you can leave the more detailed explorations as reference material.

Creating Inner Classes

You create an inner class just as you'd expect—by placing the class definition inside a surrounding class:

```
// innerclasses/Parcel1.java
// Creating inner classes

public class Parcel1 {
  class Contents {
    private int i = 11;
    public int value() { return i; }
  }
  class Destination {
    private String label;
    Destination(String whereTo) {
      label = whereTo;
    }
    String readLabel() { return label; }
  }
  // Using inner classes looks just like
  // using any other class, within Parcel1:
  public void ship(String dest) {
    Contents c = new Contents();
    Destination d = new Destination(dest);
    System.out.println(d.readLabel());
  }
  public static void main(String[] args) {
    Parcel1 p = new Parcel1();
    p.ship("Tasmania");
  }
}
/* Output:
Tasmania
*/
```

When used inside `ship()`, these inner classes look just like ordinary classes. The only apparent difference is that the names are nested within `Parcel1`.

More typically, an outer class has a method that returns a reference to an inner class, as seen in the `to()` and `contents()` methods:

```
// innerclasses/Parcel2.java
// Returning a reference to an inner class

public class Parcel2 {
  class Contents {
    private int i = 11;
    public int value() { return i; }
  }
  class Destination {
    private String label;
    Destination(String whereTo) {
      label = whereTo;
    }
    String readLabel() { return label; }
  }
  public Destination to(String s) {
    return new Destination(s);
  }
  public Contents contents() {
    return new Contents();
  }
  public void ship(String dest) {
    Contents c = contents();
    Destination d = to(dest);
    System.out.println(d.readLabel());
  }
  public static void main(String[] args) {
    Parcel2 p = new Parcel2();
    p.ship("Tasmania");
    Parcel2 q = new Parcel2();
    // Defining references to inner classes:
    Parcel2.Contents c = q.contents();
    Parcel2.Destination d = q.to("Borneo");
  }
}
/* Output:
Tasmania
*/
```

To make an object of the inner class anywhere except from within a non-`static` method of the outer class, you must specify the type of that object as *OuterClassName.InnerClassName*, as seen in `main()`.

The Link to the Outer Class

So far, it appears that inner classes are just a name-hiding and code organization scheme, helpful but not that compelling. However, there's another twist. When you create an inner class, an object of that inner class contains an implicit link to the enclosing object that made it. Through this link, it can access the members of that enclosing object, without any special qualifications. In addition, inner classes have access rights to all the elements in the enclosing class:

```
// innerclasses/Sequence.java
// Holds a sequence of Objects

interface Selector {
  boolean end();
  Object current();
  void next();
}

public class Sequence {
  private Object[] items;
  private int next = 0;
  public Sequence(int size) {
    items = new Object[size];
  }
  public void add(Object x) {
    if(next < items.length)
      items[next++] = x;
  }
  private class SequenceSelector implements Selector {
    private int i = 0;
    @Override
    public boolean end() { return i == items.length; }
    @Override
    public Object current() { return items[i]; }
    @Override
    public void next() { if(i < items.length) i++; }
  }
  public Selector selector() {
    return new SequenceSelector();
  }
```

```
  public static void main(String[] args) {
    Sequence sequence = new Sequence(10);
    for(int i = 0; i < 10; i++)
      sequence.add(Integer.toString(i));
    Selector selector = sequence.selector();
    while(!selector.end()) {
      System.out.print(selector.current() + " ");
      selector.next();
    }
  }
}
/* Output:
0 1 2 3 4 5 6 7 8 9
*/
```

The `Sequence` is a fixed-sized array of `Object` with a class wrapped around it. You call `add()` to add a new `Object` to the end of the sequence (if there's room left). To fetch each of the objects in a `Sequence`, there's an interface called `Selector`. This is an example of the *Iterator* design pattern you shall learn more about in the Collections chapter. A `Selector` tells you whether you're at the `end()`, accesses the `current()` `Object`, and moves to the `next()` `Object` in the `Sequence`. Because `Selector` is an interface, other classes can implement the interface in their own ways, and other methods can take the interface as an argument, to create more general-purpose code.

Here, the `SequenceSelector` is a `private` class that provides `Selector` functionality. In `main()`, you see the creation of a `Sequence`, followed by the addition of a number of `String` objects. Then, a `Selector` is produced with a call to `selector()`, and this is used to move through the `Sequence` and select each item.

At first, the creation of `SequenceSelector` looks like just another inner class. But examine it closely. Note that each of the methods—`end()`, `current()`, and `next()`—refers to `items`, a reference that isn't part of `SequenceSelector`, but is instead a `private` field in the enclosing class. However, the inner class can access methods and fields from the enclosing class as if it owned them. This turns out to be very convenient, as you see in the preceding example.

So an inner class has automatic access to the members of the enclosing class. How can this happen? The inner class secretly captures a reference to the particular object of the enclosing class that was responsible for creating it. Then, when you refer

to a member of the enclosing class, that reference is used to select that member. Fortunately, the compiler takes care of all these details for you, but now you see that an object of an inner class can be created only in association with an object of the enclosing class (when, as you shall see, the inner class is non-`static`). Construction of the inner-class object requires the reference to the object of the enclosing class, and the compiler will complain if it cannot access that reference. Most of the time this occurs without any intervention on the part of the programmer.

Using `.this` and `.new`

To produce the reference to the outer-class object, you name the outer class followed by a dot and `this`. The resulting reference is automatically the correct type, known and checked at compile time, so there is no runtime overhead. Here's how to use `.this`:

```
// innerclasses/DotThis.java
// Accessing the outer-class object

public class DotThis {
  void f() { System.out.println("DotThis.f()"); }
  public class Inner {
    public DotThis outer() {
      return DotThis.this;
      // A plain "this" would be Inner's "this"
    }
  }
  public Inner inner() { return new Inner(); }
  public static void main(String[] args) {
    DotThis dt = new DotThis();
    DotThis.Inner dti = dt.inner();
    dti.outer().f();
  }
}
/* Output:
DotThis.f()
*/
```

Sometimes you want to tell some other object to create an object of one of its inner classes. To do this, provide a reference to the other outer-class object in the `new` expression, using the `.new` syntax, like this:

```
// innerclasses/DotNew.java
// Creating an inner class directly using .new syntax

public class DotNew {
  public class Inner {}
  public static void main(String[] args) {
    DotNew dn = new DotNew();
    DotNew.Inner dni = dn.new Inner();
  }
}
```

To create an object of the inner class directly, you don't follow the same form and refer to the outer class name `DotNew` as you might expect. Instead, use an *object* of the outer class to make an object of the inner class, as you see above. This also resolves the name scoping issues for the inner class, so you don't say (indeed, you *cannot* say) `dn.new DotNew.Inner()`.

Here, you see `.new` applied to the "Parcel" example:

```
// innerclasses/Parcel3.java
// Using .new to create instances of inner classes

public class Parcel3 {
  class Contents {
    private int i = 11;
    public int value() { return i; }
  }
  class Destination {
    private String label;
    Destination(String whereTo) { label = whereTo; }
    String readLabel() { return label; }
  }
  public static void main(String[] args) {
    Parcel3 p = new Parcel3();
    // Must use instance of outer class
    // to create an instance of the inner class:
    Parcel3.Contents c = p.new Contents();
    Parcel3.Destination d =
      p.new Destination("Tasmania");
  }
}
```

It's not possible to create an object of the inner class unless you already have an object of the outer class. This is because the object of the inner class is quietly connected to the object of the outer class it was made from. However, if you make a *nested class* (a `static` inner class), it doesn't need a reference to the outer-class object.

Inner Classes and Upcasting

Inner classes become more interesting when you upcast to a base class, and in particular to an interface. (The effect of producing an interface reference from an object that implements it is essentially the same as upcasting to a base class.) That's because the inner class—the implementation of the interface—can then be unseen and unavailable, which is convenient for hiding the implementation. All you get back is a reference to the base class or the interface.

We can create interfaces for the previous examples:

```
// innerclasses/Destination.java
public interface Destination {
  String readLabel();
}
```

```
// innerclasses/Contents.java
public interface Contents {
  int value();
}
```

Now `Contents` and `Destination` represent interfaces available to the client programmer. Remember that an interface automatically makes all of its members `public`.

When you get a reference to the base class or the interface, it's possible you can't even find out the exact type, as shown here:

```
// innerclasses/TestParcel.java

class Parcel4 {
  private class PContents implements Contents {
    private int i = 11;
    @Override public int value() { return i; }
  }
  protected final class
  PDestination implements Destination {
    private String label;
    private PDestination(String whereTo) {
      label = whereTo;
    }
    @Override
    public String readLabel() { return label; }
  }
  public Destination destination(String s) {
    return new PDestination(s);
  }
  public Contents contents() {
    return new PContents();
  }
}

public class TestParcel {
  public static void main(String[] args) {
    Parcel4 p = new Parcel4();
    Contents c = p.contents();
    Destination d = p.destination("Tasmania");
    // Illegal -- can't access private class:
    //- Parcel4.PContents pc = p.new PContents();
  }
}
```

In `Parcel4`, the inner class `PContents` is `private`, so nothing but `Parcel4` can access it. Normal (non-inner) classes cannot be made `private` or `protected`; they can only be given `public` or package access.

`PDestination` is `protected`, so it can only be accessed by `Parcel4`, classes in the same package (because `protected` also gives package access), and the inheritors of `Parcel4`. This means the client programmer has restricted knowledge and access to these members. In fact, you can't even downcast to a `private` inner class (or a

`protected` inner class unless you're an inheritor), because you can't access the name, as you see in `class TestParcel`.

`private` inner classes provide a way for the class designer to completely prevent any type-coding dependencies and to completely hide details about implementation. In addition, extension of an interface is useless from the client programmer's perspective because the client programmer cannot access any additional methods that aren't part of the `public` interface. This also provides an opportunity for the Java compiler to generate more efficient code.

Inner Classes in Methods and Scopes

What you've seen so far encompasses the typical use for inner classes. In general, the inner classes you'll write and read are "plain", and simple to understand. However, the syntax for inner classes covers several more obscure techniques.

Inner classes can be created within a method or even an arbitrary scope. There are two reasons for doing this:

1. As shown previously, you're implementing an interface of some kind so you can create and return a reference.
2. You're solving a complicated problem and you create a class to aid in your solution, but you don't want it publicly available.

In the following examples, the previous code is modified to use:

1. A class defined within a method.
2. A class defined within a scope inside a method.
3. An *anonymous* class implementing an interface.
4. An anonymous class extending a class that has a non-default constructor.
5. An anonymous class that performs field initialization.
6. An anonymous class that performs construction using instance initialization (anonymous inner classes cannot have constructors).

The first example shows the creation of an entire class within the scope of a method (instead of the scope of another class). This is called a *local inner class*:

```
// innerclasses/Parcel5.java
// Nesting a class within a method

public class Parcel5 {
  public Destination destination(String s) {
    final class PDestination implements Destination {
      private String label;
      private PDestination(String whereTo) {
        label = whereTo;
      }
      @Override
      public String readLabel() { return label; }
    }
    return new PDestination(s);
  }
  public static void main(String[] args) {
    Parcel5 p = new Parcel5();
    Destination d = p.destination("Tasmania");
  }
}
```

The class `PDestination` is part of `destination()` rather than being part of `Parcel5`. Therefore, `PDestination` cannot be accessed outside of `destination()`. The upcasting in the `return` statement means nothing comes out of `destination()` except a reference to a `Destination` interface. The fact that the name of the class `PDestination` is placed inside `destination()` doesn't mean `PDestination` is not a valid object once `destination()` returns.

You can use the class identifier `PDestination` for an inner class inside each class in the same subdirectory without a name clash.

Next, see how you can nest an inner class within any arbitrary scope:

```
// innerclasses/Parcel6.java
// Nesting a class within a scope

public class Parcel6 {
  private void internalTracking(boolean b) {
    if(b) {
      class TrackingSlip {
        private String id;
        TrackingSlip(String s) {
          id = s;
        }
        String getSlip() { return id; }
      }
      TrackingSlip ts = new TrackingSlip("slip");
      String s = ts.getSlip();
    }
    // Can't use it here! Out of scope:
    //- TrackingSlip ts = new TrackingSlip("x");
  }
  public void track() { internalTracking(true); }
  public static void main(String[] args) {
    Parcel6 p = new Parcel6();
    p.track();
  }
}
```

The class `TrackingSlip` is nested inside the scope of an `if` statement. This does not mean that the *class* is conditionally created—it gets compiled along with everything else. However, it's not available outside the scope where it is defined. Other than that, it looks just like an ordinary class.

Anonymous Inner Classes

The next example looks a little odd:

```
// innerclasses/Parcel7.java
// Returning an instance of an anonymous inner class

public class Parcel7 {
  public Contents contents() {
    return new Contents() { // Insert class definition
      private int i = 11;
      @Override public int value() { return i; }
    }; // Semicolon required
  }
  public static void main(String[] args) {
    Parcel7 p = new Parcel7();
    Contents c = p.contents();
  }
}
```

The `contents()` method combines the creation of the return value with the definition of the class that represents that return value. In addition, the class has no name—it's anonymous. It looks as if you're creating a `Contents` object, But then, before you get to the semicolon, you say, "But wait, I think I'll slip in a class definition."

What this strange syntax means is "Create an object of an anonymous class that's inherited from `Contents`." The reference returned by the `new` expression is automatically upcast to a `Contents` reference. The anonymous inner-class syntax is shorthand for:

```
// innerclasses/Parcel7b.java
// Expanded version of Parcel7.java

public class Parcel7b {
  class MyContents implements Contents {
    private int i = 11;
    @Override public int value() { return i; }
  }
  public Contents contents() {
    return new MyContents();
  }
  public static void main(String[] args) {
    Parcel7b p = new Parcel7b();
    Contents c = p.contents();
```

```
  }
}
```

In the anonymous inner class, `Contents` is created with a zero-argument constructor. Here's what to do if your base class needs a constructor with an argument:

```
// innerclasses/Parcel8.java
// Calling the base-class constructor

public class Parcel8 {
  public Wrapping wrapping(int x) {
    // Base constructor call:
    return new Wrapping(x) {          // [1]
      @Override public int value() {
        return super.value() * 47;
      }
    };                                // [2]
  }
  public static void main(String[] args) {
    Parcel8 p = new Parcel8();
    Wrapping w = p.wrapping(10);
  }
}
```

- **[1]** You pass the appropriate argument to the base-class constructor.
- **[2]** The semicolon at the end of an anonymous inner class doesn't mark the end of the class body. Instead, it marks the end of the expression that happens to contain the anonymous class. Thus, it's identical to the way the semicolon is used everywhere else.

Although it's an ordinary class with an implementation, `Wrapping` is also used as a common "interface" to its derived classes:

```
// innerclasses/Wrapping.java
public class Wrapping {
  private int i;
  public Wrapping(int x) { i = x; }
  public int value() { return i; }
}
```

For variety, `Wrapping`'s constructor requires an argument.

You can also perform initialization when you define fields in an anonymous class:

```
// innerclasses/Parcel9.java

public class Parcel9 {
  // Argument must be final or "effectively final"
  // to use within the anonymous inner class:
  public Destination destination(final String dest) {
    return new Destination() {
      private String label = dest;
      @Override
      public String readLabel() { return label; }
    };
  }
  public static void main(String[] args) {
    Parcel9 p = new Parcel9();
    Destination d = p.destination("Tasmania");
  }
}
```

If you're defining an anonymous inner class and must use an object that's defined outside the anonymous inner class, the compiler requires that the argument reference be `final` or "effectively final" (that is, it's never changed after initialization, so it can be treated as if it is `final`), as you see in the argument to `destination()`. Here you can leave off the `final` without a problem, but it's usually better to include it as a reminder.

As long as you're assigning a field, the approach in this example is fine. But what if you must perform some constructor-like activity? You can't have a named constructor in an anonymous class (because there's no name). With *instance initialization*, you can, in effect, create a constructor for an anonymous inner class, like this:

```
// innerclasses/AnonymousConstructor.java
// Creating a constructor for an anonymous inner class

abstract class Base {
  Base(int i) {
    System.out.println("Base constructor, i = " + i);
  }
  public abstract void f();
}

public class AnonymousConstructor {
  public static Base getBase(int i) {
    return new Base(i) {
      { System.out.println(
        "Inside instance initializer"); }
      @Override public void f() {
        System.out.println("In anonymous f()");
      }
    };
  }
  public static void main(String[] args) {
    Base base = getBase(47);
    base.f();
  }
}
/* Output:
Base constructor, i = 47
Inside instance initializer
In anonymous f()
*/
```

Here, the variable `i` did *not* have to be `final`. While `i` is passed to the base constructor of the anonymous class, it is never directly used *inside* the anonymous class.

Here's the "parcel" theme with instance initialization. Note that the arguments to `destination()` must be `final` or "effectively final" because they are used within the anonymous class:

```
// innerclasses/Parcel10.java
// Using "instance initialization" to perform
// construction on an anonymous inner class

public class Parcel10 {
  public Destination
  destination(final String dest, final float price) {
    return new Destination() {
      private int cost;
      // Instance initialization for each object:
      {
        cost = Math.round(price);
        if(cost > 100)
          System.out.println("Over budget!");
      }
      private String label = dest;
      @Override
      public String readLabel() { return label; }
    };
  }
  public static void main(String[] args) {
    Parcel10 p = new Parcel10();
    Destination d = p.destination("Tasmania", 101.395F);
  }
}
/* Output:
Over budget!
*/
```

Inside the instance initializer you see code that couldn't be executed as part of a field initializer (that is, the `if` statement). So in effect, an instance initializer is the constructor for an anonymous inner class. However, it's limited; you can't overload instance initializers, so you can have only one of these constructors.

Anonymous inner classes are somewhat limited compared to regular inheritance, because they can either extend a class or implement an interface, but not both. And if you do implement an interface, you can only implement one.

Nested Classes

If you don't need a connection between the inner-class object and the outer-class object, you can make the inner class `static`. This is commonly called a *nested class*. To understand the meaning of `static` when applied to inner classes, remember that the object of an ordinary inner class implicitly keeps a reference to the object of the enclosing class that created it. This is not true for a `static` inner class. A nested class means:

1. You don't need an outer-class object to create an object of a nested class.
2. You can't access a non-`static` outer-class object from an object of a nested class.

Nested classes are different from ordinary inner classes in another way. Fields and methods in ordinary inner classes can only be at the outer level of a class, so ordinary inner classes cannot have `static` data, `static` fields, or nested classes. However, nested classes can have all these:

```
// innerclasses/Parcel11.java
// Nested classes (static inner classes)

public class Parcel11 {
  private static class
  ParcelContents implements Contents {
    private int i = 11;
    @Override public int value() { return i; }
  }
  protected static final class ParcelDestination
  implements Destination {
    private String label;
    private ParcelDestination(String whereTo) {
      label = whereTo;
    }
    @Override
    public String readLabel() { return label; }
    // Nested classes can contain other static elements:
    public static void f() {}
    static int x = 10;
    static class AnotherLevel {
```

```
      public static void f() {}
      static int x = 10;
    }
  }
  public static Destination destination(String s) {
    return new ParcelDestination(s);
  }
  public static Contents contents() {
    return new ParcelContents();
  }
  public static void main(String[] args) {
    Contents c = contents();
    Destination d = destination("Tasmania");
  }
}
```

In `main()`, no object of `Parcel11` is necessary; instead, you use the normal syntax for selecting a `static` member to call the methods that return references to `Contents` and `Destination`.

An ordinary (non-`static`) inner class can create a link to the outer-class object using a special `this` reference. A nested class does not have a special `this` reference, which makes it analogous to a `static` method.

Classes Inside Interfaces

A nested class can be part of an interface. Any class you put inside an interface is automatically `public` and `static`. The class is `static` so the nested class is only placed inside the namespace of the interface. You can even implement the surrounding interface in the inner class, like this:

```
// innerclasses/ClassInInterface.java
// {java ClassInInterface$Test}

public interface ClassInInterface {
  void howdy();
  class Test implements ClassInInterface {
    @Override public void howdy() {
      System.out.println("Howdy!");
    }
    public static void main(String[] args) {
      new Test().howdy();
    }
  }
}
/* Output:
Howdy!
*/
```

It's convenient to nest a class inside an interface when you create common code to use with all different implementations of that interface.

Earlier I suggested putting a `main()` in every class to act as a test bed for that class. A potential drawback is that your test fixtures are exposed in your shipping product. If this is a problem, you can use a nested class to hold your test code:

```
// innerclasses/TestBed.java
// Putting test code in a nested class
// {java TestBed$Tester}

public class TestBed {
  public void f() { System.out.println("f()"); }
  public static class Tester {
    public static void main(String[] args) {
      TestBed t = new TestBed();
      t.f();
    }
  }
}
/* Output:
f()
*/
```

This generates a separate class called `TestBed$Tester` (to run the program, you say `java TestBed$Tester`, but you must escape the `$` under Unix/Linux systems). You can use this class for testing, but you don't have to include it in your shipping product; you can delete `TestBed$Tester.class` before packaging things up.

Reaching Outward from a Multiply Nested Class

It doesn't matter how deeply an inner class is nested—it can transparently access all members of all classes it is nested within, as seen here:

```
// innerclasses/MultiNestingAccess.java
// Nested classes can access all members of all
// levels of the classes they are nested within

class MNA {
  private void f() {}
  class A {
    private void g() {}
    public class B {
      void h() {
        g();
        f();
      }
    }
  }
}

public class MultiNestingAccess {
  public static void main(String[] args) {
    MNA mna = new MNA();
    MNA.A mnaa = mna.new A();
    MNA.A.B mnaab = mnaa.new B();
    mnaab.h();
  }
}
```

Notice how the `private` methods `g()` and `f()` are callable without any qualification. This example also demonstrates the syntax necessary to create objects of multiply nested inner classes when you create the objects in a different class. The "`.new`" syntax produces the correct scope, so you do not have to qualify the class name in the constructor call.

Why Inner Classes?

You've seen a lot of syntax and semantics describing the way inner classes work, but this doesn't answer the question of why they exist. Why did the Java designers go to so much trouble to add this fundamental language feature?

Typically, the inner class inherits from a class or implements an interface, and the code in the inner class manipulates the outer-class object it was created within. An inner class provides a kind of window into the outer class.

A question that cuts to the heart of inner classes is this: If I just need a reference to an interface, why don't I just make the outer class implement that interface? The answer is "If that's all you need, that's how to do it." So what is it that distinguishes an inner class implementing an interface from an outer class implementing the same interface? The answer is that you can't always have the convenience of interfaces—sometimes you're working with implementations. So the most compelling reason for inner classes is:

> Each inner class can independently inherit from an implementation. Thus, the inner class is not limited by whether the outer class is already inheriting from an implementation.

Without the ability that inner classes provide to inherit—in effect—from more than one concrete or `abstract` class, some design and programming problems are intractable. So one way to look at the inner class is as the rest of the solution of the multiple-inheritance problem. Interfaces solve part of the problem, but inner classes effectively allow "multiple implementation inheritance." That is, inner classes effectively allow you to inherit from more than one non-interface.

To see this in more detail, consider a situation where you have two interfaces that must somehow be implemented within a class. Because of the flexibility of interfaces, you have two choices: a single class or an inner class.

```
// innerclasses/mui/MultiInterfaces.java
// Two ways a class can implement multiple interfaces
// {java innerclasses.mui.MultiInterfaces}
package innerclasses.mui;

interface A {}
interface B {}

class X implements A, B {}

class Y implements A {
  B makeB() {
    // Anonymous inner class:
    return new B() {};
  }
}

public class MultiInterfaces {
  static void takesA(A a) {}
  static void takesB(B b) {}
  public static void main(String[] args) {
    X x = new X();
    Y y = new Y();
    takesA(x);
    takesA(y);
    takesB(x);
    takesB(y.makeB());
  }
}
```

This assumes the structure of your code makes logical sense either way. You'll ordinarily have some kind of guidance from the nature of the problem about whether to use a single class or an inner class. But without any other constraints, the approach in the preceding example doesn't really make much difference from an implementation standpoint. Both of them work.

Using `abstract` or concrete classes instead of interfaces suddenly limits you to inner classes if your class must somehow implement both of the others:

```
// innerclasses/MultiImplementation.java
// For concrete or abstract classes, inner classes
// produce "multiple implementation inheritance"
// {java innerclasses.MultiImplementation}
package innerclasses;

class D {}
abstract class E {}

class Z extends D {
  E makeE() { return new E() {}; }
}

public class MultiImplementation {
  static void takesD(D d) {}
  static void takesE(E e) {}
  public static void main(String[] args) {
    Z z = new Z();
    takesD(z);
    takesE(z.makeE());
  }
}
```

If you didn't have to solve the "multiple implementation inheritance" problem, you could conceivably code around everything else without the need for inner classes. But with inner classes you have these additional features:

1. The inner class can have multiple instances, each with its own state information, independent of the information in the outer-class object.
2. In a single outer class you can have several inner classes, each of which implements the same interface or inherits from the same class in a different way. An example of this is shown shortly.
3. The point of creation of the inner-class object is not tied to the creation of the outer-class object.
4. There is no potentially confusing "is-a" relationship with the inner class; it's a separate entity.

As an example, if `Sequence.java` did not use inner classes, you'd say, "A `Sequence` is a `Selector`," and you'd only be allowed one `Selector` for a particular `Sequence`.

You can easily have a second method, `reverseSelector()`, that produces a `Selector` that moves backward through the sequence. This kind of flexibility is only available with inner classes.

Closures & Callbacks

A *closure* is a callable object that retains information from the scope where it was created. From this definition, you see that an inner class is an object-oriented closure, because it doesn't just contain each piece of information from the outer-class object ("the scope where it was created"), but it automatically holds a reference back to the whole outer-class object, where it has permission to manipulate all the members, even `private` ones.

Before Java 8, the only way to produce closure-like behavior was through inner classes. With Java 8, we now have lambda expressions which also have closure behavior, but with much nicer and more concise syntax; you'll learn all about these in the Functional Programming chapter. Although you should prefer lambdas to inner-class closures, you will see pre-Java-8 code that uses the inner class approach, so it's still necessary to understand it.

One of the most compelling arguments for including some kind of pointer mechanism in Java was to allow *callbacks*. With a callback, some other object is given a piece of information that allows it to call back into the originating object at some later point. This is a powerful concept, as you will see later in the book. If a callback is implemented using a pointer, however, you must rely on the programmer to behave properly and not misuse the pointer. As you've seen by now, Java tends to be more careful than that, so pointers were not included in the language.

The closure provided by the inner class is a good solution—more flexible and far safer than a pointer. Here's an example:

```
// innerclasses/Callbacks.java
// Using inner classes for callbacks
// {java innerclasses.Callbacks}
package innerclasses;

interface Incrementable {
  void increment();
}

// Very simple to just implement the interface:
class Callee1 implements Incrementable {
  private int i = 0;
  @Override public void increment() {
    i++;
    System.out.println(i);
  }
}

class MyIncrement {
  public void increment() {
    System.out.println("Other operation");
  }
  static void f(MyIncrement mi) { mi.increment(); }
}

// If your class must implement increment() in
// some other way, you must use an inner class:
class Callee2 extends MyIncrement {
  private int i = 0;
  @Override public void increment() {
    super.increment();
    i++;
    System.out.println(i);
  }
  private class Closure implements Incrementable {
    @Override public void increment() {
      // Specify outer-class method, otherwise
      // you'll get an infinite recursion:
      Callee2.this.increment();
    }
  }
  Incrementable getCallbackReference() {
```

```
    return new Closure();
  }
}

class Caller {
  private Incrementable callbackReference;
  Caller(Incrementable cbh) {
    callbackReference = cbh;
  }
  void go() { callbackReference.increment(); }
}

public class Callbacks {
  public static void main(String[] args) {
    Callee1 c1 = new Callee1();
    Callee2 c2 = new Callee2();
    MyIncrement.f(c2);
    Caller caller1 = new Caller(c1);
    Caller caller2 =
      new Caller(c2.getCallbackReference());
    caller1.go();
    caller1.go();
    caller2.go();
    caller2.go();
  }
}
/* Output:
Other operation
1
1
2
Other operation
2
Other operation
3
*/
```

This shows a further distinction between implementing an interface in an outer class versus doing so in an inner class. `Callee1` is clearly the simpler solution in terms of the code. `Callee2` inherits from `MyIncrement`, which already has a different `increment()` method that does something unrelated to the one expected by the `Incrementable` interface. When `MyIncrement` is inherited into `Callee2`,

`increment()` can't be overridden for use by `Incrementable`, so you're forced to provide a separate implementation using an inner class. Also note that when you create an inner class, you do not add to or modify the interface of the outer class.

Everything except `getCallbackReference()` in `Callee2` is `private`. To allow *any* connection to the outside world, the interface `Incrementable` is essential. Here you see how interfaces allow for a complete separation of interface from implementation.

The inner class `Closure` implements `Incrementable` to provide a hook back into `Callee2`—but a safe hook. Whoever gets the `Incrementable` reference can only call `increment()` and has no other abilities (unlike a pointer, which would allow you to run wild).

`Caller` takes an `Incrementable` reference in its constructor (although capturing the callback reference can happen at any time) and then, sometime later, uses the reference to "call back" into the `Callee` class.

The value of the callback is in its flexibility; you can dynamically decide what methods are called at run time. In user interfaces, for example, callbacks are often used everywhere to implement GUI functionality.

Inner Classes & Control Frameworks

A more concrete example of inner classes is found in something I refer to here as a *control framework*.

An *application framework* is a class or a set of classes designed to solve a particular type of problem. To apply an application framework, you typically inherit from one or more classes and override some of the methods. The code you write in the overridden methods customizes the general solution provided by that application framework to solve your specific problem. This is an example of the *Template Method* design pattern. The Template Method contains the basic structure of the algorithm, and it calls one or more overrideable methods to complete the action of that algorithm. A design pattern separates things that change from things that stay the same, and here the Template Method is the part that stays the same, and the overrideable methods are the things that change.

A control framework is a particular type of application framework dominated by the need to respond to events. A system that primarily responds to events is called

an *event-driven system*. A common problem in application programming is the graphical user interface (GUI), which is almost entirely event-driven.

To see how inner classes allow the simple creation and use of control frameworks, consider a framework that executes events whenever those events are "ready." Although "ready" could mean anything, here it is based on clock time. What follows is a control framework that contains no specific information about what it's controlling. That information is supplied during inheritance, when the `action()` portion of the algorithm is implemented.

Here is the interface that describes any control event. It's an `abstract` class instead of an actual interface because the default behavior is to perform the control based on time. Thus, some of the implementation is included:

```
// innerclasses/controller/Event.java
// The common methods for any control event
package innerclasses.controller;
import java.time.*; // Java 8 time classes

public abstract class Event {
  private Instant eventTime;
  protected final Duration delayTime;
  public Event(long millisecondDelay) {
    delayTime = Duration.ofMillis(millisecondDelay);
    start();
  }
  public void start() { // Allows restarting
    eventTime = Instant.now().plus(delayTime);
  }
  public boolean ready() {
    return Instant.now().isAfter(eventTime);
  }
  public abstract void action();
}
```

The constructor captures the time in milliseconds (measured from the time of creation of the object) when you want the `Event` to run, then calls `start()`, which takes the current time and adds the delay time to produce the time when the event will occur. Rather than including it in the constructor, `start()` is a separate method. This way, you can restart the timer after the event has run out, so the `Event` object

can be reused. For example, if you want a repeating event, you can call `start()` inside your `action()` method.

`ready()` tells you when it's time to run the `action()` method. However, `ready()` can be overridden in a derived class to base the `Event` on something other than time.

Next we write the actual control framework to manage and fire events. The `Event` objects are held inside a collection object of type `List<Event>` (pronounced "List of Event"), which you'll learn more about in the Collections chapter. For now, all you must know is that `add()` will append an `Event` to the end of the `List`, `size()` produces the number of entries in the `List`, the *for-in* syntax fetches successive `Events` from the `List`, and `remove()` removes the specified `Event` from the `List`.

```
// innerclasses/controller/Controller.java
// The reusable framework for control systems
package innerclasses.controller;
import java.util.*;

public class Controller {
  // A class from java.util to hold Event objects:
  private List<Event> eventList = new ArrayList<>();
  public void addEvent(Event c) { eventList.add(c); }
  public void run() {
    while(eventList.size() > 0)
      // Make a copy so you're not modifying the list
      // while you're selecting the elements in it:
      for(Event e : new ArrayList<>(eventList))
        if(e.ready()) {
          System.out.println(e);
          e.action();
          eventList.remove(e);
        }
  }
}
```

The `run()` method loops through a copy of the `eventList`, hunting for an `Event` object that's `ready()` to run. For each one it finds `ready()`, it prints information using the object's `toString()` method, calls the `action()` method, then removes the `Event` from the list.

So far, you know nothing about exactly *what* an `Event` does. And this is the crux of the design—how it "separates the things that change from the things that stay the

same." Or, to use my term, the "vector of change" is the different actions of the various kinds of `Event` objects, and you express different actions by creating different `Event` subclasses.

This is where inner classes come into play. They allow two things:

1. The entire implementation of a control framework is created in a single class, thereby encapsulating everything that's unique about that implementation. Inner classes are used to express the many different kinds of `action()` necessary to solve the problem.
2. Inner classes keep this implementation from becoming awkward, because you easily access any of the members in the outer class. Without this, the code might become unpleasant enough that you'd seek an alternative.

Consider a particular implementation of the control framework designed to control greenhouse functions. Each action is entirely different: turning lights, water, and thermostats on and off, ringing bells, and restarting the system. The control framework is designed to easily isolate this different code. Inner classes allow multiple derived versions of the same base class, `Event`, within a single class. For each type of action, you inherit a new `Event` inner class, and write the control code in the `action()` implementation.

As is typical with an application framework, the class `GreenhouseControls` inherits `Controller`:

```
// innerclasses/GreenhouseControls.java
// This produces a specific application of the
// control system, all in a single class. Inner
// classes allow you to encapsulate different
// functionality for each type of event.
import innerclasses.controller.*;

public class GreenhouseControls extends Controller {
  private boolean light = false;
  public class LightOn extends Event {
    public LightOn(long delayTime) {
      super(delayTime);
    }
    @Override public void action() {
      // Put hardware control code here to
```

```
      // physically turn on the light.
      light = true;
    }
    @Override public String toString() {
      return "Light is on";
    }
  }
  public class LightOff extends Event {
    public LightOff(long delayTime) {
      super(delayTime);
    }
    @Override public void action() {
      // Put hardware control code here to
      // physically turn off the light.
      light = false;
    }
    @Override public String toString() {
      return "Light is off";
    }
  }
  private boolean water = false;
  public class WaterOn extends Event {
    public WaterOn(long delayTime) {
      super(delayTime);
    }
    @Override public void action() {
      // Put hardware control code here.
      water = true;
    }
    @Override public String toString() {
      return "Greenhouse water is on";
    }
  }
  public class WaterOff extends Event {
    public WaterOff(long delayTime) {
      super(delayTime);
    }
    @Override public void action() {
      // Put hardware control code here.
      water = false;
    }
    @Override public String toString() {
```

```
    return "Greenhouse water is off";
  }
}
private String thermostat = "Day";
public class ThermostatNight extends Event {
  public ThermostatNight(long delayTime) {
    super(delayTime);
  }
  @Override public void action() {
    // Put hardware control code here.
    thermostat = "Night";
  }
  @Override public String toString() {
    return "Thermostat on night setting";
  }
}
public class ThermostatDay extends Event {
  public ThermostatDay(long delayTime) {
    super(delayTime);
  }
  @Override public void action() {
    // Put hardware control code here.
    thermostat = "Day";
  }
  @Override public String toString() {
    return "Thermostat on day setting";
  }
}
// An example of an action() that inserts a
// new one of itself into the event list:
public class Bell extends Event {
  public Bell(long delayTime) {
    super(delayTime);
  }
  @Override public void action() {
    addEvent(new Bell(delayTime.toMillis()));
  }
  @Override public String toString() {
    return "Bing!";
  }
}
public class Restart extends Event {
```

```
    private Event[] eventList;
    public
    Restart(long delayTime, Event[] eventList) {
      super(delayTime);
      this.eventList = eventList;
      for(Event e : eventList)
        addEvent(e);
    }
    @Override public void action() {
      for(Event e : eventList) {
        e.start(); // Rerun each event
        addEvent(e);
      }
      start(); // Rerun this Event
      addEvent(this);
    }
    @Override public String toString() {
      return "Restarting system";
    }
  }
  public static class Terminate extends Event {
    public Terminate(long delayTime) {
      super(delayTime);
    }
    @Override
    public void action() { System.exit(0); }
    @Override public String toString() {
      return "Terminating";
    }
  }
}
```

Note that `light`, `water`, and `thermostat` belong to the outer class `GreenhouseControls`, and yet the inner classes can access those fields without qualification or special permission. Also, the `action()` methods usually involve some sort of hardware control.

Most of the `Event` classes look similar, but `Bell` and `Restart` are special. `Bell` rings, then adds a new `Bell` object to the event list, so it will ring again later. Notice how inner classes *almost* look like multiple inheritance: `Bell` and `Restart` have all the methods of `Event` and also appear to have all the methods of the outer class

`GreenhouseControls`.

`Restart` is given an array of `Event` objects it adds to the controller. `Restart` is just another `Event` object so you can also add a `Restart` object within `Restart.action()` so the system regularly restarts itself.

The following class configures the system by creating a `GreenhouseControls` object and adding various kinds of `Event` objects. This is an example of the *Command* design pattern—each object in `eventList` is a request encapsulated as an object:

```
// innerclasses/GreenhouseController.java
// Configure and execute the greenhouse system
import innerclasses.controller.*;

public class GreenhouseController {
  public static void main(String[] args) {
    GreenhouseControls gc = new GreenhouseControls();
    // Instead of using code, you could parse
    // configuration information from a text file:
    gc.addEvent(gc.new Bell(900));
    Event[] eventList = {
      gc.new ThermostatNight(0),
      gc.new LightOn(200),
      gc.new LightOff(400),
      gc.new WaterOn(600),
      gc.new WaterOff(800),
      gc.new ThermostatDay(1400)
    };
    gc.addEvent(gc.new Restart(2000, eventList));
    gc.addEvent(
      new GreenhouseControls.Terminate(5000));
    gc.run();
  }
}
/* Output:
Thermostat on night setting
Light is on
Light is off
Greenhouse water is on
Greenhouse water is off
Bing!
Thermostat on day setting
```

```
Bing!
Restarting system
Thermostat on night setting
Light is on
Light is off
Greenhouse water is on
Bing!
Greenhouse water is off
Thermostat on day setting
Bing!
Restarting system
Thermostat on night setting
Light is on
Light is off
Bing!
Greenhouse water is on
Greenhouse water is off
Terminating
*/
```

This class initializes the system and adds all the appropriate events. The `Restart` event is repeatedly run, and it loads the `eventList` into the `GreenhouseControls` object each time. If you provide a command-line argument indicating milliseconds, it will terminate the program after that many milliseconds (this is used for testing).

It's more flexible to read the events from a file instead of hard-coding them.

This example moves you toward an appreciation of the value of inner classes, especially when used within a control framework.

Inheriting from Inner Classes

Because the inner-class constructor must attach to a reference of the enclosing class object, things are slightly complicated when you inherit from an inner class. The problem is that the "secret" reference to the enclosing class object *must* be initialized, and yet in the derived class there's no longer a default object to attach to. You must use a special syntax to make the association explicit:

```
// innerclasses/InheritInner.java
// Inheriting an inner class

class WithInner {
  class Inner {}
}

public class InheritInner extends WithInner.Inner {
  //- InheritInner() {} // Won't compile
  InheritInner(WithInner wi) {
    wi.super();
  }
  public static void main(String[] args) {
    WithInner wi = new WithInner();
    InheritInner ii = new InheritInner(wi);
  }
}
```

`InheritInner` is extending only the inner class, not the outer one. But when it comes time to create a constructor, the default one is no good, and you can't just pass a reference to an enclosing object. In addition, you must use the syntax

```
enclosingClassReference.super();
```

inside the constructor. This provides the necessary reference, and the program will then compile.

Can Inner Classes Be Overridden?

What happens when you create an inner class, then inherit from the enclosing class and redefine the inner class? That is, is it possible to "override" the entire inner class? This seems like a powerful concept, but "overriding" an inner class as if it were another method of the outer class doesn't really do anything:

```
// innerclasses/BigEgg.java
// An inner class cannot be overridden like a method

class Egg {
  private Yolk y;
  protected class Yolk {
    public Yolk() {
      System.out.println("Egg.Yolk()");
    }
  }
  Egg() {
    System.out.println("New Egg()");
    y = new Yolk();
  }
}

public class BigEgg extends Egg {
  public class Yolk {
    public Yolk() {
      System.out.println("BigEgg.Yolk()");
    }
  }
  public static void main(String[] args) {
    new BigEgg();
  }
}
/* Output:
New Egg()
Egg.Yolk()
*/
```

The zero-argument constructor is synthesized automatically by the compiler, and this calls the base-class zero-argument constructor. You might think that, because a `BigEgg` is created, the "overridden" version of `Yolk` is used, but this is not the case, as you see from the output.

There's no extra inner-class magic when you inherit from the outer class. The two inner classes are completely separate entities, each in its own namespace. However, it's still possible to explicitly inherit from the inner class:

```
// innerclasses/BigEgg2.java
// Proper inheritance of an inner class

class Egg2 {
  protected class Yolk {
    public Yolk() {
      System.out.println("Egg2.Yolk()");
    }
    public void f() {
      System.out.println("Egg2.Yolk.f()");
    }
  }
  private Yolk y = new Yolk();
  Egg2() { System.out.println("New Egg2()"); }
  public void insertYolk(Yolk yy) { y = yy; }
  public void g() { y.f(); }
}

public class BigEgg2 extends Egg2 {
  public class Yolk extends Egg2.Yolk {
    public Yolk() {
      System.out.println("BigEgg2.Yolk()");
    }
    @Override public void f() {
      System.out.println("BigEgg2.Yolk.f()");
    }
  }
  public BigEgg2() { insertYolk(new Yolk()); }
  public static void main(String[] args) {
    Egg2 e2 = new BigEgg2();
    e2.g();
  }
}
/* Output:
Egg2.Yolk()
New Egg2()
Egg2.Yolk()
BigEgg2.Yolk()
BigEgg2.Yolk.f()
*/
```

Now `BigEgg2.Yolk` explicitly `extends` `Egg2.Yolk` and overrides its methods. The

method `insertYolk()` allows `BigEgg2` to upcast one of its own `Yolk` objects into the `y` reference in `Egg2`, so when `g()` calls `y.f()`, the overridden version of `f()` is used. The second call to `Egg2.Yolk()` is the base-class constructor call of the `BigEgg2.Yolk` constructor. The overridden version of `f()` is used when `g()` is called.

Local Inner Classes

As noted earlier, inner classes can also be created inside code blocks, typically inside the body of a method. A local inner class cannot have an access specifier because it isn't part of the outer class, but it does have access to the final variables in the current code block and all the members of the enclosing class. Here's an example comparing the creation of a local inner class with an anonymous inner class:

```
// innerclasses/LocalInnerClass.java
// Holds a sequence of Objects

interface Counter {
  int next();
}

public class LocalInnerClass {
  private int count = 0;
  Counter getCounter(final String name) {
    // A local inner class:
    class LocalCounter implements Counter {
      LocalCounter() {
        // Local inner class can have a constructor
        System.out.println("LocalCounter()");
      }
      @Override public int next() {
        System.out.print(name); // Access local final
        return count++;
      }
    }
    return new LocalCounter();
  }
  // Repeat, but with an anonymous inner class:
  Counter getCounter2(final String name) {
```

```
    return new Counter() {
      // Anonymous inner class cannot have a named
      // constructor, only an instance initializer:
      {
        System.out.println("Counter()");
      }
      @Override public int next() {
        System.out.print(name); // Access local final
        return count++;
      }
    };
  }
  public static void main(String[] args) {
    LocalInnerClass lic = new LocalInnerClass();
    Counter
      c1 = lic.getCounter("Local inner "),
      c2 = lic.getCounter2("Anonymous inner ");
    for(int i = 0; i < 5; i++)
      System.out.println(c1.next());
    for(int i = 0; i < 5; i++)
      System.out.println(c2.next());
  }
}
/* Output:
LocalCounter()
Counter()
Local inner 0
Local inner 1
Local inner 2
Local inner 3
Local inner 4
Anonymous inner 5
Anonymous inner 6
Anonymous inner 7
Anonymous inner 8
Anonymous inner 9
*/
```

`Counter` returns the next value in a sequence. It is implemented here as both a local class and an anonymous inner class, each with the same behaviors and capabilities. The name of the local inner class is not accessible outside the method so the only justification for using a local inner class instead of an anonymous inner class is if

you need a named constructor and/or an overloaded constructor (An anonymous inner class can only use instance initialization).

Another reason to make a local inner class rather than an anonymous inner class is if you make more than one object of that class.

Inner-Class Identifiers

After compilation, every class produces a `.class` file that holds all the information about how to create objects of that type. Upon loading, each class file produces a "meta-class" called the `Class` object.

You might guess that inner classes also produce `.class` files to contain the information for *their* `Class` objects. The names of these files/classes have a formula: the name of the enclosing class, followed by a `$`, followed by the name of the inner class. For example, the `.class` files created by `LocalInnerClass.java` include:

```
Counter.class
LocalInnerClass$1.class
LocalInnerClass$1LocalCounter.class
LocalInnerClass.class
```

If inner classes are anonymous, the compiler generates numbers as inner-class identifiers. If inner classes are nested within inner classes, their names are appended after a `$` and the outer-class identifier(s).

This scheme of generating internal names is simple and straightforward. It's also robust and handles most situations.[77] It is the standard naming scheme for Java so the generated files are automatically platform-independent. (Note that the Java compiler is changing your inner classes in all sorts of other ways to make them work.)

Summary

Interfaces and inner classes are more sophisticated concepts than what you'll find in many OOP languages; for example, there's nothing like them in C++. Together,

[77] On the other hand, `$` is a meta-character to the Unix shell and so you'll sometimes have trouble when listing the `.class` files. This is a bit strange coming from Sun, a Unix-based company. My guess is they weren't considering this issue, but instead thought you'd naturally focus on the source-code files.

they solve the same problem that C++ attempts to solve with its multiple inheritance (MI) feature. However, MI in C++ turns out to be rather difficult to use, whereas Java interfaces and inner classes are, by comparison, much more accessible.

Although the features themselves are reasonably straightforward, using these features is a design issue, much the same as polymorphism. Over time, you'll become better at recognizing situations where you should use an interface, or an inner class, or both. But at this point in the book, you should at least be comfortable with the syntax and semantics. As you see these language features in use, you'll eventually internalize them.

Collections

> It's a fairly simple program that only has a fixed quantity of objects with known lifetimes.

In general, your programs will always be creating new objects based on some criteria known only at run time. You won't know the quantity or even the exact type of the objects you need. To solve the general programming problem, you must create any number of objects, anytime, anywhere. So you can't rely on creating a named reference to hold each one of your objects:

```
MyType aReference;
```

because you'll never know how many of these you'll actually need.

Most languages provide some way to solve this essential problem. Java has several ways to hold objects (or rather, references to objects). The compiler-supported type is the array, previously discussed. An array is the most efficient way to hold a group of objects, and you're pointed towards this choice to hold a group of primitives. But an array has a fixed size, and in the more general case, you won't know at the time you're writing the program how many objects you're going to need, or whether you need a more sophisticated way to store your objects—so the fixed-sized constraint of an array is too limiting.

The `java.util` library has a reasonably complete set of *collection classes* to solve this problem, the basic types of which are `List`, `Set`, `Queue`, and `Map`. These types are also known as *container classes*, but I shall use the term that the Java library uses. Collections provide sophisticated ways to hold objects, and solve a surprising number of problems.

Among their other characteristics—`Set`, for example, holds only one object of each value, and `Map` is an *associative array* that lets you associate objects with other objects—the Java collection classes will automatically resize themselves. So, unlike arrays, you can put in any number of objects without worrying about how big the collection should be.

Even though they don't have direct keyword support in Java,[78] collection classes are fundamental tools that significantly increase your programming muscle. In this chapter you'll get a basic working knowledge of the Java collection library, with an emphasis on typical usage. Here, we'll focus on the collections that you'll use in day-to-day programming. Later, in the Appendix: Collection Topics, you'll learn about the rest of the collections and more details about their functionality and how to use them.

Generics and Type-Safe Collections

One of the problems of using pre-Java 5 collections was that the compiler allowed you to insert an incorrect type into a collection. For example, consider a collection of `Apple` objects, using the basic workhorse collection, `ArrayList`. For now, you can think of `ArrayList` as "an array that automatically expands itself." Using an `ArrayList` is straightforward: Create one, insert objects using `add()`, and access them with `get()`, using an index—just as you do with an array, but without the square brackets.[79] `ArrayList` also has a method `size()` that tells how many elements were added, so you don't inadvertently index off the end and cause an error (by throwing a *runtime exception*; exceptions are introduced in the chapter *Exceptions*).

In this example, `Apples` and `Oranges` are placed into the collection, then pulled out. Normally, the Java compiler will give you a warning because the example does *not* use generics. Here, a special *annotation* is used to suppress the warning. Annotations start with an `@` sign, and can take an argument; this one is `@SuppressWarnings` and the argument indicates that "unchecked" warnings only should be suppressed (You'll learn more about Java annotations in the Annotations chapter):

[78] A number of languages, such as Perl, Python, and Ruby, have native support for collections.

[79] This is a place where operator overloading would have been nice. C++ and C# collection classes produce a cleaner syntax using operator overloading.

```
// collections/ApplesAndOrangesWithoutGenerics.java
// Simple collection use (suppressing compiler warnings)
// {ThrowsException}
import java.util.*;

class Apple {
  private static long counter;
  private final long id = counter++;
  public long id() { return id; }
}

class Orange {}

public class ApplesAndOrangesWithoutGenerics {
  @SuppressWarnings("unchecked")
  public static void main(String[] args) {
    ArrayList apples = new ArrayList();
    for(int i = 0; i < 3; i++)
      apples.add(new Apple());
    // No problem adding an Orange to apples:
    apples.add(new Orange());
    for(Object apple : apples) {
      ((Apple) apple).id();
      // Orange is detected only at run time
    }
  }
}
/* Output:
___[ Error Output ]___
Exception in thread "main"
java.lang.ClassCastException: Orange cannot be cast to
Apple
        at ApplesAndOrangesWithoutGenerics.main(ApplesA
ndOrangesWithoutGenerics.java:23)
*/
```

The classes `Apple` and `Orange` are distinct; they have nothing in common except they are both `Objects`. (Remember that if you don't explicitly say what class you're inheriting from, you automatically inherit from `Object`.) `ArrayList` holds `Objects` so you can not only add `Apple` objects into this collection using the `ArrayList` method `add()`, but you can also add `Orange` objects without complaint at either

compile time or run time. When you go to fetch out what you think are `Apple` objects using the `ArrayList` method `get()`, you get back a reference to an `Object` you must cast to an `Apple`. Then you must surround the entire expression with parentheses to force the evaluation of the cast before calling the `id()` method for `Apple`; otherwise, you'll get a syntax error.

At run time, when you try to cast the `Orange` object to an `Apple`, you get the error as shown in the output.

In the Generics chapter, you'll learn that *creating* classes using Java generics can be complex. However, *applying* predefined generic classes is reasonably straightforward. For example, to define an `ArrayList` intended to hold `Apple` objects, you say `ArrayList<Apple>` instead of just `ArrayList`. The angle brackets surround the *type parameter(s)* (there might be more than one), which specify the type(s) that can be held by that instance of the collection.

With generics, you're prevented, *at compile time*, from putting the wrong type of object into a collection.[80] Here's the example again, using generics:

```
// collections/ApplesAndOrangesWithGenerics.java
import java.util.*;

public class ApplesAndOrangesWithGenerics {
  public static void main(String[] args) {
    ArrayList<Apple> apples = new ArrayList<>();
    for(int i = 0; i < 3; i++)
      apples.add(new Apple());
    // Compile-time error:
    // apples.add(new Orange());
    for(Apple apple : apples) {
      System.out.println(apple.id());
    }
  }
}
/* Output:
0
1
2
*/
```

[80] At the end of the Generics chapter, you'll find a discussion about whether this is such a bad problem. However, the Generics chapter also shows that Java generics are useful for more than just type-safe collections.

On the right-hand side of the definition for `apples`, you see `new ArrayList<>()`. This is sometimes called the "diamond syntax" because of the `<>`. Before Java 7, you had to, in effect, duplicate the type declaration on both sides, like this:

```
ArrayList<Apple> apples = new ArrayList<Apple>();
```

As types got more complex, this duplication produced code that was quite messy and hard to read. Programmers began observing that all the information is available on the left-hand side, so there was no reason for the compiler to force us to repeat ourselves on the right-hand side. This request for *type inference*, even on such a small scale, was heard by the Java language team.

With the type specified in the `ArrayList` definition, the compiler prevents you from putting an `Orange` into `apples`, so it becomes a compile-time error rather than a runtime error.

With generics, the cast is not necessary when fetching items from the `List`. The `List` knows what type it holds so it does the cast for you when you call `get()`. Thus, with generics you not only know that the compiler will check the type of object you put into a collection, but you also get cleaner syntax when using the objects in the collection.

You are not limited to putting the exact type of object into a collection when you specify that type as a generic parameter. Upcasting works the same with generics as it does with other types:

```
// collections/GenericsAndUpcasting.java
import java.util.*;

class GrannySmith extends Apple {}
class Gala extends Apple {}
class Fuji extends Apple {}
class Braeburn extends Apple {}

public class GenericsAndUpcasting {
  public static void main(String[] args) {
    ArrayList<Apple> apples = new ArrayList<>();
    apples.add(new GrannySmith());
    apples.add(new Gala());
    apples.add(new Fuji());
    apples.add(new Braeburn());
```

```
    for(Apple apple : apples)
      System.out.println(apple);
  }
}
/* Output:
GrannySmith@19e0bfd
Gala@139a55
Fuji@1db9742
Braeburn@106d69c
*/
```

Thus, you can add a subtype of `Apple` to a collection specified to hold `Apple` objects.

The output is produced from the default `toString()` method of `Object`, which prints the class name followed by the unsigned hexadecimal representation of the *hash code* of the object (generated by the `hashCode()` method). You'll learn about hash codes in detail in the Appendix: Understanding `equals()` and `hashCode()`.

Basic Concepts

The Java collection library takes the idea of "holding objects" and divides it into two distinct concepts, expressed as the basic interfaces of the library:

1. `Collection`: a sequence of individual elements with one or more rules applied to them. A `List` must hold the elements in the way they were inserted, a `Set` cannot have duplicate elements, and a `Queue` *produces* the elements in the order determined by a *queuing discipline* (usually the same order in which they are inserted).
2. `Map`: a group of key-value object pairs that looks up a value using a key. An `ArrayList` looks up an object using a number, so in a sense it associates numbers to objects. A *map* looks up an object using *another object*. It's also called an *associative array*, because it associates objects with other objects, or a *dictionary*, because you look up a value object using a key object just like you look up a definition using a word. `Maps` are powerful programming tools.

Although it's not always possible, ideally you'll write most of your code to talk to these interfaces, and the only place you'll specify the precise type you're using is at the point of creation. Thus, you can create a `List` like this:

```
List<Apple> apples = new ArrayList<>();
```

Notice that the `ArrayList` is upcast to a `List`, in contrast to the way it was handled in the previous examples. The intent of using the interface is that if you decide to change your implementation, you just change it at the point of creation, like this:

```
List<Apple> apples = new LinkedList<>();
```

Thus, you'll typically make an object of a concrete class, upcast it to the corresponding interface, then use the interface throughout the rest of your code.

This approach won't always work, because some classes have additional functionality. For example, `LinkedList` has additional methods not in the `List` interface, and a `TreeMap` has methods not in the `Map` interface. If you use those methods, you can't upcast to the more general interface.

The `Collection` interface generalizes the idea of a *sequence*—a way of holding a group of objects. Here's a simple example that fills a `Collection` (represented here with an `ArrayList`) with `Integer` objects, then prints each element in the resulting collection:

```
// collections/SimpleCollection.java
import java.util.*;

public class SimpleCollection {
  public static void main(String[] args) {
    Collection<Integer> c = new ArrayList<>();
    for(int i = 0; i < 10; i++)
      c.add(i); // Autoboxing
    for(Integer i : c)
      System.out.print(i + ", ");
  }
}
/* Output:
0, 1, 2, 3, 4, 5, 6, 7, 8, 9,
*/
```

This example only uses `Collection` methods, so any object of a class inherited from `Collection` would work. However, `ArrayList` is the most basic type of sequence.

The `add()` method name suggests that it puts a new element in the `Collection`. However, the documentation carefully states that `add()` "ensures this `Collection`

contains the specified element." This is to allow for the meaning of `Set`, which adds the element only if it isn't already there. With an `ArrayList`, or any sort of `List`, `add()` always means "put it in," because `Lists` don't care if there are duplicates.

All `Collections` can be traversed using the *for-in* syntax, as shown here. Later in this chapter you'll learn about a more flexible concept called an *Iterator*.

Adding Groups of Elements

The `Arrays` and `Collections` classes in `java.util` contain utility methods to add groups of elements to a `Collection`. `Arrays.asList()` takes either an array or a comma-separated list of elements (using varargs) and turns it into a `List` object. `Collections.addAll()` takes a `Collection` object and either an array or a comma-separated list and adds the elements to the `Collection`. Here you see both methods, as well as the more conventional `addAll()` method that's part of all `Collection` types:

```
// collections/AddingGroups.java
// Adding groups of elements to Collection objects
import java.util.*;

public class AddingGroups {
  public static void main(String[] args) {
    Collection<Integer> collection =
      new ArrayList<>(Arrays.asList(1, 2, 3, 4, 5));
    Integer[] moreInts = { 6, 7, 8, 9, 10 };
    collection.addAll(Arrays.asList(moreInts));
    // Runs significantly faster, but you can't
    // construct a Collection this way:
    Collections.addAll(collection, 11, 12, 13, 14, 15);
    Collections.addAll(collection, moreInts);
    // Produces a list "backed by" an array:
    List<Integer> list = Arrays.asList(16,17,18,19,20);
    list.set(1, 99); // OK -- modify an element
    // list.add(21); // Runtime error; the underlying
                     // array cannot be resized.
  }
}
```

The constructor for a `Collection` can accept another `Collection` which it uses for initializing itself, so you can use `Arrays.asList()` to produce input for the constructor. However, `Collections.addAll()` runs much faster, and it's just as easy to construct the `Collection` with no elements, then call `Collections.addAll()`, so this is the preferred approach.

The `Collection.addAll()` method can only take an argument of another `Collection` object, so it is not as flexible as `Arrays.asList()` or `Collections.addAll()`, which use variable argument lists.

It's also possible to use the output of `Arrays.asList()` directly, as a `List`, but the underlying representation here is the array, which cannot be resized. If you try to `add()` or `delete()` elements in such a list, that would attempt to change the size of an array, so you'll get an "Unsupported Operation" error at run time:

```
// collections/AsListInference.java
import java.util.*;

class Snow {}
class Powder extends Snow {}
class Light extends Powder {}
class Heavy extends Powder {}
class Crusty extends Snow {}
class Slush extends Snow {}

public class AsListInference {
  public static void main(String[] args) {
    List<Snow> snow1 = Arrays.asList(
      new Crusty(), new Slush(), new Powder());
    //- snow1.add(new Heavy()); // Exception

    List<Snow> snow2 = Arrays.asList(
      new Light(), new Heavy());
    //- snow2.add(new Slush()); // Exception

    List<Snow> snow3 = new ArrayList<>();
    Collections.addAll(snow3,
      new Light(), new Heavy(), new Powder());
    snow3.add(new Crusty());

    // Hint with explicit type argument specification:
```

```
    List<Snow> snow4 = Arrays.<Snow>asList(
       new Light(), new Heavy(), new Slush());
    //- snow4.add(new Powder()); // Exception
  }
}
```

In `snow4`, notice the "hint" in the middle of `Arrays.asList()`, to tell the compiler what the actual target type should be for the resulting `List` type produced by `Arrays.asList()`. This is called an *explicit type argument specification.*

Printing Collections

You must use `Arrays.toString()` to produce a printable representation of an array, but the collections print nicely without any help. Here's an example that also introduces you to the basic Java collections:

```
// collections/PrintingCollections.java
// Collections print themselves automatically
import java.util.*;

public class PrintingCollections {
  static Collection
  fill(Collection<String> collection) {
    collection.add("rat");
    collection.add("cat");
    collection.add("dog");
    collection.add("dog");
    return collection;
  }
  static Map fill(Map<String, String> map) {
    map.put("rat", "Fuzzy");
    map.put("cat", "Rags");
    map.put("dog", "Bosco");
    map.put("dog", "Spot");
    return map;
  }
  public static void main(String[] args) {
    System.out.println(fill(new ArrayList<>()));
    System.out.println(fill(new LinkedList<>()));
```

```
    System.out.println(fill(new HashSet<>()));
    System.out.println(fill(new TreeSet<>()));
    System.out.println(fill(new LinkedHashSet<>()));
    System.out.println(fill(new HashMap<>()));
    System.out.println(fill(new TreeMap<>()));
    System.out.println(fill(new LinkedHashMap<>()));
  }
}
/* Output:
[rat, cat, dog, dog]
[rat, cat, dog, dog]
[rat, cat, dog]
[cat, dog, rat]
[rat, cat, dog]
{rat=Fuzzy, cat=Rags, dog=Spot}
{cat=Rags, dog=Spot, rat=Fuzzy}
{rat=Fuzzy, cat=Rags, dog=Spot}
*/
```

This shows the two primary categories in the Java collection library. The distinction is based on the number of items held in each "slot" in the collection. The `Collection` category only holds one item in each slot. It includes `List`, which holds a group of items in a specified sequence, `Set`, which only allows the addition of one identical item, and `Queue`, which only inserts objects at one "end" of the collection and removes objects from the other "end" (for the purposes of this example, this is just another way of looking at a sequence and so is not shown). A `Map` holds two objects, a *key* and an associated *value*, in each slot.

The default printing behavior, provided via each collection's `toString()` method, produces reasonably readable results. A `Collection` is printed surrounded by square brackets, with each element separated by a comma. A `Map` is surrounded by curly braces, with each key and value associated with an equal sign (keys on the left, values on the right).

The first `fill()` method works with all types of `Collection`, each of which implements the `add()` method to include new elements.

`ArrayList` and `LinkedList` are both types of `List`, and the output shows that they both hold elements in insertion order. The difference between the two is not only performance for certain types of operations, but also that a `LinkedList` contains

more operations than an `ArrayList`. These are explored more fully later in this chapter.

`HashSet`, `TreeSet` and `LinkedHashSet` are types of `Set`. The output shows that a `Set` only holds one of each identical item, and that the different `Set` implementations store the elements differently. The `HashSet` stores elements using a rather complex approach, explored in the Appendix: Collection Topics. All you must know now is that this technique is the fastest way to retrieve elements, and as a result the storage order can seem nonsensical (you often care only that something is a member of the `Set`, and order is unimportant). If storage order is important, you can use a `TreeSet`, which keeps the objects in ascending comparison order, or a `LinkedHashSet`, which keeps the objects in the order in which they were added.

A `Map` (also called an *associative array*) looks up an object using a *key*, like a simple database. The associated object is called a *value*. If you have a `Map` that associates states with their capitals and you want the capital of Ohio, you look it up using "Ohio" as the key—almost as if you were indexing into an array. Because of this behavior, a `Map` only accepts one of each key.

`Map.put(key, value)` adds a value (what you want) and associates it with a key (how you look it up). `Map.get(key)` produces the value associated with that key. The above example only adds key-value pairs, and does not perform lookups. That is shown later.

Notice you don't specify (or think about) the size of the `Map` because it resizes itself automatically. Also, `Maps` know how to print themselves, showing the association with keys and values.

The example uses the three basic flavors of `Map`: `HashMap`, `TreeMap` and `LinkedHashMap`.

The order that the keys and values are held inside a `HashMap` is not the insertion order because the `HashMap` implementation uses a very fast algorithm that controls the order. A `TreeMap` keeps the keys sorted by ascending comparison order, and a `LinkedHashMap` keeps the keys in insertion order while retaining the lookup speed of the `HashMap`.

List

`Lists` promise to maintain elements in a particular sequence. The `List` interface adds a number of methods to `Collection` that allow insertion and removal of elements in the middle of a `List`.

There are two types of `List`:

- The basic `ArrayList`, which excels at randomly accessing elements, but is slower when inserting and removing elements in the middle of a `List`.
- The `LinkedList`, which provides optimal sequential access, with inexpensive insertions and deletions in the middle of the `List`. A `LinkedList` is relatively slow for random access, but it has a larger feature set than the `ArrayList`.

The following example reaches forward in the book to use a library from the Reflection chapter by importing `reflection.pets`. This is a library that contains a hierarchy of `Pet` classes along with some tools to randomly generate `Pet` objects. You don't need the full details, just that:

1. There's a `Pet` class and various subtypes of `Pet`.
2. `PetCreator.list()` returns an `ArrayList` filled with randomly selected `Pet` objects.

```
// collections/ListFeatures.java
import reflection.pets.*;
import java.util.*;

public class ListFeatures {
  public static void main(String[] args) {
    Random rand = new Random(47);
    List<Pet> pets = new PetCreator().list(7);
    System.out.println("1: " + pets);
    Hamster h = new Hamster();
    pets.add(h); // Automatically resizes
    System.out.println("2: " + pets);
    System.out.println("3: " + pets.contains(h));
    pets.remove(h); // Remove by object
    Pet p = pets.get(2);
    System.out.println(
```

```
    "4: " +  p + " " + pets.indexOf(p));
Pet cymric = new Cymric();
System.out.println("5: " + pets.indexOf(cymric));
System.out.println("6: " + pets.remove(cymric));
// Must be the exact object:
System.out.println("7: " + pets.remove(p));
System.out.println("8: " + pets);
pets.add(3, new Mouse()); // Insert at an index
System.out.println("9: " + pets);
List<Pet> sub = pets.subList(1, 4);
System.out.println("subList: " + sub);
System.out.println("10: " + pets.containsAll(sub));
Collections.sort(sub); // In-place sort
System.out.println("sorted subList: " + sub);
// Order is not important in containsAll():
System.out.println("11: " + pets.containsAll(sub));
Collections.shuffle(sub, rand); // Mix it up
System.out.println("shuffled subList: " + sub);
System.out.println("12: " + pets.containsAll(sub));
List<Pet> copy = new ArrayList<>(pets);
sub = Arrays.asList(pets.get(1), pets.get(4));
System.out.println("sub: " + sub);
copy.retainAll(sub);
System.out.println("13: " + copy);
copy = new ArrayList<>(pets); // Get a fresh copy
copy.remove(2); // Remove by index
System.out.println("14: " + copy);
copy.removeAll(sub); // Only removes exact objects
System.out.println("15: " + copy);
copy.set(1, new Mouse()); // Replace an element
System.out.println("16: " + copy);
copy.addAll(2, sub); // Insert a list in the middle
System.out.println("17: " + copy);
System.out.println("18: " + pets.isEmpty());
pets.clear(); // Remove all elements
System.out.println("19: " + pets);
System.out.println("20: " + pets.isEmpty());
pets.addAll(new PetCreator().list(4));
System.out.println("21: " + pets);
Object[] o = pets.toArray();
System.out.println("22: " + o[3]);
Pet[] pa = pets.toArray(new Pet[0]);
```

```
    System.out.println("23: " + pa[3].id());
  }
}
/* Output:
1: [Rat, Manx, Cymric, Mutt, Pug, Cymric, Pug]
2: [Rat, Manx, Cymric, Mutt, Pug, Cymric, Pug, Hamster]
3: true
4: Cymric 2
5: -1
6: false
7: true
8: [Rat, Manx, Mutt, Pug, Cymric, Pug]
9: [Rat, Manx, Mutt, Mouse, Pug, Cymric, Pug]
subList: [Manx, Mutt, Mouse]
10: true
sorted subList: [Manx, Mouse, Mutt]
11: true
shuffled subList: [Mouse, Manx, Mutt]
12: true
sub: [Mouse, Pug]
13: [Mouse, Pug]
14: [Rat, Mouse, Mutt, Pug, Cymric, Pug]
15: [Rat, Mutt, Cymric, Pug]
16: [Rat, Mouse, Cymric, Pug]
17: [Rat, Mouse, Mouse, Pug, Cymric, Pug]
18: false
19: []
20: true
21: [Rat, Manx, Cymric, Mutt]
22: Mutt
23: 14
*/
```

The print lines are numbered so the output can be related to the source code. The first output line shows the original `List` of `Pets`. Unlike an array, a `List` can add or remove elements after creation, and it resizes itself. That's its fundamental value: a modifiable sequence. You see the result of adding a `Hamster` in output line 2—the object is appended to the end of the list.

You can find out whether an object is in the list using the `contains()` method. To remove an object, you can pass that object's reference to the `remove()` method. Also,

with a reference to an object, you can discover the index number where that object is located in the `List` using `indexOf()`, as you see in output line 4.

When deciding whether an element is part of a `List`, discovering the index of an element, and removing an element from a `List` by reference, the `equals()` method (part of the root class `Object`) is used. Each `Pet` is defined as a unique object, so even though there are two `Cymrics` in the list, if I create a new `Cymric` object and pass it to `indexOf()`, the result is `-1` (indicating it wasn't found), and attempts to `remove()` the object will return `false`. For other classes, `equals()` can be defined differently—`Strings`, for example, are equal if the contents of two `Strings` are identical. So to prevent surprises, it's important to be aware that `List` behavior changes depending on `equals()` behavior.

Output lines 7 and 8 show the success of removing an object that exactly matches an object in the `List`.

It's possible to insert an element in the middle of the `List`, as in output line 9 and the code that precedes it, but this brings up an issue: for a `LinkedList`, insertion and removal in the middle of a list is a cheap operation (except for, in this case, the actual random access into the middle of the list), but for an `ArrayList` it is an expensive operation. Does this mean you should never insert elements in the middle of an `ArrayList`, and switch to a `LinkedList` if you do? No, it just means you should be aware of the issue, and if you start doing many insertions in the middle of an `ArrayList` *and* your program starts slowing down, you might look at your `List` implementation as the possible culprit (the best way to discover such a bottleneck is to use a profiler). Optimization is a tricky issue, and the best policy is to leave it alone until you discover you must worry about it (although understanding the issues is always a good idea).

The `subList()` method easily creates a slice out of a larger list, and this naturally produces a `true` result when passed to `containsAll()` for that larger list. Notice that order is unimportant—you see in output lines 11 and 12 that calling the intuitively named `Collections.sort()` and `Collections.shuffle()` on `sub` doesn't affect the outcome of `containsAll()`. `subList()` produces a list backed by the original list. Therefore, changes in the returned list are reflected in the original list, and vice versa.

The `retainAll()` method is effectively a "set intersection" operation, in this case keeping all the elements in `copy` that are also in `sub`. Again, the resulting behavior

depends on the `equals()` method.

Output line 14 shows the result of removing an element using its index number, which is more straightforward than removing it by object reference because you don't worry about `equals()` behavior when using indexes.

The `removeAll()` method also operates based on the `equals()` method. As the name implies, it removes all the objects from the `List` that are in the argument `List`.

The `set()` method is rather unfortunately named because of the potential confusion with the `Set` class—"replace" might be a better name here, because it replaces the element at the index (the first argument) with the second argument.

Output line 17 shows that for `Lists`, there's an overloaded `addAll()` method that inserts the new list in the middle of the original list, instead of just appending it to the end with the `addAll()` that comes from `Collection`.

Output lines 18-20 show the effect of the `isEmpty()` and `clear()` methods.

Output lines 22 and 23 show how you can convert any `Collection` to an array using `toArray()`. This is an overloaded method; the zero-argument version returns an array of `Object`, but if you pass an array of the target type to the overloaded version, it produces an array of the type specified (assuming it passes type checking). If the argument array is too small to hold all the objects in the `List` (as is the case here), `toArray()` creates a new array of the appropriate size. `Pet` objects have an `id()` method, which you see is called on one of the objects in the resulting array.

Iterators

In any collection, you must have a way to insert elements and fetch them out again. After all, that's the primary job of a collection—to hold things. In a `List`, `add()` is one way to insert elements, and `get()` is one way to fetch elements.

When you start thinking at a higher level, there's a drawback: You must program to the exact type of the collection to use it. This might not seem bad at first, but what if you write code for a `List`, and later on you discover it would be convenient to apply that same code to a `Set`? Or suppose you'd like to write, from the beginning, a piece of general-purpose code that doesn't know or care what type of collection it's

working with, so it can be used on different types of collections without rewriting that code?

The concept of an *Iterator* (another design pattern) achieves this abstraction. An iterator is an object that moves through a sequence and selects each object in that sequence without the client programmer knowing or caring about the underlying structure of that sequence. In addition, an iterator is usually what's called a *lightweight object*: one that's cheap to create. For that reason, you'll often find seemingly strange constraints for iterators; for example, the Java `Iterator` can move in only one direction. There's not much you can do with an `Iterator` except:

1. Ask a `Collection` to hand you an `Iterator` using a method called `iterator()`. That `Iterator` is ready to return the first element in the sequence.
2. Get the next object in the sequence with `next()`.
3. See if there are any more objects in the sequence with `hasNext()`.
4. Remove the last element returned by the iterator with `remove()`.

To see how it works, we again use the `Pet` tools from the Reflection chapter:

```
// collections/SimpleIteration.java
import reflection.pets.*;
import java.util.*;

public class SimpleIteration {
  public static void main(String[] args) {
    List<Pet> pets = new PetCreator().list(12);
    Iterator<Pet> it = pets.iterator();
    while(it.hasNext()) {
      Pet p = it.next();
      System.out.print(p.id() + ":" + p + " ");
    }
    System.out.println();
    // A simpler approach, when possible:
    for(Pet p : pets)
      System.out.print(p.id() + ":" + p + " ");
    System.out.println();
    // An Iterator can also remove elements:
    it = pets.iterator();
    for(int i = 0; i < 6; i++) {
      it.next();
      it.remove();
```

```
    }
    System.out.println(pets);
  }
}
/* Output:
0:Rat 1:Manx 2:Cymric 3:Mutt 4:Pug 5:Cymric 6:Pug
7:Manx 8:Cymric 9:Rat 10:EgyptianMau 11:Hamster
0:Rat 1:Manx 2:Cymric 3:Mutt 4:Pug 5:Cymric 6:Pug
7:Manx 8:Cymric 9:Rat 10:EgyptianMau 11:Hamster
[Pug, Manx, Cymric, Rat, EgyptianMau, Hamster]
*/
```

With an `Iterator`, you don't worry about the number of elements in the collection. That's taken care of for you by `hasNext()` and `next()`.

If you're moving forward through the `List` and not trying to modify the `List` object itself, you see that the *for-in* syntax is more succinct.

An `Iterator` can also remove the last element produced by `next()`, which means you must call `next()` before you call `remove()`.[81]

This idea performing an operation on each object in a collection is powerful and is seen throughout this book.

Now consider the creation of a `display()` method that is collection-agnostic:

```
// collections/CrossCollectionIteration.java
import reflection.pets.*;
import java.util.*;

public class CrossCollectionIteration {
  public static void display(Iterator<Pet> it) {
    while(it.hasNext()) {
      Pet p = it.next();
      System.out.print(p.id() + ":" + p + " ");
    }
    System.out.println();
  }
  public static void main(String[] args) {
    List<Pet> pets = new PetCreator().list(8);
```

[81] `remove()` is a so-called "optional" method (there are other such methods), which means not all `Iterator` implementations must implement it. This topic is covered in the Appendix: Collection Topics. The standard Java library collections implement `remove()`, however, so you don't need to worry about it until that chapter.

```
    LinkedList<Pet> petsLL = new LinkedList<>(pets);
    HashSet<Pet> petsHS = new HashSet<>(pets);
    TreeSet<Pet> petsTS = new TreeSet<>(pets);
    display(pets.iterator());
    display(petsLL.iterator());
    display(petsHS.iterator());
    display(petsTS.iterator());
  }
}
/* Output:
0:Rat 1:Manx 2:Cymric 3:Mutt 4:Pug 5:Cymric 6:Pug
7:Manx
0:Rat 1:Manx 2:Cymric 3:Mutt 4:Pug 5:Cymric 6:Pug
7:Manx
0:Rat 1:Manx 2:Cymric 3:Mutt 4:Pug 5:Cymric 6:Pug
7:Manx
5:Cymric 2:Cymric 7:Manx 1:Manx 3:Mutt 6:Pug 4:Pug
0:Rat
*/
```

`display()` contains no information about the type of sequence it is traversing. This shows the true power of the `Iterator`: the ability to separate the operation of traversing a sequence from the underlying structure of that sequence. For this reason, we sometimes say that iterators *unify access to collections.*

We can produce a cleaner version of the previous example by using the `Iterable` interface, which describes "anything that can produce an `Iterator`":

```
// collections/CrossCollectionIteration2.java
import reflection.pets.*;
import java.util.*;

public class CrossCollectionIteration2 {
  public static void display(Iterable<Pet> ip) {
    Iterator<Pet> it = ip.iterator();
    while(it.hasNext()) {
      Pet p = it.next();
      System.out.print(p.id() + ":" + p + " ");
    }
    System.out.println();
  }
  public static void main(String[] args) {
```

```
    List<Pet> pets = new PetCreator().list(8);
    LinkedList<Pet> petsLL = new LinkedList<>(pets);
    HashSet<Pet> petsHS = new HashSet<>(pets);
    TreeSet<Pet> petsTS = new TreeSet<>(pets);
    display(pets);
    display(petsLL);
    display(petsHS);
    display(petsTS);
  }
}
/* Output:
0:Rat 1:Manx 2:Cymric 3:Mutt 4:Pug 5:Cymric 6:Pug
7:Manx
0:Rat 1:Manx 2:Cymric 3:Mutt 4:Pug 5:Cymric 6:Pug
7:Manx
0:Rat 1:Manx 2:Cymric 3:Mutt 4:Pug 5:Cymric 6:Pug
7:Manx
5:Cymric 2:Cymric 7:Manx 1:Manx 3:Mutt 6:Pug 4:Pug
0:Rat
*/
```

All of the classes are `Iterable`, so now the calls to `display()` are notably simpler.

`ListIterator`

`ListIterator` is a more powerful subtype of `Iterator` that is produced only by `List` classes. While `Iterator` can only move forward, `ListIterator` is bidirectional. It can produce indices of the next and previous elements relative to where the iterator is pointing in the list, and it can replace the last element it visited using the `set()` method. You can produce a `ListIterator` that points to the beginning of the `List` by calling `listIterator()`, and you can also create a `ListIterator` that starts out pointing to an index `n` in the list by calling `listIterator(n)`. Here's demonstration of these abilities:

```
// collections/ListIteration.java
import reflection.pets.*;
import java.util.*;

public class ListIteration {
  public static void main(String[] args) {
    List<Pet> pets = new PetCreator().list(8);
    ListIterator<Pet> it = pets.listIterator();
    while(it.hasNext())
      System.out.print(it.next() +
        ", " + it.nextIndex() +
        ", " + it.previousIndex() + "; ");
    System.out.println();
    // Backwards:
    while(it.hasPrevious())
      System.out.print(it.previous().id() + " ");
    System.out.println();
    System.out.println(pets);
    it = pets.listIterator(3);
    while(it.hasNext()) {
      it.next();
      it.set(new PetCreator().get());
    }
    System.out.println(pets);
  }
}
/* Output:
Rat, 1, 0; Manx, 2, 1; Cymric, 3, 2; Mutt, 4, 3; Pug,
5, 4; Cymric, 6, 5; Pug, 7, 6; Manx, 8, 7;
7 6 5 4 3 2 1 0
[Rat, Manx, Cymric, Mutt, Pug, Cymric, Pug, Manx]
[Rat, Manx, Cymric, Rat, Rat, Rat, Rat, Rat]
*/
```

The `PetCreator.get()` method is used to replace all the `Pet` objects in the `List` from location 3 onward.

`LinkedList`

`LinkedList` implements the basic `List` interface like `ArrayList` does, but it performs insertion and removal in the middle of the `List` more efficiently than

`ArrayList`. However, it is less efficient for random-access operations.

`LinkedList` adds methods to use it as a stack, a queue or a double-ended queue (deque). Some of these methods are aliases or slight variations of each other, producing names familiar for a particular use (`Queue`, in particular).

- `getFirst()` and `element()` are identical—they return the head (first element) of the list without removing it, and throw `NoSuchElementException` if the `List` is empty. `peek()` is a slight variation of those two that returns `null` if the list is empty.
- `removeFirst()` and `remove()` are also identical—they remove and return the head of the list, and throw `NoSuchElementException` for an empty list. `poll()` is a slight variation that returns `null` if this list is empty.
- `addFirst()` inserts an element at the beginning of the list.
- `offer()` is the same as `add()` and `addLast()`. They all add an element to the tail (end) of a list.
- `removeLast()` removes and returns the last element of the list.

Here's an example that shows the basic similarity and differences between these features. It doesn't repeat the behavior that was shown in `ListFeatures.java`:

```
// collections/LinkedListFeatures.java
import reflection.pets.*;
import java.util.*;

public class LinkedListFeatures {
  public static void main(String[] args) {
    LinkedList<Pet> pets =
      new LinkedList<>(new PetCreator().list(5));
    System.out.println(pets);
    // Identical:
    System.out.println(
      "pets.getFirst(): " + pets.getFirst());
    System.out.println(
      "pets.element(): " + pets.element());
    // Only differs in empty-list behavior:
    System.out.println("pets.peek(): " + pets.peek());
    // Identical; remove and return the first element:
    System.out.println(
      "pets.remove(): " + pets.remove());
```

```
    System.out.println(
      "pets.removeFirst(): " + pets.removeFirst());
    // Only differs in empty-list behavior:
    System.out.println("pets.poll(): " + pets.poll());
    System.out.println(pets);
    pets.addFirst(new Rat());
    System.out.println("After addFirst(): " + pets);
    pets.offer(new PetCreator().get());
    System.out.println("After offer(): " + pets);
    pets.add(new PetCreator().get());
    System.out.println("After add(): " + pets);
    pets.addLast(new Hamster());
    System.out.println("After addLast(): " + pets);
    System.out.println(
      "pets.removeLast(): " + pets.removeLast());
  }
}
/* Output:
[Rat, Manx, Cymric, Mutt, Pug]
pets.getFirst(): Rat
pets.element(): Rat
pets.peek(): Rat
pets.remove(): Rat
pets.removeFirst(): Manx
pets.poll(): Cymric
[Mutt, Pug]
After addFirst(): [Rat, Mutt, Pug]
After offer(): [Rat, Mutt, Pug, Rat]
After add(): [Rat, Mutt, Pug, Rat, Rat]
After addLast(): [Rat, Mutt, Pug, Rat, Rat, Hamster]
pets.removeLast(): Hamster
*/
```

The result of `PetCreator.list()` is handed to the `LinkedList` constructor to populate it. If you look at the `Queue` interface, you'll see the `element()`, `offer()`, `peek()`, `poll()` and `remove()` methods that were added to `LinkedList` so it could be a `Queue` implementation. Full examples of `Queues` are given later in this chapter.

Stack

A stack is a "last-in, first-out" (LIFO) collection. It's sometimes called a *pushdown stack*, because whatever you "push" on the stack last is the first item you can "pop" off of the stack. An often-used analogy is cafeteria trays in a spring-loaded holder—the last ones that go in are the first ones that come out.

Java 1.0 came with a `Stack` class that turned out to be badly designed (for backward compatibility, we are forever stuck with old design mistakes in Java). Java 6 added `ArrayDeque`, with methods that directly implement stack functionality:

```
// collections/StackTest.java
import java.util.*;

public class StackTest {
  public static void main(String[] args) {
    Deque<String> stack = new ArrayDeque<>();
    for(String s : "My dog has fleas".split(" "))
      stack.push(s);
    while(!stack.isEmpty())
      System.out.print(stack.pop() + " ");
  }
}
/* Output:
fleas has dog My
*/
```

Even though it acts in all ways as a stack, we must still declare it as a `Deque`. Suppose that a class named `Stack` tells the story better:

```
// onjava/Stack.java
// A Stack class built with an ArrayDeque
package onjava;
import java.util.Deque;
import java.util.ArrayDeque;

public class Stack<T> {
  private Deque<T> storage = new ArrayDeque<>();
  public void push(T v) { storage.push(v); }
  public T peek() { return storage.peek(); }
  public T pop() { return storage.pop(); }
  public boolean isEmpty() { return storage.isEmpty(); }
  @Override public String toString() {
    return storage.toString();
  }
}
```

This introduces the simplest possible example of a class definition using generics. The `<T>` after the class name tells the compiler this is a *parameterized type*, and that the type parameter—which is substituted with a real type when the class is used—is `T`. Basically, this says, "We're defining a `Stack` that holds objects of type `T`." The `Stack` is implemented using an `ArrayDeque`, which also holds type `T`. Notice that `push()` takes an object of type `T`, while `peek()` and `pop()` return an object of type `T`. The `peek()` method provides you with the top element without removing it from the top of the stack, while `pop()` removes and returns the top element.

If you want only stack behavior, inheritance is inappropriate here because that would produce a class with all the rest of the `ArrayDeque` methods (you'll see in the Appendix: Collection Topics that this very mistake was made by the Java 1.0 designers when they created `java.util.Stack`). Using composition, we choose which methods to expose and how to name them.

We'll use the same code from `StackTest.java` to demonstrate this new `Stack` class:

```
// collections/StackTest2.java
import onjava.*;

public class StackTest2 {
  public static void main(String[] args) {
    Stack<String> stack = new Stack<>();
    for(String s : "My dog has fleas".split(" "))
      stack.push(s);
    while(!stack.isEmpty())
      System.out.print(stack.pop() + " ");
  }
}
/* Output:
fleas has dog My
*/
```

To use this `Stack` in your own code, you fully specify the package—or change the name of the class—when you create one; otherwise, you'll probably collide with the `Stack` in `java.util`. For example, if we `import java.util.*` into the above example, we must use package names to prevent collisions:

```
// collections/StackCollision.java

public class StackCollision {
  public static void main(String[] args) {
    onjava.Stack<String> stack = new onjava.Stack<>();
    for(String s : "My dog has fleas".split(" "))
      stack.push(s);
    while(!stack.isEmpty())
      System.out.print(stack.pop() + " ");
    System.out.println();
    java.util.Stack<String> stack2 =
      new java.util.Stack<>();
    for(String s : "My dog has fleas".split(" "))
      stack2.push(s);
    while(!stack2.empty())
      System.out.print(stack2.pop() + " ");
  }
}
/* Output:
fleas has dog My
```

```
fleas has dog My
*/
```

Even though `java.util.Stack` exists, `ArrayDeque` produces a much better `Stack` and so is preferable.

You can also control the selection of the "preferred" `Stack` implementation using an explicit import:

```
import onjava.Stack;
```

Now any reference to `Stack` will select the `onjava` version, and to select `java.util.Stack` you must use full qualification.

Set

A `Set` refuses to hold more than one instance of each object value. If you try to add more than one instance of an equivalent object, the `Set` prevents duplication. The most common use for a `Set` is to test for membership, so you can easily ask whether an object is in a `Set`. Because of this, lookup is typically the most important operation for a `Set`, so you'll usually choose a `HashSet` implementation, which is optimized for rapid lookup.

`Set` has the same interface as `Collection`, so there isn't any extra functionality like there is in the two different types of `List`. Instead, the `Set` is exactly a `Collection`—it just has different behavior. (This is the ideal use of inheritance and polymorphism: to express different behavior.) A `Set` determines membership based on the "value" of an object, a more complex topic covered in the Appendix: Collection Topics.

This uses a `HashSet` with `Integer` objects:

```
// collections/SetOfInteger.java
import java.util.*;

public class SetOfInteger {
  public static void main(String[] args) {
    Random rand = new Random(47);
    Set<Integer> intset = new HashSet<>();
    for(int i = 0; i < 10000; i++)
      intset.add(rand.nextInt(30));
    System.out.println(intset);
  }
}
/* Output:
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15,
16, 17, 18, 19, 20, 21, 22, 23, 24, 25, 26, 27, 28, 29]
*/
```

Ten thousand random numbers from 0 up to 29 are added to the `Set`, so you can imagine that each value has many duplications. And yet you see that only one instance of each appears in the result.

The `HashSet` in earlier versions of Java produced output no discernible order. A `HashSet` uses *hashing* for speed—hashing is covered in the Appendix: Collection Topics. The order maintained by a `HashSet` is different from a `TreeSet` or a `LinkedHashSet` because each implementation has a different way of storing elements. `TreeSet` keeps elements sorted into a red-black tree data structure, whereas `HashSet` uses the hashing function. `LinkedHashSet` also uses hashing for lookup speed, but *appears* to maintain elements in insertion order using a linked list. Apparently, the hashing algorithm was changed and now `Integer`s end up in sorted order. However, you should not depend on this behavior:

```
// collections/SetOfString.java
import java.util.*;

public class SetOfString {
  public static void main(String[] args) {
    Set<String> colors = new HashSet<>();
    for(int i = 0; i < 100; i++) {
      colors.add("Yellow");
      colors.add("Blue");
      colors.add("Red");
      colors.add("Red");
      colors.add("Orange");
      colors.add("Yellow");
      colors.add("Blue");
      colors.add("Purple");
    }
    System.out.println(colors);
  }
}
/* Output:
[Red, Yellow, Blue, Purple, Orange]
*/
```

`String` objects don't seem to produce an order. To sort the results, one approach is to use a `TreeSet` instead of a `HashSet`:

```
// collections/SortedSetOfString.java
import java.util.*;

public class SortedSetOfString {
  public static void main(String[] args) {
    Set<String> colors = new TreeSet<>();
    for(int i = 0; i < 100; i++) {
      colors.add("Yellow");
      colors.add("Blue");
      colors.add("Red");
      colors.add("Red");
      colors.add("Orange");
      colors.add("Yellow");
      colors.add("Blue");
      colors.add("Purple");
    }
```

```
    System.out.println(colors);
  }
}
/* Output:
[Blue, Orange, Purple, Red, Yellow]
*/
```

One of the most common operations is a test for set membership using `contains()`, but there are also operations like the Venn diagrams you might have been taught in elementary school:

```
// collections/SetOperations.java
import java.util.*;

public class SetOperations {
  public static void main(String[] args) {
    Set<String> set1 = new HashSet<>();
    Collections.addAll(set1,
      "A B C D E F G H I J K L".split(" "));
    set1.add("M");
    System.out.println("H: " + set1.contains("H"));
    System.out.println("N: " + set1.contains("N"));
    Set<String> set2 = new HashSet<>();
    Collections.addAll(set2, "H I J K L".split(" "));
    System.out.println(
      "set2 in set1: " + set1.containsAll(set2));
    set1.remove("H");
    System.out.println("set1: " + set1);
    System.out.println(
      "set2 in set1: " + set1.containsAll(set2));
    set1.removeAll(set2);
    System.out.println(
      "set2 removed from set1: " + set1);
    Collections.addAll(set1, "X Y Z".split(" "));
    System.out.println(
      "'X Y Z' added to set1: " + set1);
  }
}
/* Output:
H: true
N: false
set2 in set1: true
```

```
set1: [A, B, C, D, E, F, G, I, J, K, L, M]
set2 in set1: false
set2 removed from set1: [A, B, C, D, E, F, G, M]
'X Y Z' added to set1: [A, B, C, D, E, F, G, M, X, Y,
Z]
*/
```

The method names are self-explanatory, and there are a few more in the JDK documentation.

Producing a list of unique elements can be useful. For example, suppose you'd like to list all the words in the file `SetOperations.java`, above. Using the `java.nio.file.Files.rea` method introduced later in the book, you can open a file and read it as a `List<String>`, with each `String` a line from the input file:

```
// collections/UniqueWords.java
import java.util.*;
import java.nio.file.*;

public class UniqueWords {
  public static void
  main(String[] args) throws Exception {
    List<String> lines = Files.readAllLines(
      Paths.get("SetOperations.java"));
    Set<String> words = new TreeSet<>();
    for(String line : lines)
      for(String word : line.split("\\W+"))
        if(word.trim().length() > 0)
          words.add(word);
    System.out.println(words);
  }
}
/* Output:
[A, B, C, Collections, D, E, F, G, H, HashSet, I, J, K,
L, M, N, Output, Set, SetOperations, String, System, X,
Y, Z, add, addAll, added, args, class, collections,
contains, containsAll, false, from, import, in, java,
main, new, out, println, public, remove, removeAll,
removed, set1, set2, split, static, to, true, util,
void]
*/
```

We step through each line in the file and break it into words using `String.split()`, using the *regular expression* `\\W+`, which means it splits on one or more (that's the `+`) *non*-word letters (regular expressions are introduced in the Strings chapter). Each resulting word is added to the `words Set`. It is a `TreeSet` so the result is sorted. Here, the sorting is done *lexicographically* so the uppercase and lowercase letters are in separate groups. If you'd like to sort it *alphabetically*, you can pass the `String.CASE_INSENSITIVE_ORDER Comparator` (a *comparator* is an object that establishes order) to the `TreeSet` constructor:

```
// collections/UniqueWordsAlphabetic.java
// Producing an alphabetic listing
import java.util.*;
import java.nio.file.*;

public class UniqueWordsAlphabetic {
  public static void
  main(String[] args) throws Exception {
    List<String> lines = Files.readAllLines(
      Paths.get("SetOperations.java"));
    Set<String> words =
      new TreeSet<>(String.CASE_INSENSITIVE_ORDER);
    for(String line : lines)
      for(String word : line.split("\\W+"))
        if(word.trim().length() > 0)
          words.add(word);
    System.out.println(words);
  }
}
/* Output:
[A, add, addAll, added, args, B, C, class, collections,
contains, containsAll, D, E, F, false, from, G, H,
HashSet, I, import, in, J, java, K, L, M, main, N, new,
out, Output, println, public, remove, removeAll,
removed, Set, set1, set2, SetOperations, split, static,
String, System, to, true, util, void, X, Y, Z]
*/
```

`Comparators` are explored in detail in the Arrays chapter.

Map

The ability to map objects to other objects is a powerful way to solve programming problems. For example, consider a program to examine the randomness of Java's `Random` class. Ideally, `Random` would produce a perfect distribution of numbers, but to test this you must generate many random numbers and count the ones that fall in the various ranges. A `Map` easily solves the problem. Here, the key is the number produced by `Random`, and the value is the number of times that number appears:

```
// collections/Statistics.java
// Simple demonstration of HashMap
import java.util.*;

public class Statistics {
  public static void main(String[] args) {
    Random rand = new Random(47);
    Map<Integer, Integer> m = new HashMap<>();
    for(int i = 0; i < 10000; i++) {
      // Produce a number between 0 and 20:
      int r = rand.nextInt(20);
      Integer freq = m.get(r);               // [1]
      m.put(r, freq == null ? 1 : freq + 1);
    }
    System.out.println(m);
  }
}
/* Output:
{0=481, 1=502, 2=489, 3=508, 4=481, 5=503, 6=519,
7=471, 8=468, 9=549, 10=513, 11=531, 12=521, 13=506,
14=477, 15=497, 16=533, 17=509, 18=478, 19=464}
*/
```

- **[1]** Autoboxing converts the randomly generated `int` into an `Integer` reference that can be used with the `HashMap` (you can't use primitives with collections). `get()` returns `null` if the key is not already in the collection (which means this is the first time the number was found). Otherwise, `get()` produces the associated `Integer` value for the key, which is incremented (again, autoboxing simplifies the expression but there are actually conversions to and from `Integer` taking place).

Next, we'll use a `String` description to look up `Pet` objects. This also shows how you can test a `Map` to see if it contains a key or a value with `containsKey()` and `containsValue()`:

```
// collections/PetMap.java
import reflection.pets.*;
import java.util.*;

public class PetMap {
  public static void main(String[] args) {
    Map<String, Pet> petMap = new HashMap<>();
    petMap.put("My Cat", new Cat("Molly"));
    petMap.put("My Dog", new Dog("Ginger"));
    petMap.put("My Hamster", new Hamster("Bosco"));
    System.out.println(petMap);
    Pet dog = petMap.get("My Dog");
    System.out.println(dog);
    System.out.println(petMap.containsKey("My Dog"));
    System.out.println(petMap.containsValue(dog));
  }
}
/* Output:
{My Dog=Dog Ginger, My Cat=Cat Molly, My
Hamster=Hamster Bosco}
Dog Ginger
true
true
*/
```

`Maps`, like arrays and `Collections`, can easily be expanded to multiple dimensions; you make a `Map` whose values are `Maps` (and the values of *those* `Maps` can be other collections, even other `Maps`). Thus, it's easy to combine collections to quickly produce powerful data structures. For example, suppose you are keeping track of people who have multiple pets—all you need is a `Map<Person, List<Pet>>`:

```
// collections/MapOfList.java
// {java collections.MapOfList}
package collections;
import reflection.pets.*;
import java.util.*;

public class MapOfList {
  public static final Map<Person, List< ? extends Pet>>
    petPeople = new HashMap<>();
  static {
    petPeople.put(new Person("Dawn"),
      Arrays.asList(
        new Cymric("Molly"),
        new Mutt("Spot")));
    petPeople.put(new Person("Kate"),
      Arrays.asList(new Cat("Shackleton"),
        new Cat("Elsie May"), new Dog("Margrett")));
    petPeople.put(new Person("Marilyn"),
      Arrays.asList(
        new Pug("Louie aka Louis Snorkelstein Dupree"),
        new Cat("Stanford"),
        new Cat("Pinkola")));
    petPeople.put(new Person("Luke"),
      Arrays.asList(
        new Rat("Fuzzy"), new Rat("Fizzy")));
    petPeople.put(new Person("Isaac"),
      Arrays.asList(new Rat("Freckly")));
  }
  public static void main(String[] args) {
    System.out.println("People: " + petPeople.keySet());
    System.out.println("Pets: " + petPeople.values());
    for(Person person : petPeople.keySet()) {
      System.out.println(person + " has:");
      for(Pet pet : petPeople.get(person))
        System.out.println("    " + pet);
    }
  }
}
/* Output:
People: [Person Dawn, Person Kate, Person Isaac, Person
Marilyn, Person Luke]
Pets: [[Cymric Molly, Mutt Spot], [Cat Shackleton, Cat
```

```
Elsie May, Dog Margrett], [Rat Freckly], [Pug Louie aka
Louis Snorkelstein Dupree, Cat Stanford, Cat Pinkola],
[Rat Fuzzy, Rat Fizzy]]
Person Dawn has:
    Cymric Molly
    Mutt Spot
Person Kate has:
    Cat Shackleton
    Cat Elsie May
    Dog Margrett
Person Isaac has:
    Rat Freckly
Person Marilyn has:
    Pug Louie aka Louis Snorkelstein Dupree
    Cat Stanford
    Cat Pinkola
Person Luke has:
    Rat Fuzzy
    Rat Fizzy
*/
```

A `Map` can return a `Set` of its keys, a `Collection` of its values, or a `Set` of its pairs. The `keySet()` method produces a `Set` of all the keys in `petPeople`, used in the *for-in* statement to iterate through the `Map`.

Queue

A *queue* is typically a *first-in-first-out* (FIFO) collection. That is, you put things in at one end and pull them out at the other, and the order you put them in is the same order they come out. Queues are commonly used as a way to reliably transfer objects from one area of a program to another. Queues are especially important in Concurrent Programming, because they safely transfer objects from one task to another.

`LinkedList` implements the `Queue` interface with methods to support queue behavior, so a `LinkedList` can be used as a `Queue` implementation. By upcasting a `LinkedList` to a `Queue`, this example uses the `Queue`-specific methods in the `Queue` interface:

```
// collections/QueueDemo.java
// Upcasting to a Queue from a LinkedList
import java.util.*;

public class QueueDemo {
  public static void printQ(Queue queue) {
    while(queue.peek() != null)
      System.out.print(queue.remove() + " ");
    System.out.println();
  }
  public static void main(String[] args) {
    Queue<Integer> queue = new LinkedList<>();
    Random rand = new Random(47);
    for(int i = 0; i < 10; i++)
      queue.offer(rand.nextInt(i + 10));
    printQ(queue);
    Queue<Character> qc = new LinkedList<>();
    for(char c : "Brontosaurus".toCharArray())
      qc.offer(c);
    printQ(qc);
  }
}
/* Output:
8 1 1 1 5 14 3 1 0 1
B r o n t o s a u r u s
*/
```

`offer()` is a Queue-specific method that inserts an element at the tail of the queue if it can, or returns `false`. Both `peek()` and `element()` return the head of the queue *without removing it*, but `peek()` returns `null` if the queue is empty and `element()` throws `NoSuchElementException`. Both `poll()` and `remove()` remove and return the head of the queue, but `poll()` returns `null` if the queue is empty, while `remove()` throws `NoSuchElementException`.

Autoboxing automatically converts the `int` result of `nextInt()` into the `Integer` object required by `queue`, and the `char c` into the `Character` object required by `qc`. The `Queue` interface narrows access to the methods of `LinkedList` so only the appropriate methods are available, and you are thus less tempted to use `LinkedList` methods (here, you can actually cast `queue` back to a `LinkedList`, but you are at least discouraged from doing so).

The `Queue`-specific methods provide complete and standalone functionality. That is, you can have a usable `Queue` without any of the methods in `Collection`, from which it is inherited.

`PriorityQueue`

First-in, first-out (FIFO) describes the most typical *queuing discipline*. A queuing discipline decides, given a group of elements in the queue, which one goes next. First-in, first-out says that the next element should be the one that was waiting the longest.

A *priority queue* says that the element that goes next is the one with the greatest need (the highest priority). For example, in an airport, a customer might be pulled out of a queue if their plane is about to leave. If you build a messaging system, some messages are more important than others, and should be dealt with sooner, regardless of when they arrive. The `PriorityQueue` was added in Java 5 to provide an automatic implementation for this behavior.

When you `offer()` an object onto a `PriorityQueue`, that object is sorted into the queue.[82] The default sorting uses the *natural order* of the objects in the queue, but you can modify the order by providing your own `Comparator`. The `PriorityQueue` ensures that when you call `peek()`, `poll()` or `remove()`, the element you get is the one with the highest priority.

It's trivial to make a `PriorityQueue` that works with built-in types like `Integer`, `String` or `Character`. In the following example, the first set of values are the identical random values from the previous example, showing that they emerge differently from the `PriorityQueue`:

[82]This actually depends on the implementation. Priority queue algorithms typically sort on insertion (maintaining a *heap*), but they may also perform the selection of the most important element upon removal. The choice of algorithm can be important if an object's priority can change while it is waiting in the queue.

```
// collections/PriorityQueueDemo.java
import java.util.*;

public class PriorityQueueDemo {
  public static void main(String[] args) {
    PriorityQueue<Integer> priorityQueue =
      new PriorityQueue<>();
    Random rand = new Random(47);
    for(int i = 0; i < 10; i++)
      priorityQueue.offer(rand.nextInt(i + 10));
    QueueDemo.printQ(priorityQueue);

    List<Integer> ints = Arrays.asList(25, 22, 20,
      18, 14, 9, 3, 1, 1, 2, 3, 9, 14, 18, 21, 23, 25);
    priorityQueue = new PriorityQueue<>(ints);
    QueueDemo.printQ(priorityQueue);
    priorityQueue = new PriorityQueue<>(
        ints.size(), Collections.reverseOrder());
    priorityQueue.addAll(ints);
    QueueDemo.printQ(priorityQueue);

    String fact = "EDUCATION SHOULD ESCHEW OBFUSCATION";
    List<String> strings =
      Arrays.asList(fact.split(""));
    PriorityQueue<String> stringPQ =
      new PriorityQueue<>(strings);
    QueueDemo.printQ(stringPQ);
    stringPQ = new PriorityQueue<>(
      strings.size(), Collections.reverseOrder());
    stringPQ.addAll(strings);
    QueueDemo.printQ(stringPQ);

    Set<Character> charSet = new HashSet<>();
    for(char c : fact.toCharArray())
      charSet.add(c); // Autoboxing
    PriorityQueue<Character> characterPQ =
      new PriorityQueue<>(charSet);
    QueueDemo.printQ(characterPQ);
  }
}
/* Output:
0 1 1 1 1 1 3 5 8 14
```

```
1 1 2 3 3 9 9 14 14 18 18 20 21 22 23 25 25
25 25 23 22 21 20 18 18 14 14 9 9 3 3 2 1 1
     A A B C C C D D E E E F H H I I L N N O O O O S S
S T T U U U W
W U U U T T S S S O O O O N N L I I H H F E E E D D C C
C B A A
  A B C D E F H I L N O S T U W
*/
```

Duplicates are allowed, and the lowest values have the highest priority (in the case of `String`, spaces also count as values and are higher in priority than letters). To change the ordering using a `Comparator` object, the third constructor call to `PriorityQueue<Integer>` and the second call to `PriorityQueue<String>` use the reverse-order `Comparator` produced by `Collections.reverseOrder()`.

The last section adds a `HashSet` to eliminate duplicate `Characters`.

`Integer`, `String` and `Character` work with `PriorityQueue` because these classes already have natural ordering built in. If you want you use your own class in a `PriorityQueue`, you must include additional functionality to produce natural ordering, or provide your own `Comparator`. There's a more sophisticated example that demonstrates this in the Appendix: Collection Topics.

`Collection` vs. `Iterator`

`Collection` is the root interface common to all sequence collections. It might be thought of as an "incidental interface," one that appeared because of commonality between other interfaces. In addition, the `java.util.AbstractCollection` class provides a default implementation for a `Collection`, so you can create a new subtype of `AbstractCollection` without unnecessary code duplication.

One argument for having an interface is it creates more generic code. By writing to an interface rather than an implementation, your code can be applied to more types of objects.[83] So if I write a method that takes a `Collection`, that method can be applied to any type that implements `Collection`—and this allows a new

[83]Some people advocate the automatic creation of an interface for every possible combination of methods in a class—sometimes for every single class. I believe that an interface should have more meaning than a mechanical duplication of method combinations, so I tend to wait until I see the value added by an interface before creating one.

class to implement `Collection` for use with my method. The Standard C++ Library has no common base class for its collections—all commonality between collections is achieved through iterators. In Java, it might seem sensible to follow the C++ approach, and to express commonality between collections using an iterator rather than a `Collection`. However, the two approaches are bound together because implementing `Collection` also means providing an `iterator()` method:

```
// collections/InterfaceVsIterator.java
import reflection.pets.*;
import java.util.*;

public class InterfaceVsIterator {
  public static void display(Iterator<Pet> it) {
    while(it.hasNext()) {
      Pet p = it.next();
      System.out.print(p.id() + ":" + p + " ");
    }
    System.out.println();
  }
  public static void display(Collection<Pet> pets) {
    for(Pet p : pets)
      System.out.print(p.id() + ":" + p + " ");
    System.out.println();
  }
  public static void main(String[] args) {
    List<Pet> petList = new PetCreator().list(8);
    Set<Pet> petSet = new HashSet<>(petList);
    Map<String, Pet> petMap = new LinkedHashMap<>();
    String[] names = ("Ralph, Eric, Robin, Lacey, " +
      "Britney, Sam, Spot, Fluffy").split(", ");
    for(int i = 0; i < names.length; i++)
      petMap.put(names[i], petList.get(i));
    display(petList);
    display(petSet);
    display(petList.iterator());
    display(petSet.iterator());
    System.out.println(petMap);
    System.out.println(petMap.keySet());
    display(petMap.values());
    display(petMap.values().iterator());
  }
```

```
}
/* Output:
0:Rat 1:Manx 2:Cymric 3:Mutt 4:Pug 5:Cymric 6:Pug
7:Manx
0:Rat 1:Manx 2:Cymric 3:Mutt 4:Pug 5:Cymric 6:Pug
7:Manx
0:Rat 1:Manx 2:Cymric 3:Mutt 4:Pug 5:Cymric 6:Pug
7:Manx
0:Rat 1:Manx 2:Cymric 3:Mutt 4:Pug 5:Cymric 6:Pug
7:Manx
{Ralph=Rat, Eric=Manx, Robin=Cymric, Lacey=Mutt,
Britney=Pug, Sam=Cymric, Spot=Pug, Fluffy=Manx}
[Ralph, Eric, Robin, Lacey, Britney, Sam, Spot, Fluffy]
0:Rat 1:Manx 2:Cymric 3:Mutt 4:Pug 5:Cymric 6:Pug
7:Manx
0:Rat 1:Manx 2:Cymric 3:Mutt 4:Pug 5:Cymric 6:Pug
7:Manx
*/
```

Both versions of `display()` work with `Map` objects as well as subtypes of `Collection`. Both `Collection` and `Iterator` decouple the `display()` methods from knowing the particular implementation of the underlying collection.

here the two approaches come up even. In fact, `Collection` pulls ahead a bit because it is `Iterable`, and so in the implementation of `display(Collection)` the *for-in* construct can be used, which makes the code a little cleaner.

An `Iterator` becomes compelling when you implement a foreign class, one that is not a `Collection`, when it would be difficult or annoying to make it implement the `Collection` interface. For example, if we create a `Collection` implementation by inheriting from a class that holds `Pet` objects, we must implement all the `Collection` methods, even if we don't use them within the `display()` method. Although this can easily be accomplished by inheriting from `AbstractCollection`, you're forced to implement `iterator()` anyway, along with `size()`, to provide the methods not implemented by `AbstractCollection`, but used by the other methods in `AbstractCollection`:

```
// collections/CollectionSequence.java
import reflection.pets.*;
import java.util.*;

public class CollectionSequence
extends AbstractCollection<Pet> {
  private Pet[] pets = new PetCreator().array(8);
  @Override
  public int size() { return pets.length; }
  @Override public Iterator<Pet> iterator() {
    return new Iterator<Pet>() {              // [1]
      private int index = 0;
      @Override public boolean hasNext() {
        return index < pets.length;
      }
      @Override
      public Pet next() { return pets[index++]; }
      @Override
      public void remove() { // Not implemented
        throw new UnsupportedOperationException();
      }
    };
  }
  public static void main(String[] args) {
    CollectionSequence c = new CollectionSequence();
    InterfaceVsIterator.display(c);
    InterfaceVsIterator.display(c.iterator());
  }
}
/* Output:
0:Rat 1:Manx 2:Cymric 3:Mutt 4:Pug 5:Cymric 6:Pug
7:Manx
0:Rat 1:Manx 2:Cymric 3:Mutt 4:Pug 5:Cymric 6:Pug
7:Manx
*/
```

The `remove()` method is an "optional operation," covered in the Appendix: Collection Topics. Here, it's not necessary to implement it, and if you call it, it will throw an exception.

- **[1]** Because `iterator()` returns `Iterator<Pet>`, you might think that the anonymous inner class definition could use the diamond syntax and Java could

infer the type. But that doesn't work; the type inference is still quite limited.

This example shows that if you implement `Collection`, you also implement `iterator()`, and just implementing `iterator()` alone requires only slightly less effort than inheriting from `AbstractCollection`. However, if your class already inherits from another class, you cannot also inherit from `AbstractCollection`. In that case, to implement `Collection` you'd have to implement all the methods in the interface. Here it's much easier to inherit and add the ability to create an iterator:

```
// collections/NonCollectionSequence.java
import reflection.pets.*;
import java.util.*;

class PetSequence {
  protected Pet[] pets = new PetCreator().array(8);
}

public class NonCollectionSequence extends PetSequence {
  public Iterator<Pet> iterator() {
    return new Iterator<Pet>() {
      private int index = 0;
      @Override public boolean hasNext() {
        return index < pets.length;
      }
      @Override
      public Pet next() { return pets[index++]; }
      @Override
      public void remove() { // Not implemented
        throw new UnsupportedOperationException();
      }
    };
  }
  public static void main(String[] args) {
    NonCollectionSequence nc =
      new NonCollectionSequence();
    InterfaceVsIterator.display(nc.iterator());
  }
}
/* Output:
0:Rat 1:Manx 2:Cymric 3:Mutt 4:Pug 5:Cymric 6:Pug
7:Manx
*/
```

Producing an `Iterator` is the least-coupled way of connecting a sequence to a method that consumes that sequence, and puts far fewer constraints on the sequence class than does implementing `Collection`.

for-in and Iterators

So far, the *for-in* syntax is primarily used with arrays, but it also works with any `Collection` object. You've actually seen a few examples of this using `ArrayList`, but here's a general proof:

```
// collections/ForInCollections.java
// All collections work with for-in
import java.util.*;

public class ForInCollections {
  public static void main(String[] args) {
    Collection<String> cs = new LinkedList<>();
    Collections.addAll(cs,
      "Take the long way home".split(" "));
    for(String s : cs)
      System.out.print("'" + s + "' ");
  }
}
/* Output:
'Take' 'the' 'long' 'way' 'home'
*/
```

`cs` is a `Collection` so this code shows that working with *for-in* is a characteristic of all `Collection` objects.

The reason this works is that Java 5 introduced an interface called `Iterable` containing an `iterator()` method that produces an `Iterator`. *for-in* uses this `Iterable` interface to move through a sequence. So if you create any class that implements `Iterable`, you can use it in a *for-in* statement:

```
// collections/IterableClass.java
// Anything Iterable works with for-in
import java.util.*;

public class IterableClass implements Iterable<String> {
  protected String[] words = ("And that is how " +
    "we know the Earth to be banana-shaped."
    ).split(" ");
  @Override public Iterator<String> iterator() {
    return new Iterator<String>() {
      private int index = 0;
      @Override public boolean hasNext() {
        return index < words.length;
      }
      @Override
      public String next() { return words[index++]; }
      @Override
      public void remove() { // Not implemented
        throw new UnsupportedOperationException();
      }
    };
  }
  public static void main(String[] args) {
    for(String s : new IterableClass())
      System.out.print(s + " ");
  }
}
/* Output:
And that is how we know the Earth to be banana-shaped.
*/
```

`iterator()` returns an instance of an anonymous inner implementation of `Iterator<String>` which delivers each word in the array. In `main()` you see that `IterableClass` does indeed work in a *for-in* statement.

In Java 5, a number of classes were made `Iterable`, primarily all `Collection` classes (but not `Maps`). For example, this code displays all operating system environment variables:

```
// collections/EnvironmentVariables.java
// {VisuallyInspectOutput}
import java.util.*;

public class EnvironmentVariables {
  public static void main(String[] args) {
    for(Map.Entry entry: System.getenv().entrySet()) {
      System.out.println(entry.getKey() + ": " +
        entry.getValue());
    }
  }
}
```

`System.getenv()`[84] returns a `Map`, `entrySet()` produces a `Set` of `Map.Entry` elements, and a `Set` is `Iterable` so it can be used in a *for-in* loop.

A *for-in* statement works with an array or anything `Iterable`, but that doesn't mean that an array is automatically an `Iterable`, nor is there any autoboxing that takes place:

```
// collections/ArrayIsNotIterable.java
import java.util.*;

public class ArrayIsNotIterable {
  static <T> void test(Iterable<T> ib) {
    for(T t : ib)
      System.out.print(t + " ");
  }
  public static void main(String[] args) {
    test(Arrays.asList(1, 2, 3));
    String[] strings = { "A", "B", "C" };
    // An array works in for-in, but it's not Iterable:
    //- test(strings);
    // You must explicitly convert it to an Iterable:
    test(Arrays.asList(strings));
  }
}
/* Output:
1 2 3 A B C
*/
```

[84] This was not available before Java 5, because it was thought too tightly coupled to the operating system, and thus to violate "write once, run anywhere." The fact it is included now suggests that the Java designers became more pragmatic.

Trying to pass an array as an `Iterable` argument fails. There is no automatic conversion to an `Iterable`; you must do it by hand.

The *Adapter Method* Idiom

What if you have an existing class that is `Iterable`, and you'd like to add one or more new ways to use this class in a *for-in* statement? For example, suppose you'd like to choose whether to iterate through a list of words in either a forward or reverse direction. If you inherit from the class and override the `iterator()` method, you replace the existing method and you don't get a choice.

One solution I call the *Adapter Method* idiom. The "Adapter" part comes from design patterns, because you must provide a particular interface to satisfy the *for-in* statement. When you have one interface and you need another one, writing an adapter solves the problem. Here, I want to *add* the ability to produce a reverse iterator to the default forward iterator, so I can't override. Instead, I add a method that produces an `Iterable` object which can then be used in the *for-in* statement. As you see here, this allows us to provide multiple ways to use *for-in*:

```
// collections/AdapterMethodIdiom.java
// The "Adapter Method" idiom uses for-in
// with additional kinds of Iterables
import java.util.*;

class ReversibleArrayList<T> extends ArrayList<T> {
  ReversibleArrayList(Collection<T> c) {
    super(c);
  }
  public Iterable<T> reversed() {
    return new Iterable<T>() {
      public Iterator<T> iterator() {
        return new Iterator<T>() {
          int current = size() - 1;
          @Override public boolean hasNext() {
            return current > -1;
          }
          @Override
          public T next() { return get(current--); }
          @Override
```

```
          public void remove() { // Not implemented
            throw new UnsupportedOperationException();
          }
        };
      }
    };
  }
}

public class AdapterMethodIdiom {
  public static void main(String[] args) {
    ReversibleArrayList<String> ral =
      new ReversibleArrayList<>(
        Arrays.asList("To be or not to be".split(" ")));
    // Grabs the ordinary iterator via iterator():
    for(String s : ral)
      System.out.print(s + " ");
    System.out.println();
    // Hand it the Iterable of your choice
    for(String s : ral.reversed())
      System.out.print(s + " ");
  }
}
/* Output:
To be or not to be
be to not or be To
*/
```

In `main()`, if you put the `ral` object in the *for-in* statement, you get the (default) forward iterator. But if you call `reversed()` on the object, it produces different behavior.

Using this approach, I can add two adapter methods to the `IterableClass.java` example:

```
// collections/MultiIterableClass.java
// Adding several Adapter Methods
import java.util.*;

public class MultiIterableClass extends IterableClass {
  public Iterable<String> reversed() {
    return new Iterable<String>() {
      public Iterator<String> iterator() {
        return new Iterator<String>() {
          int current = words.length - 1;
          @Override public boolean hasNext() {
            return current > -1;
          }
          @Override public String next() {
            return words[current--];
          }
          @Override
          public void remove() { // Not implemented
            throw new UnsupportedOperationException();
          }
        };
      }
    };
  }
  public Iterable<String> randomized() {
    return new Iterable<String>() {
      public Iterator<String> iterator() {
        List<String> shuffled =
          new ArrayList<>(Arrays.asList(words));
        Collections.shuffle(shuffled, new Random(47));
        return shuffled.iterator();
      }
    };
  }
  public static void main(String[] args) {
    MultiIterableClass mic = new MultiIterableClass();
    for(String s : mic.reversed())
      System.out.print(s + " ");
    System.out.println();
    for(String s : mic.randomized())
      System.out.print(s + " ");
    System.out.println();
```

```
    for(String s : mic)
      System.out.print(s + " ");
  }
}
/* Output:
banana-shaped. be to Earth the know we how is that And
is banana-shaped. Earth that how the be And we know to
And that is how we know the Earth to be banana-shaped.
*/
```

Notice that the second method, `random()`, doesn't create its own `Iterator` but returns the one from the shuffled `List`.

The output shows that the `Collections.shuffle()` method doesn't affect the original array, but only shuffles the references in `shuffled`. This is only true because the `randomized()` method wraps an `ArrayList` around the result of `Arrays.asList()`. If the `List` produced by `Arrays.asList()` is shuffled directly, it will modify the underlying array, as shown here:

```
// collections/ModifyingArraysAsList.java
import java.util.*;

public class ModifyingArraysAsList {
  public static void main(String[] args) {
    Random rand = new Random(47);
    Integer[] ia = { 1, 2, 3, 4, 5, 6, 7, 8, 9, 10 };
    List<Integer> list1 =
      new ArrayList<>(Arrays.asList(ia));
    System.out.println("Before shuffling: " + list1);
    Collections.shuffle(list1, rand);
    System.out.println("After shuffling: " + list1);
    System.out.println("array: " + Arrays.toString(ia));

    List<Integer> list2 = Arrays.asList(ia);
    System.out.println("Before shuffling: " + list2);
    Collections.shuffle(list2, rand);
    System.out.println("After shuffling: " + list2);
    System.out.println("array: " + Arrays.toString(ia));
  }
}
/* Output:
Before shuffling: [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
```

```
After shuffling: [4, 6, 3, 1, 8, 7, 2, 5, 10, 9]
array: [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
Before shuffling: [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
After shuffling: [9, 1, 6, 3, 7, 2, 5, 10, 4, 8]
array: [9, 1, 6, 3, 7, 2, 5, 10, 4, 8]
*/
```

In the first case, the output of `Arrays.asList()` is handed to the `ArrayList` constructor, and this creates an `ArrayList` that references the elements of `ia`. Shuffling these references doesn't modify the array. However, if you use the result of `Arrays.asList(ia)` directly, shuffling modifies the order of `ia`. It's important to be aware that `Arrays.asList()` produces a `List` object that uses the underlying array as its physical implementation. If you do anything to that `List` that modifies it, and you don't want the original array modified, make a copy into another collection.

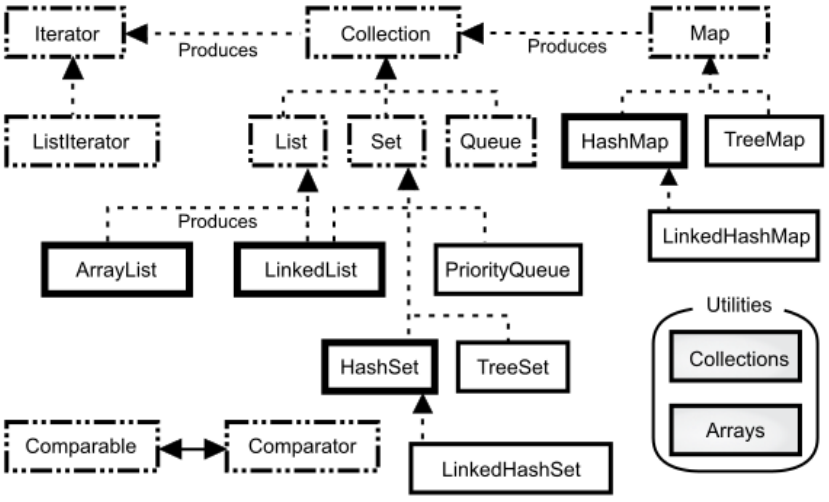
Summary

Java provides a number of ways to hold objects:

1. An array associates numerical indexes to objects. It holds objects of a known type so you don't have to cast the result when you're looking up an object. It can be multidimensional, and it can hold primitives. Although you can create arrays at run-time, the size of an array cannot be changed once you create it.
2. A `Collection` holds single elements, and a `Map` holds associated pairs. With Java generics, you specify the type of object held in the collections, so you can't put the wrong type into a collection and you don't have to cast elements when you fetch them out of a collection. Both `Collections` and `Maps` automatically resize themselves as you add more elements. A collection won't hold primitives, but autoboxing takes care of translating primitives back and forth to the wrapper types held in the collection.
3. Like an array, a `List` also associates numerical indexes to objects—thus, arrays and `Lists` are ordered collections.
4. Use an `ArrayList` if you perform many random accesses, but a `LinkedList` for many insertions and removals in the middle of the list.
5. The behavior of `Queues` and stacks is provided via the `LinkedList`.

6. A `Map` is a way to associate, not integral values, but *objects* with other objects. `HashMaps` are designed for rapid access, whereas a `TreeMap` keeps its keys in sorted order, and thus is not as fast as a `HashMap`. A `LinkedHashMap` keeps its elements in insertion order, but provides rapid access with hashing.
7. A `Set` only accepts one of each type of object. `HashSets` provide maximally fast lookups, whereas `TreeSets` keep the elements in sorted order. `LinkedHashSets` keep elements in insertion order.
8. Don't use the legacy classes `Vector`, `Hashtable`, and `Stack` in new code.

It's helpful to look at a simplified diagram of the Java collections (without the abstract classes or legacy components). This only includes the interfaces and classes you encounter on a regular basis.



Simple Collection Taxonomy

You'll see there are really only four basic collection components—`Map`, `List`, `Set`, and `Queue`—and only two or three implementations of each one (the `java.util.concurrent` implementations of `Queue` are not included in this diagram). The collections you use most often have heavy black lines around them.

The dotted boxes represent interfaces, and the solid boxes are regular (concrete) classes. The dotted lines with hollow arrows indicate that a particular class is implementing an interface. The solid arrows show that a class can produce objects of the class where the arrow is pointing. For example, any `Collection` can produce

an `Iterator`, and a `List` can produce a `ListIterator` (as well as an ordinary `Iterator` because `List` inherits `Collection`).

Here's an example that shows the different methods between the various classes. The actual code is from the Generics chapter; I'm just calling it here to produce the output. The output also shows the interfaces implemented in each class or interface:

```
// collections/CollectionDifferences.java
import onjava.*;

public class CollectionDifferences {
  public static void main(String[] args) {
    CollectionMethodDifferences.main(args);
  }
}
/* Output:
Collection: [add, addAll, clear, contains, containsAll,
equals, forEach, hashCode, isEmpty, iterator,
parallelStream, remove, removeAll, removeIf, retainAll,
size, spliterator, stream, toArray]
Interfaces in Collection: [Iterable]
Set extends Collection, adds: []
Interfaces in Set: [Collection]
HashSet extends Set, adds: []
Interfaces in HashSet: [Set, Cloneable, Serializable]
LinkedHashSet extends HashSet, adds: []
Interfaces in LinkedHashSet: [Set, Cloneable,
Serializable]
TreeSet extends Set, adds: [headSet,
descendingIterator, descendingSet, pollLast, subSet,
floor, tailSet, ceiling, last, lower, comparator,
pollFirst, first, higher]
Interfaces in TreeSet: [NavigableSet, Cloneable,
Serializable]
List extends Collection, adds: [replaceAll, get,
indexOf, subList, set, sort, lastIndexOf, listIterator]
Interfaces in List: [Collection]
ArrayList extends List, adds: [trimToSize,
ensureCapacity]
Interfaces in ArrayList: [List, RandomAccess,
Cloneable, Serializable]
LinkedList extends List, adds: [offerFirst, poll,
```

```
getLast, offer, getFirst, removeFirst, element,
removeLastOccurrence, peekFirst, peekLast, push,
pollFirst, removeFirstOccurrence, descendingIterator,
pollLast, removeLast, pop, addLast, peek, offerLast,
addFirst]
Interfaces in LinkedList: [List, Deque, Cloneable,
Serializable]
Queue extends Collection, adds: [poll, peek, offer,
element]
Interfaces in Queue: [Collection]
PriorityQueue extends Queue, adds: [comparator]
Interfaces in PriorityQueue: [Serializable]
Map: [clear, compute, computeIfAbsent,
computeIfPresent, containsKey, containsValue, entrySet,
equals, forEach, get, getOrDefault, hashCode, isEmpty,
keySet, merge, put, putAll, putIfAbsent, remove,
replace, replaceAll, size, values]
HashMap extends Map, adds: []
Interfaces in HashMap: [Map, Cloneable, Serializable]
LinkedHashMap extends HashMap, adds: []
Interfaces in LinkedHashMap: [Map]
SortedMap extends Map, adds: [lastKey, subMap,
comparator, firstKey, headMap, tailMap]
Interfaces in SortedMap: [Map]
TreeMap extends Map, adds: [descendingKeySet,
navigableKeySet, higherEntry, higherKey, floorKey,
subMap, ceilingKey, pollLastEntry, firstKey, lowerKey,
headMap, tailMap, lowerEntry, ceilingEntry,
descendingMap, pollFirstEntry, lastKey, firstEntry,
floorEntry, comparator, lastEntry]
Interfaces in TreeMap: [NavigableMap, Cloneable,
Serializable]
*/
```

All `Sets` except `TreeSet` have exactly the same interface as `Collection`. `List` and `Collection` differ significantly, although `List` requires methods from `Collection`. On the other hand, the methods in the `Queue` interface stand alone; the `Collection` methods are not required to create a functioning `Queue` implementation. Finally, the only intersection between `Map` and `Collection` is the fact that a `Map` can produce `Collections` using the `entrySet()` and `values()` methods.

Notice the tagging interface `java.util.RandomAccess`, which is attached to `Ar-`

`rayList` but not to `LinkedList`. This provides information for algorithms that dynamically change their behavior depending on a particular `List`.

It's true this organization is somewhat odd, as object-oriented hierarchies go. However, as you learn more about the collections in `java.util` (in particular, in the Appendix: Collection Topics), you'll see there are more issues than just a slightly odd inheritance structure. Collection libraries have always been difficult design problems—solving these problems involves satisfying a set of forces that often oppose each other. So be prepared for some compromises here and there.

Despite these issues, the Java collections are fundamental tools you can use on a day-to-day basis to make your programs simpler, more powerful, and more effective. It might take you a little while to get comfortable with some aspects of the library, but I think you'll find yourself rapidly acquiring and using the classes in this library.

Functional Programming

> A functional programming language manipulates pieces of code as easily as it manipulates data. Although Java is not a functional language, Java 8 *Lambda Expressions* and *Method References* allow you to program in a functional style.

In the early days of the computer age, memory was scarce and precious. Nearly everyone programmed in assembly language. People knew about compilers, but the mere thought of the inefficient code generation from such a thing—many bytes would certainly be generated that hand-coded assembly would never produce!

Often, just to fit a program into limited memory, programmers saved code space by modifying in-memory code to make it do something different, *while the program was executing*. This technique is called *self-modifying code*, and as long as a program was small enough for a handful of people to maintain all the tricky and arcane assembly code, you could probably get it to work.

Memory got cheaper and processors got faster. The C language appeared and was thought of as "high level" by most assembly-language programmers. Others discovered that C could make them significantly more productive. And with C, it still wasn't that hard to create self-modifying code.

With cheaper hardware, programs grew in size and complexity. It became difficult just to get programs to work. We sought ways to make code more consistent and understandable. Self-modifying code, in its purest form, turns out to be a really bad idea because it's very hard to be quite sure what it is doing. It's also difficult to test, because are you testing the output, some code in transition, the process of modification, etc.?

And yet, the idea of using code to somehow manipulate other code remains intriguing, as long as there is some way to make it safer.

From a code creation, maintenance, and reliability standpoint this idea is quite compelling. If, instead of writing lots of code from scratch, you start with existing

small pieces that are understandable, well-tested, and reliable. Then compose them together to create new code. Wouldn't that make us more productive and at the same time create more robust code?

This is what *functional programming* (FP) is about. By incorporating existing code to produce new functionality instead of writing everything from scratch, you get more reliable code, faster. This theory seems to hold up, at least in some situations. On the way, functional languages have produced nice syntax which some non-functional languages have appropriated.

You can also think of it this way:

> OO abstracts data, FP abstracts behavior.

A *pure functional* language goes further in its endeavor for safety. It imposes the additional constraint that all data must be *immutable*: set once and never changed. Values are handed to a function which then produces new values but never modifies anything external to itself (including its arguments or elements outside of that function's scope). When this is enforced, you know that any bugs are not caused by so-called *side effects*, because the function only creates and returns a result, nothing else.

Even better, the "immutable objects and no side effects" paradigm solves one of the most fundamental and vexing problems in parallel programming (when parts of your program are running simultaneously on multiple processors). This is the problem of *mutable shared state*, which means different parts of your code (running on different processors) can try to modify the same piece of memory at the same time (Who wins? Nobody knows). If functions never modify existing values but only produce new values—the definition of a pure functional language—there can be no contention over memory. Thus, pure functional languages are often put forward as the solution to parallel programming (there are also other viable solutions).

Be aware, then, that there are numerous motivations behind functional languages, which means describing them can be somewhat confusing. It often depends on perspective. The reasons span "it's for parallel programming," to "code reliability" and "code creation and library reuse."[85] Also remember that the arguments for FP—in particular, that programmers will create more robust code, faster—are still at least

[85]Pasting functionality together is a rather different approach, but it still enables a kind of library.

partly hypothetical. We have seen some good results,[86] but we haven't proven that a pure functional language is the best approach to solving the programming problem.

Ideas from FP are worth incorporating into non-FP languages. This happened in Python, for example, to the great benefit of that language. Java 8 adds its own features from FP, which we explore in this chapter.

Old vs. New

Ordinarily, methods produce different results depending on the data we pass. What if you want a method to behave differently from one call to the next? If we pass code to the method, we can control its behavior.

Previously, we've done this by creating an object containing the desired behavior inside a method, then passing that object to the method we want to control. The following example shows this, then adds the Java 8 approaches: method references and lambda expressions.

```
// functional/Strategize.java

interface Strategy {
  String approach(String msg);
}

class Soft implements Strategy {
  @Override
  public String approach(String msg) {
    return msg.toLowerCase() + "?";
  }
}

class Unrelated {
  static String twice(String msg) {
    return msg + " " + msg;
  }
}
```

[86] For example, a version of this eBook was produced using Pandoc, a program written in the pure functional language Haskell.

```
public class Strategize {
  Strategy strategy;
  String msg;
  Strategize(String msg) {
    strategy = new Soft();                // [1]
    this.msg = msg;
  }
  void communicate() {
    System.out.println(strategy.approach(msg));
  }
  void changeStrategy(Strategy strategy) {
    this.strategy = strategy;
  }
  public static void main(String[] args) {
    Strategy[] strategies = {
      new Strategy() {                    // [2]
        public String approach(String msg) {
          return msg.toUpperCase() + "!";
        }
      },
      msg -> msg.substring(0, 5),         // [3]
      Unrelated::twice                    // [4]
    };
    Strategize s = new Strategize("Hello there");
    s.communicate();
    for(Strategy newStrategy : strategies) {
      s.changeStrategy(newStrategy);      // [5]
      s.communicate();                    // [6]
    }
  }
}
/* Output:
hello there?
HELLO THERE!
Hello
Hello there Hello there
*/
```

`Strategy` provides the interface that carries functionality in its single `approach()` method. By creating different `Strategy` objects you create different behavior.

Traditionally we implement this behavior by making a class that `implements` the

`Strategy` interface, as in `Soft`.

- **[1]** In `Strategize`, you can see that `Soft` is the default strategy, as it is assigned in the constructor.
- **[2]** A slightly less verbose and more spontaneous approach is to create an anonymous inner class. There's still a fair amount of repetitious code, and you always have to look at it until you say "oh, I see, they're using an anonymous inner class."
- **[3]** This is the Java 8 *lambda expression*, distinguished by the arrow `->` separating the argument and function body. To the right of the arrow is the expression that is returned from the lambda. This achieves the same effect as the class definition and anonymous inner class, but with a lot less code.
- **[4]** This is the Java 8 *method reference*, distinguished by the `::`. To the left of the `::` is the name of a class or object, and to the right of the `::` is the name of a method, but without an argument list.
- **[5]** After using the default `Soft` strategy, we step through all the strategies in the array and place each one into `s` using `changeStrategy()`.
- **[6]** Now, each call to `communicate()` produces different behavior, depending on the strategy "code object" that's being used at the moment. We pass behavior, rather than only passing data.[87]

Prior to Java 8, we *have* been able to pass functionality via **[1]** and **[2]** However, the syntax has been so awkward to write and read that we've only done it when forced. Method references and lambda expressions make it possible to pass functionality when you want to, instead of only when you must.

Lambda Expressions

Lambda Expressions are function definitions written using the minimum possible syntax:

1. Lambda expressions produce functions, not classes. On the Java Virtual Machine (JVM), everything is a class, so there are various manipulations performed behind the scenes that make lambdas look like functions—but as a programmer, you can happily pretend they are "just functions."

[87] Sometimes functional languages describe this as "code as data."

2. The lambda syntax is as spare as possible, precisely to make lambdas easy to write and use.

You saw one lambda expression in `Strategize.java`, but there are other syntax variations:

```
// functional/LambdaExpressions.java

interface Description {
  String brief();
}

interface Body {
  String detailed(String head);
}

interface Multi {
  String twoArg(String head, Double d);
}

public class LambdaExpressions {

  static Body bod = h -> h + " No Parens!";      // [1]

  static Body bod2 = (h) -> h + " More details"; // [2]

  static Description desc = () -> "Short info";  // [3]

  static Multi mult = (h, n) -> h + n;           // [4]

  static Description moreLines = () -> {         // [5]
    System.out.println("moreLines()");
    return "from moreLines()";
  };

  public static void main(String[] args) {
    System.out.println(bod.detailed("Oh!"));
    System.out.println(bod2.detailed("Hi!"));
    System.out.println(desc.brief());
    System.out.println(mult.twoArg("Pi! ", 3.14159));
    System.out.println(moreLines.brief());
  }
```

```
}
/* Output:
Oh! No Parens!
Hi! More details
Short info
Pi! 3.14159
moreLines()
from moreLines()
*/
```

We start with three interfaces, each with a single method (you'll understand the significance of this shortly). However, each method has a different number of arguments, in order to demonstrate lambda expression syntax.

The basic syntax of any lambda expression is:

1. The arguments.
2. Followed by the `->`, which you might choose to read as "produces."
3. Everything after the `->` is the method body.

- **[1]** With a single argument, you can just say it without parentheses. This, however, is a special case.
- **[2]** The normal case is to use parentheses around the arguments. For consistency, you can also use parentheses around a single argument, although this is not commonly done.
- **[3]** With no arguments, you *must* use parentheses to indicate an empty argument list.
- **[4]** For more than one argument, place them in a parenthesized argument list.

So far, all the lambda expression method bodies have been a single line. The result of that expression automatically becomes the return value of the lambda expression, and it's illegal to use the `return` keyword here. This is another way that lambda expressions abbreviate the syntax for describing functionality.

- **[5]** If you do need multiple lines in your lambda expression, you must put those lines inside curly braces. In this situation, you revert to using `return` to produce a value from the lambda expression.

Lambda expressions typically produce more readable code than anonymous inner classes, so we'll use them when possible in this book.

Recursion

A *recursive* function is one that calls itself. It's possible to write recursive lambda expressions, with a caveat: The recursive method must be an instance variable or a `static` variable, otherwise you'll get a compile-time error. We'll create an example for each case.

Both examples need an interface that accepts an `int` and produces an `int`:

```
// functional/IntCall.java

interface IntCall {
  int call(int arg);
}
```

A *factorial*[88] for an integer `n` multiplies together all the positive integers less than or equal to `n`. A factorial function is a common recursive example:

```
// functional/RecursiveFactorial.java

public class RecursiveFactorial {
  static IntCall fact;
  public static void main(String[] args) {
    fact = n -> n == 0 ? 1 : n * fact.call(n - 1);
    for(int i = 0; i <= 10; i++)
      System.out.println(fact.call(i));
  }
}
/* Output:
1
1
2
6
24
120
720
5040
40320
362880
3628800
*/
```

[88] https://en.wikipedia.org/wiki/Factorial

Here, `fact` is a `static` variable. Note the use of the ternary if-else. The recursive function will keep calling itself until `i == 0`. All recursive functions have some kind of "stop condition," otherwise they'll recurse infinitely and produce an exception.

We can implement a Fibonacci sequence[89] as a recursive lambda expression, this time using an instance variable:

```
// functional/RecursiveFibonacci.java

public class RecursiveFibonacci {
  IntCall fib;
  RecursiveFibonacci() {
    fib = n -> n == 0 ? 0 :
               n == 1 ? 1 :
               fib.call(n - 1) + fib.call(n - 2);
  }
  int fibonacci(int n) { return fib.call(n); }
  public static void main(String[] args) {
    RecursiveFibonacci rf = new RecursiveFibonacci();
    for(int i = 0; i <= 10; i++)
      System.out.println(rf.fibonacci(i));
  }
}
/* Output:
0
1
1
2
3
5
8
13
21
34
55
*/
```

The Fibonacci sequence sums the last two elements in the sequence to produce the next one.

[89] https://en.wikipedia.org/wiki/Fibonacci_number

Method References

Java 8 *Method References* refer to methods without the extra baggage required by previous versions of Java. A method reference is a class name or an object name, followed by a : :[90], then the name of the method:

```
// functional/MethodReferences.java
import java.util.*;

interface Callable {                          // [1]
  void call(String s);
}

class Describe {
  void show(String msg) {                     // [2]
    System.out.println(msg);
  }
}

public class MethodReferences {
  static void hello(String name) {            // [3]
    System.out.println("Hello, " + name);
  }
  static class Description {
    String about;
    Description(String desc) { about = desc; }
    void help(String msg) {                   // [4]
      System.out.println(about + " " + msg);
    }
  }
  static class Helper {
    static void assist(String msg) {          // [5]
      System.out.println(msg);
    }
  }
  public static void main(String[] args) {
    Describe d = new Describe();
    Callable c = d::show;                     // [6]
    c.call("call()");                         // [7]
```

[90]This syntax came from C++.

```
    c = MethodReferences::hello;           // [8]
    c.call("Bob");

    c = new Description("valuable")::help; // [9]
    c.call("information");

    c = Helper::assist;                    // [10]
    c.call("Help!");
  }
}
/* Output:
call()
Hello, Bob
valuable information
Help!
*/
```

- **[1]** We start out with a single-method interface (again, you'll soon learn the importance of this).
- **[2]** The *signature* (argument types and return type) of `show()` conforms to the signature of `Callable`'s `call()`.
- **[3]** `hello()` is also signature-conformant to `call()`.
- **[4]** ... as is `help()`, a non-`static` method within a `static` inner class.
- **[5]** `assist()` is a `static` method inside a `static` inner class.
- **[6]** We assign a method reference for the `Describe` object to a `Callable`—which doesn't have a `show()` method but rather a `call()` method. However, Java seems fine with this seemingly-odd assignment, because the method reference is signature-conformant to `Callable`'s `call()` method.
- **[7]** We can now invoke `show()` by calling `call()`, because Java maps `call()` onto `show()`.
- **[8]** This is a `static` method reference.
- **[9]** This is another version of **[6]**: a method reference for a method attached to a live object, which is sometimes called a *bound method reference*.
- **[10]** Finally, getting a method reference for a `static` method of a `static` inner class looks just like the outer-class version at **[8]**.

This isn't an exhaustive example; we'll look at all the variations of method references shortly.

Runnable

The `Runnable` interface has been in Java since version 1.0, so no `import` is necessary. It also conforms to the special single-method interface format: Its method `run()` takes no arguments and has no return value. We can therefore use a lambda expression and a method reference as a `Runnable`:

```
// functional/RunnableMethodReference.java
// Method references with interface Runnable

class Go {
  static void go() {
    System.out.println("Go::go()");
  }
}

public class RunnableMethodReference {
  public static void main(String[] args) {

    new Thread(new Runnable() {
      public void run() {
        System.out.println("Anonymous");
      }
    }).start();

    new Thread(
      () -> System.out.println("lambda")
    ).start();

    new Thread(Go::go).start();
  }
}
/* Output:
Anonymous
lambda
Go::go()
*/
```

A `Thread` object takes a `Runnable` as its constructor argument, and has a method `start()` which calls `run()`. Notice that only the anonymous inner class is required to have a method named `run()`.

Unbound Method References

An *unbound method reference* refers to an ordinary (non-static) method, without an associated object. To apply an unbound reference, you must supply the object:

```
// functional/UnboundMethodReference.java
// Method reference without an object

class X {
  String f() { return "X::f()"; }
}

interface MakeString {
  String make();
}

interface TransformX {
  String transform(X x);
}

public class UnboundMethodReference {
  public static void main(String[] args) {
    // MakeString ms = X::f;               // [1]
    TransformX sp = X::f;
    X x = new X();
    System.out.println(sp.transform(x));    // [2]
    System.out.println(x.f()); // Same effect
  }
}
/* Output:
X::f()
X::f()
*/
```

So far, we've seen references to methods that have the same signature as their associated interface. At **[1]**, we try to do the same thing for `f()` in `X`, attempting to assign to a `MakeString`. This produces an error from the compiler about an "invalid method reference," even though `make()` has the same signature as `f()`. The problem is that there's actually another (hidden) argument involved: our old friend `this`. You can't call `f()` without an `X` object to call it upon. Thus, `X::f` represents an unbound method reference, because it hasn't been "bound" to an object.

To solve the problem we need an `X` object, so our interface actually needs an extra argument, as you see in `TransformX`. If you assign `X::f` to a `TransformX`, Java is happy. We must now make a second mental adjustment—with an unbound reference, the signature of the *functional method* (the single method in the interface) no longer quite matches the signature of the method reference. There's a good reason, which is that you need an object to call the method on.

The result at **[2]** is a bit of a brain-teaser. I take the unbound reference and call `transform()` on it, passing it an `X`, and somehow that results in the call to `x.f()`. Java knows it must take the first argument, which is effectively `this`, and call the method on it.

If your method has more arguments, just follow the pattern of the first argument taking `this`:

```
// functional/MultiUnbound.java
// Unbound methods with multiple arguments

class This {
  void two(int i, double d) {}
  void three(int i, double d, String s) {}
  void four(int i, double d, String s, char c) {}
}

interface TwoArgs {
  void call2(This athis, int i, double d);
}

interface ThreeArgs {
  void call3(This athis, int i, double d, String s);
}

interface FourArgs {
  void call4(
    This athis, int i, double d, String s, char c);
}

public class MultiUnbound {
  public static void main(String[] args) {
    TwoArgs twoargs = This::two;
    ThreeArgs threeargs = This::three;
```

```
    FourArgs fourargs = This::four;
    This athis = new This();
    twoargs.call2(athis, 11, 3.14);
    threeargs.call3(athis, 11, 3.14, "Three");
    fourargs.call4(athis, 11, 3.14, "Four", 'Z');
  }
}
```

To make a point, I named the class `This` and the first arguments for the functional methods `athis`, but you should choose other names to prevent confusion in production code.

Constructor Method References

You can also capture a reference for a constructor, and later call that constructor via the reference.

```
// functional/CtorReference.java

class Dog {
  String name;
  int age = -1; // For "unknown"
  Dog() { name = "stray"; }
  Dog(String nm) { name = nm; }
  Dog(String nm, int yrs) { name = nm; age = yrs; }
}

interface MakeNoArgs {
  Dog make();
}

interface Make1Arg {
  Dog make(String nm);
}

interface Make2Args {
  Dog make(String nm, int age);
}

public class CtorReference {
```

```
  public static void main(String[] args) {
    MakeNoArgs mna = Dog::new;        // [1]
    Make1Arg m1a = Dog::new;          // [2]
    Make2Args m2a = Dog::new;         // [3]

    Dog dn = mna.make();
    Dog d1 = m1a.make("Comet");
    Dog d2 = m2a.make("Ralph", 4);
  }
}
```

`Dog` has three constructors, and the `make()` methods inside the functional interfaces reflect the constructor argument lists (the `make()` methods can have different names).

Notice how we use `Dog::new` for each of **[1]**, **[2]**, and **[3]**. There's only one name for all three constructors: `::new`. But the constructor reference is assigned to a different interface in each case, and the compiler can detect from that which constructor reference to use.

The compiler can see that calling your functional method (`make()`, in this example) means to call the constructor.

Functional Interfaces

Both method references and lambda expressions must be assigned, and those assignations require type information for the compiler to ensure type correctness. Lambda expressions in particular introduce new requirements. Consider:

```
x -> x.toString()
```

We see that the return type must be a `String`, but what type is `x`? Because lambda expressions include a form of *type inference* (the compiler figures some things out about types instead of requiring the programmer to be explicit), the compiler must somehow be able to deduce the type of `x`.

Here's a second example:

```
(x, y) -> x + y
```

Now `x` and `y` can be any types that support the `+` operator, including two different numeric types or a `String` and some other type that will automatically convert to a `String` (this includes most types). But when this lambda expression is assigned, the compiler must determine the exact types of `x` and `y` to generate the correct code.

The same issue holds true for method references. Suppose you want to pass

```
System.out::println
```

to a method you are writing—what type do you give for the method's argument?

To solve this problem, Java 8 introduces `java.util.function` containing a set of interfaces which are target types for lambda expressions and method references. Each interface contains only one abstract method, called the *functional method*.

This "functional method" pattern can be enforced using the `@FunctionalInterface` annotation when you write interfaces:

```
// functional/FunctionalAnnotation.java

@FunctionalInterface
interface Functional {
  String goodbye(String arg);
}

interface FunctionalNoAnn {
  String goodbye(String arg);
}

/*
@FunctionalInterface
interface NotFunctional {
  String goodbye(String arg);
  String hello(String arg);
}
Produces error message:
NotFunctional is not a functional interface
multiple non-overriding abstract methods
found in interface NotFunctional
*/
```

```
public class FunctionalAnnotation {
  public String goodbye(String arg) {
    return "Goodbye, " + arg;
  }
  public static void main(String[] args) {
    FunctionalAnnotation fa =
      new FunctionalAnnotation();
    Functional f = fa::goodbye;
    FunctionalNoAnn fna = fa::goodbye;
    // Functional fac = fa; // Incompatible
    Functional fl = a -> "Goodbye, " + a;
    FunctionalNoAnn fnal = a -> "Goodbye, " + a;
  }
}
```

The `@FunctionalInterface` annotation is optional; Java treats both `Functional` and `FunctionalNoAnn` as functional interfaces in `main()`. The value of `@FunctionalInterface` is seen in the definition of `NotFunctional`: More than one method in your interface produces a compile-time error message.

Look closely at what happens in the definitions of `f` and `fna`. `Functional` and `FunctionalNoAnn` define interfaces. Yet what is assigned is just the method `goodbye`. First, this is only a method and not a class. Second, it's not even a method of a class that implements one of those interfaces. This is a bit of magic that was added to Java 8: if you assign a method reference or a lambda expression to a functional interface (and the types fit), Java will adapt your assignment to the target interface. Under the covers, the compiler wraps your method reference or lambda expression in an instance of a class that implements the target interface.

A `@FunctionalInterface` is also called a *Single Abstract Method* (SAM) type.

Even though `FunctionalAnnotation` does fit the `Functional` mold, if we try to assign a `FunctionalAnnotation` directly to a `Functional` as in the definition for `fac`, Java won't let us—which we expect because it doesn't explicitly implement `Functional`. The only surprise is that Java 8 allows us to assign functions to interfaces, which results in a much nicer, simpler syntax.

The goal of `java.util.function` is to create a complete-enough set of target interfaces that you don't ordinarily need to define your own. Mostly because of primitive types, this produces a small explosion of interfaces. If you understand the

naming pattern, you can generally detect what a particular interface does by looking at the name. Here are the basic naming guidelines:

1. If it only deals with objects, not primitives, then it's just a straightforward name, like `Function`, `Consumer`, `Predicate`, etc. The argument types are added via generics.
2. If it takes a primitive argument, this is denoted by the first part of the name, as in `LongConsumer`, `DoubleFunction`, `IntPredicate`, etc. An exception to this is the primitive `Supplier` types.
3. If it returns a primitive result, this is indicated with `To`, as in `ToLongFunction<T>` and `IntToLongFunction`.
4. If it returns the same type as its argument(s), it's an `Operator`, with `UnaryOperator` for one argument and `BinaryOperator` for two arguments.
5. If it takes two arguments and returns a `boolean`, it's a `Predicate`.
6. If it takes two arguments of different types, there's a `Bi` in the name.

This table describes the target types in `java.util.function` (with noted exceptions):

Characteristic	Name Functional Method	Usage
No arguments; Returns nothing	`Runnable` (java.lang) `run()`	`Runnable`

No arguments; Returns any type	`Supplier` `get()`
	`getAstype()`
		`Supplier<T>`
		`BooleanSupplier`
		`IntSupplier`
		`LongSupplier`
		`DoubleSupplier`

No arguments; Returns any type	`Callable`
	(java.util.concurrent)
	`call()`
		`Callable<V>`

One argument; Returns nothing	`Consumer` `accept()`	`Consumer<T>`
		`IntConsumer`
		`LongConsumer`
		`DoubleConsumer`

Two-argument `Consumer`	`BiConsumer` `accept()`	`BiConsumer<T,U>`

Characteristic	Name Functional Method	Usage
Two-argument `Consumer`; First arg is a reference; Second arg is a primitive	`ObjtypeConsumer` `accept()`	`ObjIntConsumer<T>` `ObjLongConsumer<T>` `ObjDoubleConsumer<T>`
One argument; Returns a different type	`Function` `apply()` `Totype` & `typeTotype`: `applyAstype()`	`Function<T,R>` `IntFunction<R>` `LongFunction<R>` `DoubleFunction<R>` `ToIntFunction<T>` `ToLongFunction<T>` `ToDoubleFunction<T>` `IntToLongFunction` `IntToDoubleFunction` `LongToIntFunction` `LongToDoubleFunction` `DoubleToIntFunction` `DoubleToLongFunction`
One argument; Returns the same type	`UnaryOperator` `apply()`	`UnaryOperator<T>` `IntUnaryOperator` `LongUnaryOperator` `DoubleUnaryOperator`
Two arguments, same type; Returns the same type	`BinaryOperator` `apply()`	`BinaryOperator<T>` `IntBinaryOperator` `LongBinaryOperator` `DoubleBinaryOperator`
Two arguments, same type; Returns `int`	`Comparator` (java.util) `compare()`	`Comparator<T>`
Two arguments; Returns `boolean`	`Predicate` `test()`	`Predicate<T>` `BiPredicate<T,U>` `IntPredicate` `LongPredicate` `DoublePredicate`

Characteristic	Name Functional Method	Usage
Primitive argument; Returns a primitive	typeTotypeFunction applyAstype()	IntToLongFunction IntToDoubleFunction LongToIntFunction LongToDoubleFunction DoubleToIntFunction DoubleToLongFunction
Two arguments; Different types	Bi operation (method name varies)	BiFunction<T,U,R> BiConsumer<T,U> BiPredicate<T,U> ToIntBiFunction<T,U> ToLongBiFunction<T,U> ToDoubleBiFunction<T,U>

You might imagine additional rows for further illumination, but this table gives the general idea, and should allow you to, more or less, deduce the functional interface you need.

You can see that some choices were made in creating `java.util.function`. For example, why no `IntComparator`, `LongComparator` and `DoubleComparator`? There's a `BooleanSupplier` but no other interfaces where `Boolean` is represented. There's a generic `BiConsumer`, but no `BiConsumers` for all the `int`, `long` and `double` variations (I can sympathize with why they gave up on that one). Are these oversights or did someone decide that the usage of the other permutations were too small (and how did they come to that conclusion)?

You can also see how much complexity the primitive types add to Java. They were included in the first version of the language because of efficiency concerns—which were mitigated fairly soon. Now, for the lifetime of the language, we are stuck with the effects of a poor language design choice.

Here's an example that enumerates all the different `Function` variants, applied to lambda expressions:

```
// functional/FunctionVariants.java
import java.util.function.*;

class Foo {}

class Bar {
  Foo f;
  Bar(Foo f) { this.f = f; }
}

class IBaz {
  int i;
  IBaz(int i) {
    this.i = i;
  }
}

class LBaz {
  long l;
  LBaz(long l) {
    this.l = l;
  }
}

class DBaz {
  double d;
  DBaz(double d) {
    this.d = d;
  }
}

public class FunctionVariants {
  static Function<Foo,Bar> f1 = f -> new Bar(f);
  static IntFunction<IBaz> f2 = i -> new IBaz(i);
  static LongFunction<LBaz> f3 = l -> new LBaz(l);
  static DoubleFunction<DBaz> f4 = d -> new DBaz(d);
  static ToIntFunction<IBaz> f5 = ib -> ib.i;
  static ToLongFunction<LBaz> f6 = lb -> lb.l;
  static ToDoubleFunction<DBaz> f7 = db -> db.d;
  static IntToLongFunction f8 = i -> i;
  static IntToDoubleFunction f9 = i -> i;
  static LongToIntFunction f10 = l -> (int)l;
```

```
  static LongToDoubleFunction f11 = l -> l;
  static DoubleToIntFunction f12 = d -> (int)d;
  static DoubleToLongFunction f13 = d -> (long)d;

  public static void main(String[] args) {
    Bar b = f1.apply(new Foo());
    IBaz ib = f2.apply(11);
    LBaz lb = f3.apply(11);
    DBaz db = f4.apply(11);
    int i = f5.applyAsInt(ib);
    long l = f6.applyAsLong(lb);
    double d = f7.applyAsDouble(db);
    l = f8.applyAsLong(12);
    d = f9.applyAsDouble(12);
    i = f10.applyAsInt(12);
    d = f11.applyAsDouble(12);
    i = f12.applyAsInt(13.0);
    l = f13.applyAsLong(13.0);
  }
}
```

These lambda expressions attempt to produce the simplest code that will fit the signature. In some cases casts were necessary, otherwise the compiler complains about truncation errors.

Each test in `main()` shows the different kinds of `apply` methods in the `Function` interfaces. Each one produces a call to its associated lambda expression.

Method references have their own magic:

```
// functional/MethodConversion.java
import java.util.function.*;

class In1 {}
class In2 {}

public class MethodConversion {
  static void accept(In1 i1, In2 i2) {
    System.out.println("accept()");
  }
  static void someOtherName(In1 i1, In2 i2) {
    System.out.println("someOtherName()");
```

```
  }
  public static void main(String[] args) {
    BiConsumer<In1,In2> bic;

    bic = MethodConversion::accept;
    bic.accept(new In1(), new In2());

    bic = MethodConversion::someOtherName;
    // bic.someOtherName(new In1(), new In2()); // Nope
    bic.accept(new In1(), new In2());
  }
}
/* Output:
accept()
someOtherName()
*/
```

Look up the documentation for `BiConsumer`. You'll see its functional method is `accept()`. Indeed, if we name our method `accept()`, it works as a method reference. But we can give it a completely different name like `someOtherName()` and it works as well, as long as the argument types and return type are the same as `BiConsumer`'s `accept()`.

Thus, when working with functional interfaces, the name doesn't matter—only the argument types and return type. Java performs a mapping of your name onto the functional method of the interface. To invoke your method, you call the functional method name (`accept()`, in this case), not your method name.

Now we'll look at all the class-based functionals (that is, those that don't involve primitives), applied to method references. Again, I've created the simplest methods that fit the functional signatures:

```
// functional/ClassFunctionals.java
import java.util.*;
import java.util.function.*;

class AA {}
class BB {}
class CC {}

public class ClassFunctionals {
  static AA f1() { return new AA(); }
  static int f2(AA aa1, AA aa2) { return 1; }
  static void f3(AA aa) {}
  static void f4(AA aa, BB bb) {}
  static CC f5(AA aa) { return new CC(); }
  static CC f6(AA aa, BB bb) { return new CC(); }
  static boolean f7(AA aa) { return true; }
  static boolean f8(AA aa, BB bb) { return true; }
  static AA f9(AA aa) { return new AA(); }
  static AA f10(AA aa1, AA aa2) { return new AA(); }
  public static void main(String[] args) {
    Supplier<AA> s = ClassFunctionals::f1;
    s.get();
    Comparator<AA> c = ClassFunctionals::f2;
    c.compare(new AA(), new AA());
    Consumer<AA> cons = ClassFunctionals::f3;
    cons.accept(new AA());
    BiConsumer<AA,BB> bicons = ClassFunctionals::f4;
    bicons.accept(new AA(), new BB());
    Function<AA,CC> f = ClassFunctionals::f5;
    CC cc = f.apply(new AA());
    BiFunction<AA,BB,CC> bif = ClassFunctionals::f6;
    cc = bif.apply(new AA(), new BB());
    Predicate<AA> p = ClassFunctionals::f7;
    boolean result = p.test(new AA());
    BiPredicate<AA,BB> bip = ClassFunctionals::f8;
    result = bip.test(new AA(), new BB());
    UnaryOperator<AA> uo = ClassFunctionals::f9;
    AA aa = uo.apply(new AA());
    BinaryOperator<AA> bo = ClassFunctionals::f10;
    aa = bo.apply(new AA(), new AA());
  }
}
```

Note that each method name is arbitrary (`f1()`, `f2()`, etc.), but as you just saw, once the method reference is assigned to a functional interface, you can call the functional method associated with that interface. In this example, these are `get()`, `compare()`, `accept()`, `apply()`, and `test()`.

Functional Interfaces with More Arguments

The interfaces in `java.util.function` are limited. There's a `BiFunction`, but it stops there. What if you need an interface for three-argument functions? Those interfaces are fairly straightforward, so it's easy to look at the Java library source code and make our own:

```
// functional/TriFunction.java

@FunctionalInterface
public interface TriFunction<T, U, V, R> {
    R apply(T t, U u, V v);
}
```

A short test will verify it works:

```
// functional/TriFunctionTest.java

public class TriFunctionTest {
  static int f(int i, long l, double d) { return 99; }
  public static void main(String[] args) {
    TriFunction<Integer, Long, Double, Integer> tf =
      TriFunctionTest::f;
    tf = (i, l, d) -> 12;
  }
}
```

We test both a method reference and a lambda expression.

Missing Primitive Functionals

Let's revisit `BiConsumer`, to see how we would create the various missing permutations involving `int`, `long` and `double`:

```
// functional/BiConsumerPermutations.java
import java.util.function.*;

public class BiConsumerPermutations {
  static BiConsumer<Integer, Double> bicid = (i, d) ->
    System.out.format("%d, %f%n", i, d);
  static BiConsumer<Double, Integer> bicdi = (d, i) ->
    System.out.format("%d, %f%n", i, d);
  static BiConsumer<Integer, Long> bicil = (i, l) ->
    System.out.format("%d, %d%n", i, l);
  public static void main(String[] args) {
    bicid.accept(47, 11.34);
    bicdi.accept(22.45, 92);
    bicil.accept(1, 11L);
  }
}
/* Output:
47, 11.340000
92, 22.450000
1, 11
*/
```

For display, I use `System.out.format()` which is like `System.out.println()` except it provides far more display options. Here, the `%f` says I'm giving it `n` as a floating-point value, and the `%d` says that `n` is an integral value. I'm able to include spaces, and it doesn't add a newline unless you put in a `%n`—it will also accept the traditional `\n` for newlines, but `%n` is automatically cross-platform, which is another reason to use `format()`.

The example simply uses the appropriate wrapper types, and boxing and unboxing takes care of converting back and forth between primitives. We can also use wrapped types with, for example, `Function`, instead of its predefined primitive variations:

```
// functional/FunctionWithWrapped.java
import java.util.function.*;

public class FunctionWithWrapped {
  public static void main(String[] args) {
    Function<Integer, Double> fid = i -> (double)i;
    IntToDoubleFunction fid2 = i -> i;
  }
}
```

Without the cast, you get an error message: "Integer cannot be converted to Double," whereas the `IntToDoubleFunction` version has no such problem. The Java library code for `IntToDoubleFunction` looks like this:

```
@FunctionalInterface
public interface IntToDoubleFunction {
  double applyAsDouble(int value);
}
```

Because we can simply write `Function<Integer,Double>` and produce working results, it's clear that the only reason for the primitive variants of the functionals is to prevent the autoboxing and autounboxing involved with passing arguments and returning results. That is, for performance.

It seems safe to conjecture that the reason some of the functional types have definitions and others don't is because of projected frequency of use.

Of course, if performance actually becomes a problem because of a missing primitive functional, you can easily write your own interface(using the Java library sources for reference)—although it seems unlikely this is your performance bottleneck.

Higher-Order Functions

This name can sound intimidating, but:

> A higher-order function[91] is simply a function that consumes or produces functions.

Let's first look at producing a function:

[91] https://en.wikipedia.org/wiki/Higher-order_function

```
// functional/ProduceFunction.java
import java.util.function.*;

interface
FuncSS extends Function<String, String> {}   // [1]

public class ProduceFunction {
  static FuncSS produce() {
    return s -> s.toLowerCase();             // [2]
  }
  public static void main(String[] args) {
    FuncSS f = produce();
    System.out.println(f.apply("YELLING"));
  }
}
/* Output:
yelling
*/
```

Here, `produce()` is the higher-order function.

- **[1]** Using inheritance, you can easily create an alias for a specialized interface.
- **[2]** With lambda expressions, it becomes almost effortless to create and return a function from within a method.

To consume a function, the consuming method must describe the function type correctly in its argument list:

```
// functional/ConsumeFunction.java
import java.util.function.*;

class One {}
class Two {}

public class ConsumeFunction {
  static Two consume(Function<One,Two> onetwo) {
    return onetwo.apply(new One());
  }
  public static void main(String[] args) {
    Two two = consume(one -> new Two());
  }
}
```

Things get particularly interesting when you produce a new function based on the consumed function:

```
// functional/TransformFunction.java
import java.util.function.*;

class I {
  @Override public String toString() { return "I"; }
}

class O {
  @Override public String toString() { return "O"; }
}

public class TransformFunction {
  static Function<I,O> transform(Function<I,O> in) {
    return in.andThen(o -> {
      System.out.println(o);
      return o;
    });
  }
  public static void main(String[] args) {
    Function<I,O> f2 = transform(i -> {
      System.out.println(i);
      return new O();
    });
    O o = f2.apply(new I());
  }
}
/* Output:
I
O
*/
```

here, `transform()` produces a function with the same signature as the one passed in, but you can produce any kind you'd like.

This uses a `default` method in the `Function` interface called `andThen()` which is specifically designed for manipulating functions. As the name implies, `andThen()` is invoked *after* the `in` function is called (there's also `compose()`, which applies the new function before the `in` function). To attach an `andThen()` function, we simply pass

that function as an argument. What comes out of `transform()` is a new function, which combines the action of `in` with that of the `andThen()` argument.

Closures

In `ProduceFunction.java` in the previous section, we returned a lambda function from a method. That example kept things simple, but there are some issues we must explore around returning lambdas. These issues are summarized by the term *closure*. Closures are important because they make it easy to produce functions.

Consider a more complex lambda, one that uses variables outside its function scope. What happens when you return that function? That is, what does it reference for those "outside" variables when you call the function? It becomes quite challenging if the language doesn't automatically solve this problem. Languages that do solve it are said to support closures, or to be *lexically scoped* (the term *variable capture* is also used). Java 8 provides limited, but reasonable, closure support, which we'll examine using some simple examples.

First, here's a method to return a function that accesses an object field and a method argument:

```
// functional/Closure1.java
import java.util.function.*;

public class Closure1 {
  int i;
  IntSupplier makeFun(int x) {
    return () -> x + i++;
  }
}
```

Under further consideration, this use of `i` isn't that big of a challenge, because the object is likely to exist after you call `makeFun()`—indeed, the garbage collector almost certainly preserves an object with extant functions tied back to it this way.[92] Of course, if you call `makeFun()` multiple times for the same object, you'll end up with several functions all sharing the same storage for `i`:

[92] I have not tested this statement.

```
// functional/SharedStorage.java
import java.util.function.*;

public class SharedStorage {
  public static void main(String[] args) {
    Closure1 c1 = new Closure1();
    IntSupplier f1 = c1.makeFun(0);
    IntSupplier f2 = c1.makeFun(0);
    IntSupplier f3 = c1.makeFun(0);
    System.out.println(f1.getAsInt());
    System.out.println(f2.getAsInt());
    System.out.println(f3.getAsInt());
  }
}
/* Output:
0
1
2
*/
```

Every call to `getAsInt()` increments `i`, demonstrating that the storage is shared.

What if `i` is local to `makeFun()`? Under normal circumstances, `i` is gone when `makeFun()` completes. It still compiles, however:

```
// functional/Closure2.java
import java.util.function.*;

public class Closure2 {
  IntSupplier makeFun(int x) {
    int i = 0;
    return () -> x + i;
  }
}
```

The `IntSupplier` returned by `makeFun()` "closes over" `i` and `x`, thus both are still valid when you invoke the returned function. Notice, however, I didn't increment `i` as in `Closure1.java`. Trying to increment it produces a compile-time error:

```
// functional/Closure3.java
// {WillNotCompile}
import java.util.function.*;

public class Closure3 {
  IntSupplier makeFun(int x) {
    int i = 0;
    // Neither x++ nor i++ will work:
    return () -> x++ + i++;
  }
}
```

The error, repeated for both `x` and `i`, is:

```
local variables referenced from a lambda
expression must be final or effectively final
```

Clearly, if we make `x` and `i` `final` it will work, because then we can't increment either one:

```
// functional/Closure4.java
import java.util.function.*;

public class Closure4 {
  IntSupplier makeFun(final int x) {
    final int i = 0;
    return () -> x + i;
  }
}
```

But why did `Closure2.java` work when `x` and `i` were not `final`? This is where the meaning of "effectively" `final` appears. The term was created for Java 8 to say you haven't explicitly declared a variable to be `final` but you're still treating it that way—you aren't changing it. A local variable is effectively final if its initial value is never changed.

If `x` and `i` are changed elsewhere in the method (but not inside of the returned function), the compiler still considers it an error. Each increment produces a separate error message:

```
// functional/Closure5.java
// {WillNotCompile}
import java.util.function.*;

public class Closure5 {
  IntSupplier makeFun(int x) {
    int i = 0;
    i++;
    x++;
    return () -> x + i;
  }
}
```

To be "effectively final" means you *could* apply the `final` keyword to the variable declaration without changing any of the rest of the code. It's actually `final`, you just haven't bothered to say so.

We can actually fix the issue in `Closure5.java` by assigning `x` and `i` to `final` variables before using them in the closure:

```
// functional/Closure6.java
import java.util.function.*;

public class Closure6 {
  IntSupplier makeFun(int x) {
    int i = 0;
    i++;
    x++;
    final int iFinal = i;
    final int xFinal = x;
    return () -> xFinal + iFinal;
  }
}
```

We never change `iFinal` and `xFinal` after assignment so the use of `final` here is redundant.

What if you're using references? We can change from `int` to `Integer`:

```
// functional/Closure7.java
// {WillNotCompile}
import java.util.function.*;

public class Closure7 {
  IntSupplier makeFun(int x) {
    Integer i = 0;
    i = i + 1;
    return () -> x + i;
  }
}
```

The compiler is still smart enough to see that `i` is being changed. The wrapper types might be getting special treatment, so let's try a `List`:

```
// functional/Closure8.java
import java.util.*;
import java.util.function.*;

public class Closure8 {
  Supplier<List<Integer>> makeFun() {
    final List<Integer> ai = new ArrayList<>();
    ai.add(1);
    return () -> ai;
  }
  public static void main(String[] args) {
    Closure8 c7 = new Closure8();
    List<Integer>
      l1 = c7.makeFun().get(),
      l2 = c7.makeFun().get();
    System.out.println(l1);
    System.out.println(l2);
    l1.add(42);
    l2.add(96);
    System.out.println(l1);
    System.out.println(l2);
  }
}
/* Output:
[1]
[1]
[1, 42]
```

```
[1, 96]
*/
```

This time it works: we modify the contents of the `List` without producing a compile-time error. When you look at the output from this example, it does seem pretty safe, because each time `makeFun()` is called, a brand new `ArrayList` is created and returned—which means it is not shared, so each generated closure has its own separate `ArrayList` and they can't interfere with each other.

And notice I've declared `ai` to be `final`, although for this example you can take `final` off and get the same results (try it!). The `final` keyword applied to object references only says that the reference doesn't get reassigned. It doesn't say you can't modify the object itself.

Looking at the difference between `Closure7.java` and `Closure8.java`, we see that `Closure7.java` actually has a reassignment of `i`. Maybe that's the trigger for the "effectively final" error message:

```
// functional/Closure9.java
// {WillNotCompile}
import java.util.*;
import java.util.function.*;

public class Closure9 {
  Supplier<List<Integer>> makeFun() {
    List<Integer> ai = new ArrayList<>();
    ai = new ArrayList<>(); // Reassignment
    return () -> ai;
  }
}
```

The reassignment does indeed trigger the message. If you only modify the object that's pointed to, Java accepts it. And that's probably safe as long as no one else ever gets the reference to that object (that would mean you have more than one entity that can modify the object, at which point things can get very confusing).[93]

There's a conundrum, however, if we now look back at `Closure1.java`. `i` is modified without compiler complaints. It is neither `final` nor "effectively final."

[93] This will make additional sense in the Concurrent Programming chapter, when you'll understand that mutating shared variables is "not thread-safe."

Because `i` is a member of the surrounding class, it's certainly safe to do this (other than the fact you are creating multiple functions that share mutable memory). Indeed, you can argue that no variable capture occurs in that case. And to be sure, the error message for `Closure3.java` specifically refers to *local* variables. Thus, the rule isn't as simple as saying "any variable defined outside the lambda must be final or effectively final." Instead, you must think in terms of *captured* variables being effectively final. If it's a field in an object, it has an independent lifetime and doesn't need any special capturing in order to exist later, when the lambda is called.

Inner Classes as Closures

We can duplicate our example using an anonymous inner class:

```
// functional/AnonymousClosure.java
import java.util.function.*;

public class AnonymousClosure {
  IntSupplier makeFun(int x) {
    int i = 0;
    // Same rules apply:
    // i++; // Not "effectively final"
    // x++; // Ditto
    return new IntSupplier() {
      public int getAsInt() { return x + i; }
    };
  }
}
```

It turns out that as long as there have been inner classes, there have been closures (Java 8 just makes closures easier). Prior to Java 8, the requirement was that `x` and `i` be explicitly declared `final`. With Java 8, the rule for inner classes has been relaxed to include "effectively `final`."

Function Composition

Function composition basically means "pasting functions together to create new functions," and it's commonly considered a part of functional programming. You

saw one example of function composition in `TransformFunction.java` with the use of `andThen()`. Some of the `java.util.function` interfaces contain methods to support function composition.[94]

Compositional Method	Supporting Interfaces
`andThen(argument)` Performs the original operation followed by the argument operation.	`Function` `BiFunction` `Consumer` `BiConsumer` `IntConsumer` `LongConsumer` `DoubleConsumer` `UnaryOperator` `IntUnaryOperator` `LongUnaryOperator` `DoubleUnaryOperator` `BinaryOperator`
`compose(argument)` Performs the argument operation followed by the original operation.	`Function` `UnaryOperator` `IntUnaryOperator` `LongUnaryOperator` `DoubleUnaryOperator`
`and(argument)` Short-circuiting logical AND of the original predicate and the argument predicate.	`Predicate` `BiPredicate` `IntPredicate` `LongPredicate` `DoublePredicate`
`or(argument)` Short-circuiting logical OR of the original predicate and the argument predicate.	`Predicate` `BiPredicate` `IntPredicate` `LongPredicate` `DoublePredicate`
`negate()` A predicate that is the logical negation of this predicate.	`Predicate` `BiPredicate` `IntPredicate` `LongPredicate` `DoublePredicate`

This example uses both `compose()` and `andThen()` from `Function`:

[94]The reason interfaces can support methods is that they are Java 8 `default` methods, which you'll learn about in the next chapter.

```
// functional/FunctionComposition.java
import java.util.function.*;

public class FunctionComposition {
  static Function<String, String>
    f1 = s -> {
      System.out.println(s);
      return s.replace('A', '_');
    },
    f2 = s -> s.substring(3),
    f3 = s -> s.toLowerCase(),
    f4 = f1.compose(f2).andThen(f3);
  public static void main(String[] args) {
    System.out.println(
      f4.apply("GO AFTER ALL AMBULANCES"));
  }
}
/* Output:
AFTER ALL AMBULANCES
_fter _ll _mbul_nces
*/
```

The important thing to see here is that we are creating a new function `f4`, which can then be called using `apply()` (almost) like any other function.[95]

By the time `f1` gets the `String`, it's already had the first three characters stripped off by `f2`. That's because the call to `compose(f2)` means `f2` gets called before `f1`.

Here's a demonstration of the logical operations for `Predicate`:

[95] Some languages, Python for example, allow composed functions to be called *exactly* like any other function. But this is Java, so we take what we can get.

```
// functional/PredicateComposition.java
import java.util.function.*;
import java.util.stream.*;

public class PredicateComposition {
  static Predicate<String>
    p1 = s -> s.contains("bar"),
    p2 = s -> s.length() < 5,
    p3 = s -> s.contains("foo"),
    p4 = p1.negate().and(p2).or(p3);
  public static void main(String[] args) {
    Stream.of("bar", "foobar", "foobaz", "fongopuckey")
      .filter(p4)
      .forEach(System.out::println);
  }
}
/* Output:
foobar
foobaz
*/
```

p4 takes all the predicates and combines them into a more complex predicate, which reads: "If the `String` does NOT contain 'bar' AND the length is less than 5, OR it contains 'foo', the result is `true`."

Because it produces such clear syntax, I've cheated a bit in `main()` and borrowed from the next chapter. First I create a "stream" (a sequence) of `String` objects, then feed each one to the `filter()` operation. `filter()` uses our p4 predicate to decide which object in the stream to keep, and which to discard. Finally, `forEach()` applies the `println` method reference to each surviving object.

You can see from the output how p4 works: anything with a "foo" survives, even if its length is greater than five. "fongopuckey" is too long and doesn't have a "bar" to save it.

Currying and Partial Evaluation

Currying[96] is named after Haskell Curry, one of its inventors and possibly the only computer person to have an important thing named after each of his names (the other

[96] https://en.wikipedia.org/wiki/Currying

is the *Haskell* programming language). Currying means starting with a function that takes multiple arguments and turning it into a sequence of functions, each of which only takes a single argument.

```
// functional/CurryingAndPartials.java
import java.util.function.*;

public class CurryingAndPartials {
   // Uncurried:
   static String uncurried(String a, String b) {
      return a + b;
   }
   public static void main(String[] args) {
      // Curried function:
      Function<String, Function<String, String>> sum =
         a -> b -> a + b;                          // [1]

      System.out.println(uncurried("Hi ", "Ho"));

      Function<String, String>
        hi = sum.apply("Hi ");                     // [2]
      System.out.println(hi.apply("Ho"));

      // Partial application:
      Function<String, String> sumHi =
        sum.apply("Hup ");
      System.out.println(sumHi.apply("Ho"));
      System.out.println(sumHi.apply("Hey"));
   }
}
/* Output:
Hi Ho
Hi Ho
Hup Ho
Hup Hey
*/
```

- **[1]** This is the tricky line: a cascade of arrows. And notice how, in the function interface declaration, `Function` has another `Function` as its second argument.
- **[2]** The goal of currying is to be able to create a new function by providing a single argument, so you now have an "argumented function" and the remaining

"free argument." In effect, you start with a two-argument function and end up with a one-argument function.

You can curry a three-argument function by adding another level:

```
// functional/Curry3Args.java
import java.util.function.*;

public class Curry3Args {
   public static void main(String[] args) {
      Function<String,
        Function<String,
          Function<String, String>>> sum =
            a -> b -> c -> a + b + c;
      Function<String,
        Function<String, String>> hi =
          sum.apply("Hi ");
      Function<String, String> ho =
        hi.apply("Ho ");
      System.out.println(ho.apply("Hup"));
   }
}
/* Output:
Hi Ho Hup
*/
```

For each level of arrow-cascading, you wrap another `Function` around the type declaration.

When dealing with primitives and boxing, use the appropriate functional interfaces:

```
// functional/CurriedIntAdd.java
import java.util.function.*;

public class CurriedIntAdd {
  public static void main(String[] args) {
    IntFunction<IntUnaryOperator>
      curriedIntAdd = a -> b -> a + b;
    IntUnaryOperator add4 = curriedIntAdd.apply(4);
    System.out.println(add4.applyAsInt(5));
  }
}
```

```
/* Output:
9
*/
```

You can find more examples of currying on the Internet. Usually these are in languages other than Java, but if you understand the basic concepts they should be fairly easy to translate.

Pure Functional Programming

It's possible, with much discipline, to write pure functional programs in languages without functional support, even in a language as primitive as C. Java makes it easier than that, but you must carefully make everything `final` and ensure that all your methods and functions have no side effects. Because Java is by nature *not* an immutable language, you don't get any help from the compiler if you make a mistake.

There are third-party tools to help you[97], but it's arguably easier to use a language like Scala or Clojure, designed from the start for immutability. These languages make it possible to have part of your project in Java, and if you must write in a pure functional style, other parts in Scala (which requires some discipline) or Clojure (which requires much less). Although you'll see in the Concurrent Programming chapter that Java does support parallelism, if that is a core part of your project you might consider using something like Scala or Clojure for at least part of the project.

Summary

Lambda expressions and method references do not turn Java into a functional language, but rather provide support for programming in a more functional style. They are a huge improvement to Java because they allow you to write more succinct, cleaner, and easier-to-understand code. In the following chapter you'll see how they enable *streams*. If you're like me, you'll love streams.

These features are likely to satisfy a large portion of Java programmers who have become antsy and jealous about new, more functional languages like Clojure and

[97] See, for example, Immutables and Mutability Detector.

Scala, and stem the flow of Java programmers to those languages (or at least prepare them better if they still decide to move).

Lambdas and method references are far from perfect, however—we forever pay the price of the rash decisions made by the Java designers in the heady, early days of the language. In particular, there is no generic lambda, so lambdas really aren't first class citizens in Java. This doesn't mean Java 8 isn't a big improvement, but it does mean that, like many Java features, you eventually reach a point where you start getting frustrated.

As you come up the learning curve, remember you can get help from IDEs like NetBeans, IntelliJ Idea, and Eclipse, which will suggest when you can use a lambda expression or method reference (and often rewrite the code for you!).

Streams

> Collections optimize the storage of objects. *Streams* are about processing groups of objects.

A stream is a sequence of elements that is not associated with any particular storage mechanism—indeed, we say that streams have "no storage."

Instead of iterating through elements in a collection, with a stream you draw elements from a pipe and operate on them. These pipes are typically strung together to form a pipeline of operations upon the stream.

Most of the time, the reason you store objects in a collection is to process them, so you'll find yourself moving away from collections as the primary focus of your programming, and towards streams.

One of the core benefits of streams is that they make your programs smaller and easier to understand. Lambda expressions and method references come into their own when used with streams. Streams make Java 8 particularly attractive.[98]

For example, suppose you want to display a random selection of unique `int`s between 5 and 20, sorted. The fact that you're sorting them might make you focus first on choosing a sorted collection, and work around that. But with streams, you simply state what you want done:

[98] I found a number of sites very useful when creating this chapter, including Java2s and LambdaFAQ.

```
// streams/Randoms.java
import java.util.*;

public class Randoms {
  public static void main(String[] args) {
    new Random(47)
      .ints(5, 20)
      .distinct()
      .limit(7)
      .sorted()
      .forEach(System.out::println);
  }
}
/* Output:
6
10
13
16
17
18
19
*/
```

We start by seeding the `Random` object (to produce identical output when the program runs). The `ints()` method produces a stream and is overloaded in a number of ways—two arguments set the bounds of the values produced. This produces a stream of random `ints`. We tell it to make them unique using the *intermediate stream operation* `distinct()`, then choose the first seven using `limit()`. Then we tell it we want the elements to be `sorted()`. Finally, we'd like to display each item so we use `forEach()`, which performs an operation on each stream object, according to the function we pass it. Here, we pass a method reference `System.out::println` which it uses to show each item on the console.

Note that `Randoms.java` declares no variables. Streams can model systems that have state without ever using assignment or mutable data, and this can be very helpful.

Declarative programming is a style where we state *what* we want done, rather than specifying *how*, and it's what you see in functional programming. Notice it's much more difficult to understand the *imperative* form:

```
// streams/ImperativeRandoms.java
import java.util.*;

public class ImperativeRandoms {
  public static void main(String[] args) {
    Random rand = new Random(47);
    SortedSet<Integer> rints = new TreeSet<>();
    while(rints.size() < 7) {
      int r = rand.nextInt(20);
      if(r < 5) continue;
      rints.add(r);
    }
    System.out.println(rints);
  }
}
/* Output:
[7, 8, 9, 11, 13, 15, 18]
*/
```

In `Randoms.java`, we didn't have to define any variables at all, but here we have three: `rand`, `rints` and `r`. The code is further complicated by the fact that `nextInt()` has no option for a lower bound—its built-in lower bound is always zero, so we must generate extra values and filter out the ones that are less than five.

Notice how you must study the code to figure out what's going on, whereas in `Randoms.java`, the code just *tells* you what it's doing. This clarity is one of the most compelling reasons for using Java 8 streams.

Explicitly writing the mechanics of iteration as in `ImperativeRandoms.java` is called *external iteration*. In `Randoms`, you don't see any of these mechanisms and so it is called *internal iteration*, a core characteristic of streams programming. Internal iteration produces more readable code, but it also makes it easier to use multiple processors: By loosening control of how iteration happens, you can hand that control over to a parallelizing mechanism. You'll learn about this in the Concurrent Programming chapter.

Another important aspect of streams is that they are *lazy*, which means they are only evaluated when absolutely necessary. You can think of a stream as a "delayed list." Because of *delayed evaluation*, streams enable us to represent very large (even infinite) sequences without memory concerns.

Java 8 Stream Support

The Java designers faced a conundrum. They had an existing set of libraries, used not only within the Java libraries themselves, but in millions of lines of code written by users. How would they integrate the new, fundamental concept of streams into existing libraries?

In simple cases like `Random`, they could just add more methods. As long as the existing methods were not changed, legacy code would not get disturbed.

The big challenge came from libraries that used interfaces. Collection classes are an essential part of this, because you want to convert collections into streams. But if you add a new method to an interface, you break every class that implements your interface but doesn't implement your new method.

The solution, introduced in Java 8, is `default` methods in interfaces, which were covered in the Interfaces chapter. With `default` methods, the Java designers could shoehorn stream methods into existing classes, and they added virtually every operation you might need. There are three types of operations: creating streams, modifying elements of a stream (*intermediate operations*), and consuming stream elements (*terminal operations*). This last type often means collecting elements of a stream (typically into a collection).

We'll look at each type of operation.

Stream Creation

You can easily turn a group of items into a stream using `Stream.of()` (`Bubble` is defined later in the chapter):

```
// streams/StreamOf.java
import java.util.stream.*;

public class StreamOf {
  public static void main(String[] args) {
    Stream.of(
      new Bubble(1), new Bubble(2), new Bubble(3))
      .forEach(System.out::println);
    Stream.of("It's ", "a ", "wonderful ",
      "day ", "for ", "pie!")
      .forEach(System.out::print);
    System.out.println();
    Stream.of(3.14159, 2.718, 1.618)
      .forEach(System.out::println);
  }
}
/* Output:
Bubble(1)
Bubble(2)
Bubble(3)
It's a wonderful day for pie!
3.14159
2.718
1.618
*/
```

In addition, every `Collection` can produce a stream using the `stream()` method:

```
// streams/CollectionToStream.java
import java.util.*;
import java.util.stream.*;

public class CollectionToStream {
  public static void main(String[] args) {
    List<Bubble> bubbles = Arrays.asList(
      new Bubble(1), new Bubble(2), new Bubble(3));
    System.out.println(
      bubbles.stream()
        .mapToInt(b -> b.i)
        .sum());

    Set<String> w = new HashSet<>(Arrays.asList(
```

```
      "It's a wonderful day for pie!".split(" ")));
    w.stream()
      .map(x -> x + " ")
      .forEach(System.out::print);
    System.out.println();

    Map<String, Double> m = new HashMap<>();
    m.put("pi", 3.14159);
    m.put("e", 2.718);
    m.put("phi", 1.618);
    m.entrySet().stream()
      .map(e -> e.getKey() + ": " + e.getValue())
      .forEach(System.out::println);
  }
}
/* Output:
6
a pie! It's for wonderful day
phi: 1.618
e: 2.718
pi: 3.14159
*/
```

After creating a `List<Bubble>`, we simply ask it for a `stream()`, the common method for all collections. The intermediate `map()` operation takes each element in a stream and applies an operation to create a new element, which it then passes on down the stream. The normal `map()` takes objects and produces objects, but there are special versions when the output stream holds a numeric type. Here, `mapToInt()` converts from an object stream to an `IntStream` containing `Integers`. There are similarly-named operations for `Float` and `Double`.

To define `w` we take a `String` and apply the `split()` function, which splits the `String` according to its argument. You'll see later that this argument can be quite sophisticated, but here we are just telling it to split at spaces.

To produce a stream from a `Map` collection, we first call `entrySet()` to produce a stream of objects that each contains both a key and its associated value, then pull that apart using `getKey()` and `getValue()`.

Random Number Streams

The Random class has been enhanced with a set of methods to produce streams:

```
// streams/RandomGenerators.java
import java.util.*;
import java.util.stream.*;

public class RandomGenerators {
  public static <T> void show(Stream<T> stream) {
      stream
        .limit(4)
        .forEach(System.out::println);
      System.out.println("++++++++");
  }
  public static void main(String[] args) {
    Random rand = new Random(47);
    show(rand.ints().boxed());
    show(rand.longs().boxed());
    show(rand.doubles().boxed());
    // Control the lower and upper bounds:
    show(rand.ints(10, 20).boxed());
    show(rand.longs(50, 100).boxed());
    show(rand.doubles(20, 30).boxed());
    // Control the stream size:
    show(rand.ints(2).boxed());
    show(rand.longs(2).boxed());
    show(rand.doubles(2).boxed());
    // Control the stream size and bounds:
    show(rand.ints(3, 3, 9).boxed());
    show(rand.longs(3, 12, 22).boxed());
    show(rand.doubles(3, 11.5, 12.3).boxed());
  }
}
/* Output:
-1172028779
1717241110
-2014573909
229403722
++++++++
2955289354441303771
3476817843704654257
```

```
-8917117694134521474
4941259272818818752
++++++++
0.2613610344283964
0.0508673570556899
0.8037155449603999
0.7620665811558285
++++++++
16
10
11
12
++++++++
65
99
54
58
++++++++
29.86777681078574
24.83968447804611
20.09247112332014
24.046793846338723
++++++++
1169976606
1947946283
++++++++
2970202997824602425
-2325326920272830366
++++++++
0.7024254510631527
0.6648552384607359
++++++++
6
7
7
++++++++
17
12
20
++++++++
12.27872414236691
11.732085449736195
```

```
12.196509449817267
++++++++
*/
```

To eliminate redundant code, I created the generic method `show(Stream<T> stream)` (I'm cheating a bit here by using the generics feature before its chapter, but the payoff is worth it). The `T` type parameter can be anything, so it works with `Integer`, `Long` and `Double`. However, the `Random` class only produces the primitive types `int`, `long` and `double`. Fortunately, the `boxed()` stream operation automatically converts the primitives to their boxed counterparts, thus enabling `show()` to accept the stream.

We can use `Random` to create a `Supplier` for any set of objects. Here's an example that supplies `String` objects, taken from this text file:

```
// streams/Cheese.dat
Not much of a cheese shop really, is it?
Finest in the district, sir.
And what leads you to that conclusion?
Well, it's so clean.
It's certainly uncontaminated by cheese.
```

We use the `Files` class to read all the lines from a file into a `List<String>`:

```
// streams/RandomWords.java
import java.util.*;
import java.util.stream.*;
import java.util.function.*;
import java.io.*;
import java.nio.file.*;

public class RandomWords implements Supplier<String> {
  List<String> words = new ArrayList<>();
  Random rand = new Random(47);
  RandomWords(String fname) throws IOException {
    List<String> lines =
      Files.readAllLines(Paths.get(fname));
    // Skip the first line:
    for(String line : lines.subList(1, lines.size())) {
      for(String word : line.split("[ .?,]+"))
        words.add(word.toLowerCase());
    }
```

```
  }
  @Override public String get() {
    return words.get(rand.nextInt(words.size()));
  }
  @Override public String toString() {
    return words.stream()
      .collect(Collectors.joining(" "));
  }
  public static void
  main(String[] args) throws Exception {
    System.out.println(
      Stream.generate(new RandomWords("Cheese.dat"))
        .limit(10)
        .collect(Collectors.joining(" ")));
  }
}
/* Output:
it shop sir the much cheese by conclusion district is
*/
```

Here you see a slightly more sophisticated use of `split()`. In the constructor, each `line` is `split()` on either a space or any of the punctuation characters defined within the square brackets. The `+` after the closing square bracket indicates "the previous thing, one or more times."

You'll note that the constructor uses imperative programming (external iteration) for its loops. In future examples you'll see how we eliminate even this. The older forms are not particularly bad, but it often just feels nicer to use streams everywhere.

In `toString()` and `main()` you see the `collect()` operation, which combines all the stream elements according to its argument. When you use `Collectors.joining()`, you get a `String` result, with each element separated by the argument to `joining()`. There are numerous other `Collectors` to produce different results.

In `main()`, we see a preview of `Stream.generate()`, which takes any `Supplier<T>` and produces a stream of `T` objects.

Ranges of `int`

The `IntStream` class provides a `range()` method to produce a stream that is a sequence of `int`s. This can be convenient when writing loops:

```
// streams/Ranges.java
import static java.util.stream.IntStream.*;

public class Ranges {
  public static void main(String[] args) {
    // The traditional way:
    int result = 0;
    for(int i = 10; i < 20; i++)
      result += i;
    System.out.println(result);

    // for-in with a range:
    result = 0;
    for(int i : range(10, 20).toArray())
      result += i;
    System.out.println(result);

    // Use streams:
    System.out.println(range(10, 20).sum());
  }
}
/* Output:
145
145
145
*/
```

The first approach shown in `main()` is the traditional way of writing a `for` loop. In the second approach, we create a `range()` and turn it into an array which then works in a *for-in* statement. If you're able, however, it's always nice to go full-streams as in the third approach. In each case, we sum the integers in the range, and, conveniently, there's a `sum()` operation for streams.

Note that `IntStream.range()` is more limited than `onjava.Range.range()`. Because of its optional third *step* argument, the latter has the ability to generate ranges that step by more than one, and that can count down from a higher value to a lower one.

To replace simple `for` loops, here's a `repeat()` utility:

```
// onjava/Repeat.java
package onjava;
import static java.util.stream.IntStream.*;

public class Repeat {
  public static void repeat(int n, Runnable action) {
    range(0, n).forEach(i -> action.run());
  }
}
```

The resulting loops are arguably cleaner:

```
// streams/Looping.java
import static onjava.Repeat.*;

public class Looping {
  static void hi() { System.out.println("Hi!"); }
  public static void main(String[] args) {
    repeat(3, () -> System.out.println("Looping!"));
    repeat(2, Looping::hi);
  }
}
/* Output:
Looping!
Looping!
Looping!
Hi!
Hi!
*/
```

It might not be worth it, however, to include and explain `repeat()` in your code. It seems like a reasonably transparent tool, but it depends on how your team and company works.

`generate()`

`RandomWords.java` used `Supplier<T>` with `Stream.generate()`. Here's a second example:

```
// streams/Generator.java
import java.util.*;
import java.util.function.*;
import java.util.stream.*;

public class Generator implements Supplier<String> {
  Random rand = new Random(47);
  char[] letters =
    "ABCDEFGHIJKLMNOPQRSTUVWXYZ".toCharArray();
  @Override public String get() {
    return "" + letters[rand.nextInt(letters.length)];
  }
  public static void main(String[] args) {
    String word = Stream.generate(new Generator())
      .limit(30)
      .collect(Collectors.joining());
    System.out.println(word);
  }
}
/* Output:
YNZBRNYGCFOWZNTCQRGSEGZMMJMROE
*/
```

The uppercase letters of the alphabet are selected using `Random.nextInt()`. The argument tells it the largest acceptable random number, so the array bounds are respected.

If you want to create a stream of identical objects, simply pass a lambda that produces those objects to `generate()`:

```
// streams/Duplicator.java
import java.util.stream.*;

public class Duplicator {
  public static void main(String[] args) {
    Stream.generate(() -> "duplicate")
      .limit(3)
      .forEach(System.out::println);
  }
}
/* Output:
duplicate
```

```
duplicate
duplicate
*/
```

Here's the `Bubble` class used in earlier examples in this chapter. Note it contains its own `static` generator method:

```
// streams/Bubble.java
import java.util.function.*;

public class Bubble {
  public final int i;
  public Bubble(int n) { i = n; }
  @Override public String toString() {
    return "Bubble(" + i + ")";
  }
  private static int count = 0;
  public static Bubble bubbler() {
    return new Bubble(count++);
  }
}
```

Because `bubbler()` is interface-compatible with `Supplier<Bubble>`, we can pass its method reference to `Stream.generate()`:

```
// streams/Bubbles.java
import java.util.stream.*;

public class Bubbles {
  public static void main(String[] args) {
    Stream.generate(Bubble::bubbler)
      .limit(5)
      .forEach(System.out::println);
  }
}
/* Output:
Bubble(0)
Bubble(1)
Bubble(2)
Bubble(3)
Bubble(4)
*/
```

This is an alternative approach to creating a separate factory class. In many ways it's neater, but it's a matter of taste and code organization—you can always just create a completely different factory class.

iterate()

`Stream.iterate()` starts with a seed (the first argument) and passes it to the method (the second argument). The result is added to the stream and also stored for use as the first argument the next time `iterate()` is called, and so on. We can `iterate()` a Fibonacci sequence (which you first encountered in the last chapter):

```
// streams/Fibonacci.java
import java.util.stream.*;

public class Fibonacci {
  int x = 1;
  Stream<Integer> numbers() {
    return Stream.iterate(0, i -> {
      int result = x + i;
      x = i;
      return result;
    });
  }
  public static void main(String[] args) {
    new Fibonacci().numbers()
      .skip(20) // Don't use the first 20
      .limit(10) // Then take 10 of them
      .forEach(System.out::println);
  }
}
/* Output:
6765
10946
17711
28657
46368
75025
121393
196418
317811
```

```
514229
*/
```

The Fibonacci sequence sums the last *two* elements in the sequence to produce the next one. `iterate()` only remembers the result, so we must use `x` to keep track of the other element.

In `main()`, we use the `skip()` operation, which you haven't seen before. It simply discards the number of stream elements specified by its argument. Here, we discard the first 20 items.

Stream Builders

In the *Builder* design pattern, you create a builder object, give it multiple pieces of construction information, and finally perform the "build" act. The `Stream` library provides such a `Builder`. Here, we revisit the process of reading a file and turning it into a stream of words:

```
// streams/FileToWordsBuilder.java
import java.io.*;
import java.nio.file.*;
import java.util.stream.*;

public class FileToWordsBuilder {
  Stream.Builder<String> builder = Stream.builder();
  public FileToWordsBuilder(String filePath)
  throws Exception {
    Files.lines(Paths.get(filePath))
      .skip(1) // Skip the comment line at the beginning
      .forEach(line -> {
        for(String w : line.split("[ .?,]+"))
          builder.add(w);
      });
  }
  Stream<String> stream() { return builder.build(); }
  public static void
  main(String[] args) throws Exception {
    new FileToWordsBuilder("Cheese.dat").stream()
      .limit(7)
      .map(w -> w + " ")
```

```
      .forEach(System.out::print);
  }
}
/* Output:
Not much of a cheese shop really
*/
```

Notice that the constructor adds all the words from the file (except the first line, which is the comment containing the file path information), but it doesn't call `build()`. This means, as long as you don't call `stream()`, you can continue to add words to the `builder` object. In a more complete version of this class, you might add a flag to see whether `build()` has been called, and a method to add further words if possible. Trying to add to a `Stream.Builder` after calling `build()` produces an exception.

Arrays

The `Arrays` class contains `static` methods named `stream()` that convert arrays to streams. We can rewrite `main()` from `interfaces/Machine.java` to create a stream and apply `execute()` to each element:

```
// streams/Machine2.java
import java.util.*;
import onjava.Operations;

public class Machine2 {
  public static void main(String[] args) {
    Arrays.stream(new Operations[] {
      () -> Operations.show("Bing"),
      () -> Operations.show("Crack"),
      () -> Operations.show("Twist"),
      () -> Operations.show("Pop")
    }).forEach(Operations::execute);
  }
}
/* Output:
Bing
Crack
Twist
Pop
*/
```

The `new Operations[]` expression dynamically creates a typed array of `Operations` objects.

The `stream()` methods can also produce an `IntStream`, `LongStream` and `DoubleStream`:

```
// streams/ArrayStreams.java
import java.util.*;
import java.util.stream.*;

public class ArrayStreams {
  public static void main(String[] args) {
    Arrays.stream(
      new double[] { 3.14159, 2.718, 1.618 })
      .forEach(n -> System.out.format("%f ", n));
    System.out.println();
    Arrays.stream(new int[] { 1, 3, 5 })
      .forEach(n -> System.out.format("%d ", n));
    System.out.println();
    Arrays.stream(new long[] { 11, 22, 44, 66 })
      .forEach(n -> System.out.format("%d ", n));
    System.out.println();
    // Select a subrange:
    Arrays.stream(
      new int[] { 1, 3, 5, 7, 15, 28, 37 }, 3, 6)
      .forEach(n -> System.out.format("%d ", n));
  }
}
/* Output:
3.141590 2.718000 1.618000
1 3 5
11 22 44 66
7 15 28
*/
```

The last call to `stream()` has two additional arguments: the first tells `stream()` where to start selecting elements from the array, and the second tells it where to stop. Each different type of `stream()` method also has this version.

Regular Expressions

Java's *regular expressions* are covered in the Strings chapter. Java 8 added a new method `splitAsStream()` to the `java.util.regex.Pattern` class, which takes a sequence of characters and splits it into a stream, according to the formula you hand it. There's a constraint, which is that the input is a `CharSequence`, so you cannot feed a stream into `splitAsStream()`.

We'll look again at the process of turning a file into words. This time, we use streams to turn the file into a single `String`, then *regular expressions* to split the `String` into a stream of words:

```
// streams/FileToWordsRegexp.java
import java.io.*;
import java.nio.file.*;
import java.util.stream.*;
import java.util.regex.Pattern;

public class FileToWordsRegexp {
  private String all;
  public FileToWordsRegexp(String filePath)
  throws Exception {
    all = Files.lines(Paths.get(filePath))
      .skip(1) // First (comment) line
      .collect(Collectors.joining(" "));
  }
  public Stream<String> stream() {
    return Pattern
      .compile("[ .,?]+").splitAsStream(all);
  }
  public static void
  main(String[] args) throws Exception {
    FileToWordsRegexp fw =
      new FileToWordsRegexp("Cheese.dat");
    fw.stream()
      .limit(7)
      .map(w -> w + " ")
      .forEach(System.out::print);
    fw.stream()
      .skip(7)
      .limit(2)
```

```
      .map(w -> w + " ")
      .forEach(System.out::print);
  }
}
/* Output:
Not much of a cheese shop really is it
*/
```

The constructor reads all the lines in the file (again, skipping the first comment line) and turns them into a single `String`). Now, when you call `stream()`, you get a stream as before, but this time you can come back and call `stream()` multiple times, creating a new stream from the stored `String` each time. The limit here is that the whole file must be stored in memory; most of the time that probably won't be an issue but it loses important benefits of streams:

1. They "don't require storage." Of course they actually require some internal storage, but it's only a fraction of the sequence, and nothing like what it takes to hold the entire sequence.
2. They are lazily evaluated.

Fortunately, we'll see how to solve this problem a bit later.

Intermediate Operations

These operations take objects in from one stream and feed objects out the back end as another stream, to be connected to other operations.

Tracing and Debugging

The `peek()` operation is intended as an aid to debugging. It allows you to view stream objects without modifying them:

```
// streams/Peeking.java

class Peeking  {
  public static void
  main(String[] args) throws Exception {
    FileToWords.stream("Cheese.dat")
      .skip(21)
      .limit(4)
      .map(w -> w + " ")
      .peek(System.out::print)
      .map(String::toUpperCase)
      .peek(System.out::print)
      .map(String::toLowerCase)
      .forEach(System.out::print);
  }
}
/* Output:
Well WELL well it IT it s S s so SO so
*/
```

`FileToWords` is defined shortly, but it acts like the versions we've seen already: producing a stream of `String` objects. We `peek()` at them as they travel through the pipeline.

Because `peek()` takes a function that conforms to the `Consumer` functional interface, which has no return value, it's not possible to replace the objects in the stream with different ones. You can only look at them.

Sorting Stream Elements

You saw the use of `sorted()` with the default comparison in `Randoms.java`. There's a second form of `sorted()` that takes a `Comparator` argument:

```
// streams/SortedComparator.java
import java.util.*;

public class SortedComparator {
  public static void
  main(String[] args) throws Exception {
    FileToWords.stream("Cheese.dat")
      .skip(10)
      .limit(10)
      .sorted(Comparator.reverseOrder())
      .map(w -> w + " ")
      .forEach(System.out::print);
  }
}
/* Output:
you what to the that sir leads in district And
*/
```

You can pass in a lambda function as the argument for `sorted()`, but there are also pre-defined `Comparators`—here we use one that reverses the "natural order."

Removing Elements

- `distinct()`: In `Randoms.java`, `distinct()` removed duplicates from the stream. Using `distinct()` is far less work than creating a `Set` to eliminate duplicates.
- `filter(Predicate)`: The filter operation keeps only the elements that produce `true` when passed to the argument: the *filter function*.

In this example, the filter function `isPrime()` detects prime numbers:

```
// streams/Prime.java
import java.util.stream.*;
import static java.util.stream.LongStream.*;

public class Prime {
  public static boolean isPrime(long n) {
    return rangeClosed(2, (long)Math.sqrt(n))
      .noneMatch(i -> n % i == 0);
  }
  public LongStream numbers() {
    return iterate(2, i -> i + 1)
      .filter(Prime::isPrime);
  }
  public static void main(String[] args) {
    new Prime().numbers()
      .limit(10)
      .forEach(n -> System.out.format("%d ", n));
    System.out.println();
    new Prime().numbers()
      .skip(90)
      .limit(10)
      .forEach(n -> System.out.format("%d ", n));
  }
}
/* Output:
2 3 5 7 11 13 17 19 23 29
467 479 487 491 499 503 509 521 523 541
*/
```

`rangeClosed()` includes the top boundary value. The `noneMatch()` operation returns `true` if no modulus produces zero, and `false` if any equal zero. `noneMatch()` quits after the first failure rather than trying them all.

Applying a function to every element

- `map(Function)`: Applies `Function` to every object in the input stream, passing on the result values as the output stream.
- `mapToInt(ToIntFunction)`: As above, but results in an `IntStream`.
- `mapToLong(ToLongFunction)`: As above, but results in a `LongStream`.
- `mapToDouble(ToDoubleFunction)`: As above, but results in a `DoubleStream`.

Here, we `map()` various `Functions` onto a stream of `Strings`:

```
// streams/FunctionMap.java
import java.util.*;
import java.util.stream.*;
import java.util.function.*;

class FunctionMap {
  static String[] elements = { "12", "", "23", "45" };
  static Stream<String> testStream() {
    return Arrays.stream(elements);
  }
  static void
  test(String descr, Function<String, String> func) {
    System.out.println(" ---( " + descr + " )---");
    testStream()
      .map(func)
      .forEach(System.out::println);
  }
  public static void main(String[] args) {

    test("add brackets", s -> "[" + s + "]");

    test("Increment", s -> {
      try {
        return Integer.parseInt(s) + 1 + "";
      } catch(NumberFormatException e) {
        return s;
      }
    });

    test("Replace", s -> s.replace("2", "9"));

    test("Take last digit", s -> s.length() > 0 ?
      s.charAt(s.length() - 1) + "" : s);
  }
}
/* Output:
 ---( add brackets )---
[12]
[]
[23]
```

```
[45]
 ---( Increment )---
13

24
46
 ---( Replace )---
19

93
45
 ---( Take last digit )---
2

3
5
*/
```

In the "Increment" test, we use `Integer.parseInt()` to attempt to turn the `String` into an `Integer`. If the `String` can't be represented as an `Integer` it throws a `NumberFormatException` and we just fall back to putting the original `String` back on the output stream.

In the above example, the `map()` maps from a `String` to a `String`, but there's no reason you can't produce a different type than you take in, thus changing the type of the stream from that point on. Here's what it looks like:

```
// streams/FunctionMap2.java
// Different input and output types
import java.util.*;
import java.util.stream.*;

class Numbered {
  final int n;
  Numbered(int n) { this.n = n; }
  @Override public String toString() {
    return "Numbered(" + n + ")";
  }
}

class FunctionMap2 {
  public static void main(String[] args) {
```

```
    Stream.of(1, 5, 7, 9, 11, 13)
      .map(Numbered::new)
      .forEach(System.out::println);
  }
}
/* Output:
Numbered(1)
Numbered(5)
Numbered(7)
Numbered(9)
Numbered(11)
Numbered(13)
*/
```

We take `int`s and turn them into `Numbereds` using the constructor `Numbered::new`.

If the result type produced by `Function` is one of the numeric types, you must use the appropriate `mapTo`-operations instead:

```
// streams/FunctionMap3.java
// Producing numeric output streams
import java.util.*;
import java.util.stream.*;

class FunctionMap3 {
  public static void main(String[] args) {
    Stream.of("5", "7", "9")
      .mapToInt(Integer::parseInt)
      .forEach(n -> System.out.format("%d ", n));
    System.out.println();
    Stream.of("17", "19", "23")
      .mapToLong(Long::parseLong)
      .forEach(n -> System.out.format("%d ", n));
    System.out.println();
    Stream.of("17", "1.9", ".23")
      .mapToDouble(Double::parseDouble)
      .forEach(n -> System.out.format("%f ", n));
  }
}
/* Output:
5 7 9
17 19 23
```

```
17.000000 1.900000 0.230000
*/
```

It's unfortunate that the Java designers didn't make the initial effort to eliminate primitive types.

Combining Streams During `map()`

Suppose you've got a stream of incoming elements, and you're applying a `map()` function to them. You've found some lovely functionality for your function you can't find anywhere else, but there's a problem: that functionality produces a stream. What you want is to produce a stream of elements, but instead you're producing a stream of streams of elements.

`flatMap()` does two things: it takes your stream-producing function and applies it to incoming elements (just like `map()` does), then it takes each stream and "flattens" it into elements. So what comes out is just elements.

- `flatMap(Function)`: Use when `Function` produces a stream.
- `flatMapToInt(Function)`: For a `Function` that produces an `IntStream`.
- `flatMapToLong(Function)`: For a `Function` that produces a `LongStream`.
- `flatMapToDouble(Function)`: For a `Function` that produces a `DoubleStream`.

To see how this works, we'll start with a contrived function for `map()`, which takes in an `Integer` and produces a stream of `Strings`:

```
// streams/StreamOfStreams.java
import java.util.stream.*;

public class StreamOfStreams {
  public static void main(String[] args) {
    Stream.of(1, 2, 3)
      .map(i -> Stream.of("Gonzo", "Kermit", "Beaker"))
      .map(e-> e.getClass().getName())
      .forEach(System.out::println);
  }
}
/* Output:
java.util.stream.ReferencePipeline$Head
java.util.stream.ReferencePipeline$Head
java.util.stream.ReferencePipeline$Head
*/
```

We were naively hoping for a stream of `String`, but what we got instead was a stream of "heads" to other streams. We can easily solve this with `flatMap()`:

```
// streams/FlatMap.java
import java.util.stream.*;

public class FlatMap {
  public static void main(String[] args) {
    Stream.of(1, 2, 3)
      .flatMap(
        i -> Stream.of("Gonzo", "Fozzie", "Beaker"))
      .forEach(System.out::println);
  }
}
/* Output:
Gonzo
Fozzie
Beaker
Gonzo
Fozzie
Beaker
Gonzo
Fozzie
Beaker
*/
```

So each stream that comes back from the mapping is automatically flattened into its component `String`s.

Here's another demonstration. We start with a stream of integer, then use each one to create that many random numbers:

```
// streams/StreamOfRandoms.java
import java.util.*;
import java.util.stream.*;

public class StreamOfRandoms {
  static Random rand = new Random(47);
  public static void main(String[] args) {
    Stream.of(1, 2, 3, 4, 5)
      .flatMapToInt(i -> IntStream.concat(
        rand.ints(0, 100).limit(i), IntStream.of(-1)))
      .forEach(n -> System.out.format("%d ", n));
  }
}
/* Output:
58 -1 55 93 -1 61 61 29 -1 68 0 22 7 -1 88 28 51 89 9
-1
*/
```

I've introduced `concat()` here, which combines two streams in argument order. So, at the end of each stream of random `Integer`, I'm adding a `-1` to use as a marker, so you can see that the final stream is indeed being created from a group of flattened streams.

Because `rand.ints()` produces an `IntStream`, I must use the special `Integer` versions of `flatMap()`, `concat()`, and `of()`.

Let's take one more look at the task of breaking a file into a stream of words. Our last encounter was `FileToWordsRegexp.java`, which had the problem that it required us to read the whole file into a `List` of lines—thus requiring storage for that `List`. What we really want is to create a stream of words without requiring intermediate storage. Once again, this is exactly the problem solved by `flatMap()`:

```
// streams/FileToWords.java
import java.nio.file.*;
import java.util.stream.*;
import java.util.regex.Pattern;

public class FileToWords {
  public static Stream<String> stream(String filePath)
  throws Exception {
    return Files.lines(Paths.get(filePath))
      .skip(1) // First (comment) line
      .flatMap(line ->
        Pattern.compile("\\W+").splitAsStream(line));
  }
}
```

`stream()` is now a static method because it can accomplish the whole stream-creation process by itself.

Note the use of `\\W+` as the regular-expression pattern. The `\\W` means a "non-word character," and the `+` means "one or more." Lowercase `\\w` refers to a "word character."

The problem we had before was that `Pattern.compile().splitAsStream()` produces a stream result, which means invoking `map()` on the incoming stream of lines produces a stream of streams of words, when what we want is simply a stream of words. Fortunately, `flatMap()` flattens a stream of streams of elements into a simple stream of elements. Alternatively, we could have used `String.split()`, which produces an array, which can be converted using `Arrays.stream()`:

```
.flatMap(line -> Arrays.stream(line.split("\\W+"))))
```

Because we have a true stream (rather than a stream based on a stored collection as in `FileToWordsRegexp.java`), every time we want a new stream we must create it from scratch, because it can't be reused:

```
// streams/FileToWordsTest.java
import java.util.stream.*;

public class FileToWordsTest {
  public static void
  main(String[] args) throws Exception {
    FileToWords.stream("Cheese.dat")
      .limit(7)
      .forEach(s -> System.out.format("%s ", s));
    System.out.println();
    FileToWords.stream("Cheese.dat")
      .skip(7)
      .limit(2)
      .forEach(s -> System.out.format("%s ", s));
  }
}
/* Output:
Not much of a cheese shop really
is it
*/
```

Here, the `%s` in `System.out.format()` indicates that the argument is a `String`.

Optional

Before we can look at terminal operations, we must consider what happens if you ask for an object in a stream and there's nothing there. We like to connect up our streams for the "happy path" and assume nothing will break. Putting a `null` in a stream is a good way to break it. Is there some kind of object we can use that will act as a holder for a stream element, but can also kindly tell us (that is, no exceptions) if the element we're looking for isn't there?

This idea is implemented as the `Optional` type. Certain standard stream operations return `Optional` objects because they cannot guarantee the desired result will exist. These include:

- `findFirst()` returns an `Optional` containing the first element. If the stream is empty, it returns `Optional.empty`.

- `findAny()` returns an `Optional` containing any element, or `Optional.empty` if the stream is empty.
- `max()` and `min()` return an `Optional` containing the maximum or minimum values in the stream, or an `Optional.empty` if the stream is empty.
- The version of `reduce()` that does *not* start with an "identity" object (the "identity" object becomes the default result in the other version of `reduce()` so there's no risk of an empty result) wraps its return value in an `Optional`.
- For the numerical streams `IntStream`, `LongStream` and `DoubleStream`, the `average()` operation wraps its result in an `Optional` in case the stream is empty.

Here are simple tests of all of these operations upon empty streams:

```
// streams/OptionalsFromEmptyStreams.java
import java.util.*;
import java.util.stream.*;

class OptionalsFromEmptyStreams {
  public static void main(String[] args) {
    System.out.println(Stream.<String>empty()
      .findFirst());
    System.out.println(Stream.<String>empty()
      .findAny());
    System.out.println(Stream.<String>empty()
      .max(String.CASE_INSENSITIVE_ORDER));
    System.out.println(Stream.<String>empty()
      .min(String.CASE_INSENSITIVE_ORDER));
    System.out.println(Stream.<String>empty()
      .reduce((s1, s2) -> s1 + s2));
    System.out.println(IntStream.empty()
      .average());
  }
}
/* Output:
Optional.empty
Optional.empty
Optional.empty
Optional.empty
Optional.empty
OptionalDouble.empty
*/
```

Instead of throwing an exception because the stream is empty, you get an `Optional.empty` object. `Optional` has a `toString()` which displays useful information.

Note the creation of the empty stream via `Stream.<String>empty()`. If you just say `Stream.empty()` without any context information, Java doesn't know what the type is; this syntax solves the problem. If the compiler has enough context information, as in:

```
Stream<String> s = Stream.empty();
```

It can infer the type for the `empty()` call.

This example shows the two basic activities for an `Optional`:

```
// streams/OptionalBasics.java
import java.util.*;
import java.util.stream.*;

class OptionalBasics {
  static void test(Optional<String> optString) {
    if(optString.isPresent())
      System.out.println(optString.get());
    else
      System.out.println("Nothing inside!");
  }
  public static void main(String[] args) {
    test(Stream.of("Epithets").findFirst());
    test(Stream.<String>empty().findFirst());
  }
}
/* Output:
Epithets
Nothing inside!
*/
```

When you receive an `Optional`, you first discover whether there's anything inside by calling `isPresent()`. If there is, you fetch it using `get()`.

Convenience Functions

There are a number of convenience functions for unpacking `Optionals`, which simplifies the above process of "checking and doing something with the contained object":

- `ifPresent(Consumer)`: Call the `Consumer` with the value if it's there, otherwise do nothing.
- `orElse(otherObject)`: Produce the object if it's there, otherwise produce `otherObject`.
- `orElseGet(Supplier)`: Produce the object if it's there, otherwise produce a replacement object using the `Supplier` function.
- `orElseThrow(Supplier)`: Produce the object if it's there, otherwise produce an exception using the `Supplier` function.

Here are simple demonstrations for the different convenience functions:

```
// streams/Optionals.java
import java.util.*;
import java.util.stream.*;
import java.util.function.*;

public class Optionals {
  static void basics(Optional<String> optString) {
    if(optString.isPresent())
      System.out.println(optString.get());
    else
      System.out.println("Nothing inside!");
  }
  static void ifPresent(Optional<String> optString) {
    optString.ifPresent(System.out::println);
  }
  static void orElse(Optional<String> optString) {
    System.out.println(optString.orElse("Nada"));
  }
  static void orElseGet(Optional<String> optString) {
    System.out.println(
      optString.orElseGet(() -> "Generated"));
  }
  static void orElseThrow(Optional<String> optString) {
```

```
    try {
      System.out.println(optString.orElseThrow(
        () -> new Exception("Supplied")));
    } catch(Exception e) {
      System.out.println("Caught " + e);
    }
  }
  static void test(String testName,
    Consumer<Optional<String>> cos) {
    System.out.println(" === " + testName + " === ");
    cos.accept(Stream.of("Epithets").findFirst());
    cos.accept(Stream.<String>empty().findFirst());
  }
  public static void main(String[] args) {
    test("basics", Optionals::basics);
    test("ifPresent", Optionals::ifPresent);
    test("orElse", Optionals::orElse);
    test("orElseGet", Optionals::orElseGet);
    test("orElseThrow", Optionals::orElseThrow);
  }
}
/* Output:
 === basics ===
Epithets
Nothing inside!
 === ifPresent ===
Epithets
 === orElse ===
Epithets
Nada
 === orElseGet ===
Epithets
Generated
 === orElseThrow ===
Epithets
Caught java.lang.Exception: Supplied
*/
```

The `test()` method prevents code duplication by taking a `Consumer` that matches all the example methods.

`orElseThrow()` uses the `catch` keyword to capture the exception that is thrown by

`orElseThrow()`. You'll learn about this in detail in the Exceptions chapter.

Creating `Optionals`

When you're writing your own code that produces `Optionals`, there are three `static` methods you can use:

- `empty()`: Produces an `Optional` with nothing inside.
- `of(value)`: If you already know that `value` is not `null`, use this to wrap it in an `Optional`.
- `ofNullable(value)`: Use this if you don't know that `value` is not `null`. It automatically produces `Optional.empty` if `value` is `null`, and otherwise wraps `value` inside an `Optional`.

You can see how these work:

```
// streams/CreatingOptionals.java
import java.util.*;
import java.util.stream.*;
import java.util.function.*;

class CreatingOptionals {
  static void
  test(String testName, Optional<String> opt) {
    System.out.println(" === " + testName + " === ");
    System.out.println(opt.orElse("Null"));
  }
  public static void main(String[] args) {
    test("empty", Optional.empty());
    test("of", Optional.of("Howdy"));
    try {
      test("of", Optional.of(null));
    } catch(Exception e) {
      System.out.println(e);
    }
    test("ofNullable", Optional.ofNullable("Hi"));
    test("ofNullable", Optional.ofNullable(null));
  }
}
/* Output:
```

```
 === empty ===
Null
 === of ===
Howdy
java.lang.NullPointerException
 === ofNullable ===
Hi
 === ofNullable ===
Null
*/
```

If we try to create an `Optional` by passing `null` to `of()`, it blows up. `ofNullable()` handles `null` gracefully, so it seems the safest one to use.

Operations on `Optional` Objects

Three methods enable post-processing on `Optionals`, so if your stream pipeline produces an `Optional` you can do one more thing at the end:

- `filter(Predicate)`: Apply the `Predicate` to the contents of the `Optional` and return the result. If the `Optional` fails the `Predicate`, convert it to `empty`. If the `Optional` is already `empty`, just pass it through.
- `map(Function)`: If the `Optional` is not `empty`, apply `Function` to the contents of the `Optional` and return the result. Otherwise, pass through the `Optional.empty`.
- `flatMap(Function)`: Just like `map()`, but the supplied mapping function wraps the results in `Optional` objects so `flatMap()` doesn't do any wrapping at the end.

None of these are available for the numeric `Optionals`.

The normal stream `filter()` removes elements from the stream if the `Predicate` returns `false`. `Optional.filter()` doesn't delete the `Optional` if the `Predicate` fails—it leaves it, but converts it to `empty`. This example explores `filter()`:

```
// streams/OptionalFilter.java
import java.util.*;
import java.util.stream.*;
import java.util.function.*;

class OptionalFilter {
  static String[] elements = {
    "Foo", "", "Bar", "Baz", "Bingo"
  };
  static Stream<String> testStream() {
    return Arrays.stream(elements);
  }
  static void
  test(String descr, Predicate<String> pred) {
    System.out.println(" ---( " + descr + " )---");
    for(int i = 0; i <= elements.length; i++) {
      System.out.println(
        testStream()
          .skip(i)
          .findFirst()
          .filter(pred));
    }
  }
  public static void main(String[] args) {
    test("true", str -> true);
    test("false", str -> false);
    test("str != \"\"", str -> str != "");
    test("str.length() == 3", str -> str.length() == 3);
    test("startsWith(\"B\")",
         str -> str.startsWith("B"));
  }
}
/* Output:
 ---( true )---
Optional[Foo]
Optional[]
Optional[Bar]
Optional[Baz]
Optional[Bingo]
Optional.empty
 ---( false )---
Optional.empty
```

```
Optional.empty
Optional.empty
Optional.empty
Optional.empty
Optional.empty
 ---( str != "" )---
Optional[Foo]
Optional.empty
Optional[Bar]
Optional[Baz]
Optional[Bingo]
Optional.empty
 ---( str.length() == 3 )---
Optional[Foo]
Optional.empty
Optional[Bar]
Optional[Baz]
Optional.empty
Optional.empty
 ---( startsWith("B") )---
Optional.empty
Optional.empty
Optional[Bar]
Optional[Baz]
Optional[Bingo]
Optional.empty
*/
```

Although the output looks like a stream, pay special attention to the `for` loop inside `test()`. It's restarting the stream each time through the `for` loop, and skipping the number of elements set by the `for`-loop index, which is what makes it end up on each successive element in the stream. Then it does a `findFirst()` to take the first remaining element, which comes back wrapped in an `Optional`.

Unlike the usual `for` loop, this index isn't bounded by `i < elements.length`, but rather `i <= elements.length`, so the final element actually overruns the stream. Conveniently, this automatically becomes an `Optional.empty`, which you see at the end of each test.

Like `map()`, `Optional.map()` applies a function, but in the case of `Optional`, it only applies the mapping function if the `Optional` is not `empty`. It also extracts the

contents of the `Optional` to hand to the mapping function:

```
// streams/OptionalMap.java
import java.util.*;
import java.util.stream.*;
import java.util.function.*;

class OptionalMap {
  static String[] elements = { "12", "", "23", "45" };
  static Stream<String> testStream() {
    return Arrays.stream(elements);
  }
  static void
  test(String descr, Function<String, String> func) {
    System.out.println(" ---( " + descr + " )---");
    for(int i = 0; i <= elements.length; i++) {
      System.out.println(
        testStream()
          .skip(i)
          .findFirst() // Produces an Optional
          .map(func));
    }
  }
  public static void main(String[] args) {

    // If Optional is not empty, map() first extracts
    // the contents which it then passes
    // to the function:

    test("Add brackets", s -> "[" + s + "]");

    test("Increment", s -> {
      try {
        return Integer.parseInt(s) + 1 + "";
      } catch(NumberFormatException e) {
        return s;
      }
    });

    test("Replace", s -> s.replace("2", "9"));

    test("Take last digit", s -> s.length() > 0 ?
```

```
      s.charAt(s.length() - 1) + "" : s);
  }
  // After the function is finished, map() wraps the
  // result in an Optional before returning it:
}
/* Output:
 ---( Add brackets )---
Optional[[12]]
Optional[[]]
Optional[[23]]
Optional[[45]]
Optional.empty
 ---( Increment )---
Optional[13]
Optional[]
Optional[24]
Optional[46]
Optional.empty
 ---( Replace )---
Optional[19]
Optional[]
Optional[93]
Optional[45]
Optional.empty
 ---( Take last digit )---
Optional[2]
Optional[]
Optional[3]
Optional[5]
Optional.empty
*/
```

The result of the mapping function is automatically wrapped back into an `Optional`. As you can see, an `Optional.empty` is simply passed through, without applying the mapping function.

The `flatMap()` for `Optional` is applied to a mapping function that already produces an `Optional`, so `flatMap()` doesn't wrap the result in an `Optional`, the way `map()` does:

```
// streams/OptionalFlatMap.java
import java.util.*;
import java.util.stream.*;
import java.util.function.*;

class OptionalFlatMap {
  static String[] elements = { "12", "", "23", "45" };
  static Stream<String> testStream() {
    return Arrays.stream(elements);
  }
  static void test(String descr,
    Function<String, Optional<String>> func) {
    System.out.println(" ---( " + descr + " )---");
    for(int i = 0; i <= elements.length; i++) {
      System.out.println(
        testStream()
          .skip(i)
          .findFirst()
          .flatMap(func));
    }
  }
  public static void main(String[] args) {

    test("Add brackets",
         s -> Optional.of("[" + s + "]"));

    test("Increment", s -> {
      try {
        return Optional.of(
          Integer.parseInt(s) + 1 + "");
      } catch(NumberFormatException e) {
        return Optional.of(s);
      }
    });

    test("Replace",
      s -> Optional.of(s.replace("2", "9")));

    test("Take last digit",
      s -> Optional.of(s.length() > 0 ?
        s.charAt(s.length() - 1) + ""
        : s));
```

```
  }
}
/* Output:
 ---( Add brackets )---
Optional[[12]]
Optional[[]]
Optional[[23]]
Optional[[45]]
Optional.empty
 ---( Increment )---
Optional[13]
Optional[]
Optional[24]
Optional[46]
Optional.empty
 ---( Replace )---
Optional[19]
Optional[]
Optional[93]
Optional[45]
Optional.empty
 ---( Take last digit )---
Optional[2]
Optional[]
Optional[3]
Optional[5]
Optional.empty
*/
```

Like `map()`, `flatMap()` unpacks the contents of non-empty `Optionals` to hand to the mapping function. The only difference is that `flatMap()` doesn't wrap the result in an `Optional`, because the mapping function has already done that. In the above example, I've explicitly done the wrapping inside each mapping function, but clearly `Optional.flatMap()` is designed for functions already producing `Optionals` by themselves.

Streams of `Optionals`

Suppose you have a generator that might produce `null` values. If you create a stream of these using that generator, you'll naturally want to wrap the elements in `Optionals`. Here's what it looks like:

```
// streams/Signal.java
import java.util.*;
import java.util.stream.*;
import java.util.function.*;

public class Signal {
  private final String msg;
  public Signal(String msg) { this.msg = msg; }
  public String getMsg() { return msg; }
  @Override public String toString() {
    return "Signal(" + msg + ")";
  }
  static Random rand = new Random(47);
  public static Signal morse() {
    switch(rand.nextInt(4)) {
      case 1: return new Signal("dot");
      case 2: return new Signal("dash");
      default: return null;
    }
  }
  public static Stream<Optional<Signal>> stream() {
    return Stream.generate(Signal::morse)
      .map(signal -> Optional.ofNullable(signal));
  }
}
```

When you use this stream, you'll have to figure out how you want to unpack the `Optionals`:

```
// streams/StreamOfOptionals.java
import java.util.*;
import java.util.stream.*;

public class StreamOfOptionals {
  public static void main(String[] args) {
    Signal.stream()
      .limit(10)
      .forEach(System.out::println);
    System.out.println(" ---");
    Signal.stream()
      .limit(10)
      .filter(Optional::isPresent)
```

```
      .map(Optional::get)
      .forEach(System.out::println);
  }
}
/* Output:
Optional[Signal(dash)]
Optional[Signal(dot)]
Optional[Signal(dash)]
Optional.empty
Optional.empty
Optional[Signal(dash)]
Optional.empty
Optional[Signal(dot)]
Optional[Signal(dash)]
Optional[Signal(dash)]
 ---
Signal(dot)
Signal(dot)
Signal(dash)
Signal(dash)
*/
```

Here, I use `filter()` to keep only the non-`empty` `Optionals`, then use `map()` to `get()` the values. Because each situation requires you to decide the meaning of "no value," you usually need a different approach for each application.

Terminal Operations

These operations take a stream and produce a final result; they do not feed anything to a back-end stream. Thus, a terminal operation is always the last thing you can do within a pipeline.

Convert to an Array

- `toArray()`: Converts the stream elements into an array of the proper type.
- `toArray(generator)`: The `generator` is for allocating your own array storage, in special cases.

This is useful if the stream operations produce something you must use in array form. For example, suppose we want to capture random numbers in a way that we can reuse them as a stream, such that we get the identical stream each time. We can do this by storing them in an array:

```
// streams/RandInts.java
package streams;
import java.util.*;
import java.util.stream.*;

public class RandInts {
  private static int[] rints =
    new Random(47).ints(0, 1000).limit(100).toArray();
  public static IntStream rands() {
    return Arrays.stream(rints);
  }
}
```

A stream of 100 random `int`s between 0 and 1000 is converted to an array and stored in `rints` so that each time you call `rands()` you get a repeat of the same stream.

Apply a Final Operation to Every Element

- `forEach(Consumer)`: You've already seen this used many times with `System.out::println` as the `Consumer` function.
- `forEachOrdered(Consumer)`: This version ensures that the order on which the elements are operated on by `forEach` is the original stream order.

The first form is explicitly designed to operate on elements in any order, which only makes sense if you introduce the `parallel()` operation. We won't look at this in depth until the Concurrent Programming chapter, but here's a quick introduction: `parallel()` tells Java to try to run operations on multiple processors. It can do this precisely because we use streams—it can split the stream into multiple streams (often, one stream per processor) and run each stream on a different processor. Because we use internal iteration rather than external iteration, this is possible.

Before you get too excited about the seeming ease of `parallel()`, it's actually rather tricky to use, so hold off until we get to the Concurrent Programming chapter.

We can get an idea of the effect and of the need for `forEachOrdered(Consumer)` by introducing `parallel()` into an example:

```
// streams/ForEach.java
import java.util.*;
import java.util.stream.*;
import static streams.RandInts.*;

public class ForEach {
  static final int SZ = 14;
  public static void main(String[] args) {
    rands().limit(SZ)
      .forEach(n -> System.out.format("%d ", n));
    System.out.println();
    rands().limit(SZ)
      .parallel()
      .forEach(n -> System.out.format("%d ", n));
    System.out.println();
    rands().limit(SZ)
      .parallel()
      .forEachOrdered(n -> System.out.format("%d ", n));
  }
}
/* Output:
258 555 693 861 961 429 868 200 522 207 288 128 551 589
551 589 861 555 288 128 429 207 693 200 258 522 868 961
258 555 693 861 961 429 868 200 522 207 288 128 551 589
*/
```

I've isolated `sz` to make it easy to try different sizes. However, even a `sz` of `14` produces interesting results. In the first stream, we don't use `parallel()` and so display the results in the order they appear from `rands()`. The second stream *does* introduce `parallel()` and, even for such a small stream, you see that the output is not in the same order as before. That's because multiple processors are working on the problem, and if you run the program multiple times you'll see that this output is different, due to the non-deterministic factors produced by having more than one processor working on the problem at the same time.

The final stream still uses `parallel()` but forces the result back into its original order using `forEachOrdered()`. Thus, using `forEachOrdered()` for non-`parallel()` streams doesn't have any effect.

Collecting

- `collect(Collector)`: Uses the `Collector` to accumulate stream elements into a result collection.
- `collect(Supplier, BiConsumer, BiConsumer)`: As above, but `Supplier` creates a new result collection, the first `BiConsumer` is a function for including the next element into the result, and the second `BiConsumer` is used for combining two values.

You've only seen a few examples of `Collectors` objects. If you look at the documentation for `java.util.stream.Collectors`, you'll see that some of them are quite sophisticated. For example, we can collect into any specific kind of collection. Suppose we want our items to end up inside of a `TreeSet` so they are always sorted. There's no specific `toTreeSet()` method in `Collectors`, but you can use `Collectors.toCollection()` and hand it the constructor reference for any type of `Collection`. This program pulls the words from a file into a `TreeSet`:

```
// streams/TreeSetOfWords.java
import java.util.*;
import java.nio.file.*;
import java.util.stream.*;

public class TreeSetOfWords {
  public static void
  main(String[] args) throws Exception {
    Set<String> words2 =
      Files.lines(Paths.get("TreeSetOfWords.java"))
        .flatMap(s -> Arrays.stream(s.split("\\W+")))
        .filter(s -> !s.matches("\\d+")) // No numbers
        .map(String::trim)
        .filter(s -> s.length() > 2)
        .limit(100)
        .collect(Collectors.toCollection(TreeSet::new));
    System.out.println(words2);
  }
}
/* Output:
[Arrays, Collectors, Exception, Files, Output, Paths,
Set, String, System, TreeSet, TreeSetOfWords, args,
class, collect, file, filter, flatMap, get, import,
```

```
java, length, limit, lines, main, map, matches, new,
nio, numbers, out, println, public, split, static,
stream, streams, throws, toCollection, trim, util,
void, words2]
*/
```

`Files.lines()` opens the `Path` and turns it into a `Stream` of lines. The next line splits those lines on boundaries of one or more non-word characters (`\\W+`), which produces an array which is turned into a `Stream` with `Arrays.stream()`, and the result is flat-mapped back into a `Stream` of words. The `matches(\\d+)` finds and removes `Strings` that are *all* digits (note that `words2` makes it through). Next we apply `String.trim()` to shave off any surrounding whitespace, `filter()` out any words less than a length of three, take only the first 100 words, and finally put them into a `TreeSet`.

We can produce a `Map` from a stream:

```
// streams/MapCollector.java
import java.util.*;
import java.util.stream.*;

class Pair {
  public final Character c;
  public final Integer i;
  Pair(Character c, Integer i) {
    this.c = c;
    this.i = i;
  }
  public Character getC() { return c; }
  public Integer getI() { return i; }
  @Override public String toString() {
    return "Pair(" + c + ", " + i + ")";
  }
}

class RandomPair {
  Random rand = new Random(47);
  // An infinite iterator of random capital letters:
  Iterator<Character> capChars = rand.ints(65,91)
    .mapToObj(i -> (char)i)
    .iterator();
```

```
  public Stream<Pair> stream() {
    return rand.ints(100, 1000).distinct()
      .mapToObj(i -> new Pair(capChars.next(), i));
  }
}

public class MapCollector {
  public static void main(String[] args) {
    Map<Integer, Character> map =
      new RandomPair().stream()
        .limit(8)
        .collect(
          Collectors.toMap(Pair::getI, Pair::getC));
    System.out.println(map);
  }
}
/* Output:
{688=W, 309=C, 293=B, 761=N, 858=N, 668=G, 622=F,
751=N}
*/
```

`Pair` is just a basic data object. `RandomPair` creates a stream of randomly-generated `Pair` objects. It would be nice if we could just somehow combine two streams, but Java fights us on this one. So I create a stream of `ints` and use `mapToObj` to turn that into a stream of `Pairs`. The `capChars` randomly-generated `Iterator` of capital letters starts as a stream, then the `iterator()` method allows us to use it in the `stream()` method. As far as I can tell, this is the only way to combine more than one stream to produce a new stream of objects.

Here, we use the simplest form of `Collectors.toMap()`, which just needs functions to fetch keys and values from the stream. There are additional forms, one of which takes a function to handle the case when you get a key collision.

Most of the time, you'll be able to find a predefined `Collector` that will do what you need by looking through `java.util.stream.Collectors`. In the rare situation when you don't, you can use the second form of `collect()`. I'll basically leave that as a more advanced exercise, but here's one example to give the basic idea:

```
// streams/SpecialCollector.java
import java.util.*;
import java.util.stream.*;

public class SpecialCollector {
  public static void
  main(String[] args) throws Exception {
    ArrayList<String> words =
      FileToWords.stream("Cheese.dat")
        .collect(ArrayList::new,
                 ArrayList::add,
                 ArrayList::addAll);
    words.stream()
      .filter(s -> s.equals("cheese"))
      .forEach(System.out::println);
  }
}
/* Output:
cheese
cheese
*/
```

Here, the `ArrayList` methods already do what you need but it seems more likely that if you must use this form of `collect()` you'll have to create special definitions.

Combining All Stream Elements

- `reduce(BinaryOperator)`: Uses `BinaryOperator` to combine all stream elements. Returns an `Optional` because the stream might be empty.
- `reduce(identity, BinaryOperator)`: As above, but using `identity` as the initial value for the combination. Thus, if the stream is empty, you still get `identity` as the result.
- `reduce(identity, BiFunction, BinaryOperator)`: This is more complicated (so we won't cover it), but is included because it can be more efficient. You can usually express this more simply by combining explicit `map()` and `reduce()` operations.

Here's a contrived example to demonstrate `reduce()`:

```
// streams/Reduce.java
import java.util.*;
import java.util.stream.*;

class Frobnitz {
  int size;
  Frobnitz(int sz) { size = sz; }
  @Override public String toString() {
    return "Frobnitz(" + size + ")";
  }
  // Generator:
  static Random rand = new Random(47);
  static final int BOUND = 100;
  static Frobnitz supply() {
    return new Frobnitz(rand.nextInt(BOUND));
  }
}

public class Reduce {
  public static void main(String[] args) {
    Stream.generate(Frobnitz::supply)
      .limit(10)
      .peek(System.out::println)
      .reduce((fr0, fr1) -> fr0.size < 50 ? fr0 : fr1)
      .ifPresent(System.out::println);
  }
}
/* Output:
Frobnitz(58)
Frobnitz(55)
Frobnitz(93)
Frobnitz(61)
Frobnitz(61)
Frobnitz(29)
Frobnitz(68)
Frobnitz(0)
Frobnitz(22)
Frobnitz(7)
Frobnitz(29)
*/
```

`Frobnitz` contains its own generator named `supply()`; we can pass a method

reference to `Stream.generate()` because it is signature-compatible as a `Supplier<Frobnitz>` (This signature compatibility is called *structural conformance*). We use the `reduce()` method without the first-argument "starter value", which means it produces an `Optional`. The `Optional.ifPresent()` method calls a `Consumer<Frobnitz>` (`println` conforms because it can take a `String` by converting the `Frobnitz` via `toString()`) only if the result is non-`empty`.

The first argument in the lambda expression, `fr0`, is the result that is carried over from the last invocation of this `reduce()`, and the second argument, `fr1`, is the new value that's coming in from the stream.

The `reduce()` lambda uses a ternary if-else to take `fr0` if its `size` is less than 50, otherwise it takes `fr1`, the next one in the sequence. As a result, you get the *first* `Frobnitz` with a `size` less than 50—it just keeps holding onto that one once you've found it, even though other candidates appear. Although this is a rather odd constraint, it does give you a bit more insight into `reduce()`.

Matching

- `allMatch(Predicate)`: Returns `true` if *every* element of the stream produces `true` when provided to the supplied `Predicate`. This will short-circuit upon the first `false`; it won't continue the calculation once it finds one `false`.
- `anyMatch(Predicate)`: Returns `true` if *any* element of the stream produces `true` when provided to the supplied `Predicate`. This will short-circuit upon the first `true`.
- `noneMatch(Predicate)`: Returns `true` if *no* elements of the stream produce `true` when provided to the supplied `Predicate`. This will short-circuit upon the first `true`.

You've seen an example of `noneMatch()` in `Prime.java`; the usage of `allMatch()` and `anyMatch()` are virtually identical.

Let's explore the short-circuiting behavior. To create a `show()` method that eliminates repeated code, we must first discover how to generically describe all three of the matcher operations, which we then turn into an interface called `Matcher`:

```
// streams/Matching.java
// Demonstrates short-circuiting of *Match() operations
import java.util.stream.*;
import java.util.function.*;
import static streams.RandInts.*;

interface Matcher extends
  BiPredicate<Stream<Integer>, Predicate<Integer>> {}

public class Matching {
  static void show(Matcher match, int val) {
    System.out.println(
      match.test(
        IntStream.rangeClosed(1, 9)
          .boxed()
          .peek(n -> System.out.format("%d ", n)),
        n -> n < val));
  }
  public static void main(String[] args) {
    show(Stream::allMatch, 10);
    show(Stream::allMatch, 4);
    show(Stream::anyMatch, 2);
    show(Stream::anyMatch, 0);
    show(Stream::noneMatch, 5);
    show(Stream::noneMatch, 0);
  }
}
/* Output:
1 2 3 4 5 6 7 8 9 true
1 2 3 4 false
1 true
1 2 3 4 5 6 7 8 9 false
1 false
1 2 3 4 5 6 7 8 9 true
*/
```

`BiPredicate` is a binary predicate, which only means it takes two arguments and returns `true` or `false`. The first argument is the stream of numbers we are going to test, and the second argument is the `Predicate` itself. Because `Matcher` fits the pattern of all the `Stream::*Match` functions, we can pass each one to `show()`. The call to `match.test()` is translated into an invocation of the `Stream::*Match`

function.

`show()` takes a `Matcher` and a `val` indicating the maximum number in the predicate's test `n < val`. It generates a stream of `Integers` from 1 through 9. The `peek()` is to show us how far the test gets before it short-circuits. You can see from the output that the short-circuiting happens every time.

Selecting an Element

- `findFirst()`: returns an `Optional` containing the first element of the stream, or `Optional.empty` if the stream has no elements.
- `findAny()`: returns an `Optional` containing some element of the stream, or `Optional.empty` if the stream has no elements.

```
// streams/SelectElement.java
import java.util.*;
import java.util.stream.*;
import static streams.RandInts.*;

public class SelectElement {
  public static void main(String[] args) {
    System.out.println(rands().findFirst().getAsInt());
    System.out.println(
      rands().parallel().findFirst().getAsInt());
    System.out.println(rands().findAny().getAsInt());
    System.out.println(
      rands().parallel().findAny().getAsInt());
  }
}
/* Output:
258
258
258
242
*/
```

`findFirst()` will always select the first element in the stream, whether or not the stream is `parallel()`. For a non-`parallel()` stream, `findAny()` chooses the first element (although from the definition it has the option to choose any element). In this

example, making the stream `parallel()` introduces the possibility that `findAny()` chooses other than the first element.

If you must select the last element in a stream, use `reduce()`:

```
// streams/LastElement.java
import java.util.*;
import java.util.stream.*;

public class LastElement {
  public static void main(String[] args) {
    OptionalInt last = IntStream.range(10, 20)
      .reduce((n1, n2) -> n2);
    System.out.println(last.orElse(-1));
    // Non-numeric object:
    Optional<String> lastobj =
      Stream.of("one", "two", "three")
        .reduce((n1, n2) -> n2);
    System.out.println(
      lastobj.orElse("Nothing there!"));
  }
}
/* Output:
19
three
*/
```

The argument to `reduce()` just replaces the last two elements with the last element, ultimately producing only the last element. If the stream is numeric, you must use the appropriate numeric optional type, otherwise you use a typed `Optional` as in `Optional<String>`.

Informational

- `count()`: The number of elements in this stream.
- `max(Comparator)`: The "maximum" element of this stream as determined by the `Comparator`.
- `min(Comparator)`: The "minimum" element of this stream as determined by the `Comparator`.

`Strings` have a predefined `Comparator`, which simplifies our example:

```
// streams/Informational.java
import java.util.stream.*;
import java.util.function.*;

public class Informational {
  public static void
  main(String[] args) throws Exception {
      System.out.println(
        FileToWords.stream("Cheese.dat").count());
      System.out.println(
        FileToWords.stream("Cheese.dat")
        .min(String.CASE_INSENSITIVE_ORDER)
        .orElse("NONE"));
      System.out.println(
        FileToWords.stream("Cheese.dat")
        .max(String.CASE_INSENSITIVE_ORDER)
        .orElse("NONE"));
  }
}
/* Output:
32
a
you
*/
```

`min()` and `max()` return `Optionals`, which I unpack using `orElse()`;

Information for Numeric Streams

- `average()`: The usual meaning.
- `max()` & `min()`: These don't need a `Comparator` because they work on numeric streams.
- `sum()`: Add up the values in the stream.
- `summaryStatistics()`: Produces potentially useful data. It's not quite clear why they felt the need for this one because you can produce all the data yourself with the direct methods.

```
// streams/NumericStreamInfo.java
import java.util.stream.*;
import static streams.RandInts.*;

public class NumericStreamInfo {
  public static void main(String[] args) {
    System.out.println(rands().average().getAsDouble());
    System.out.println(rands().max().getAsInt());
    System.out.println(rands().min().getAsInt());
    System.out.println(rands().sum());
    System.out.println(rands().summaryStatistics());
  }
}
/* Output:
507.94
998
8
50794
IntSummaryStatistics{count=100, sum=50794, min=8,
average=507.940000, max=998}
*/
```

The same operations are available for `LongStreams` and `DoubleStreams`.

Summary

Streams change—and greatly improve—the nature of Java programming, and may stem the flow of Java programmers to functional JVM languages like Kotlin and Scala. I use streams whenever possible throughout the rest of this book.

Exceptions

> The basic philosophy of Java is that "badly formed code will not run."

(At least, that's what I infer.)

Improved error recovery is one of the most powerful ways you can increase the robustness of your code. Error recovery is a fundamental concern for every program you write, but it's especially important in Java, a primary goal is to create program components for others to use.

The ideal time to catch an error is at compile time, before you even try to run the program. However, not all errors can be detected at compile time. Other problems must be dealt with at run time through some formality that allows the originator of the error to pass appropriate information to a recipient who will know how to handle the difficulty properly.

> To create a robust system, each component must be robust.

By providing a consistent error-reporting model using exceptions, Java allows components to reliably communicate problems to client code.

The goals for exception handling in Java are to simplify the creation of large, reliable programs using less code than currently possible, and to do so with more confidence that your application doesn't have an unhandled error. Exceptions are not terribly difficult to learn, and can provide immediate and significant benefits to your project.

Because exception handling is the only official way that Java reports errors, and is enforced by the Java compiler, you can only go so far without learning about exception handling. This chapter teaches you to write code to properly handle exceptions, and to generate your own exceptions if one of your methods gets into trouble.

Concepts

C and other earlier languages often had multiple error-handling schemes, and these were generally established by convention and not as part of the programming language. Typically, you returned a special value or set a flag, and the recipient determined that something was amiss by looking at the value or flag. Over time, we realized that programmers who use a library tend to think of themselves as invincible—as in "Yes, errors might happen to others, but not in *my* code." So, not too surprisingly, they wouldn't check for the error conditions (and sometimes the error conditions were too silly to check for[99]). If you *did* thoroughly check for errors every time you called a method, your code turned into an unreadable nightmare. Because programmers could still coax systems out of these languages, they resisted the truth: that this approach to handling errors is a major limitation to creating large, robust, maintainable programs.

One solution is to take the casual nature out of error handling and to enforce formality. This actually has a long history, because implementations of *exception handling* go back to operating systems in the 1960s, and even to BASIC's "`on error goto`." But C++ exception handling was based on Ada, and Java's is based primarily on C++ (although it looks more like Object Pascal).

The word "exception" is meant in the sense of "I take exception to that." When the problem occurs, you might not know what to do with it, but you do know you can't just continue on merrily; you must stop, and somebody, somewhere, must figure out what to do. But you don't have enough information in the current context to fix the problem. So you hand the problem to a higher context where someone is qualified to make the proper decision.

Exceptions can reduce some of the complexity of error-handling code. Without exceptions, you must check for a particular error and deal with it, possibly at multiple places in your program. With exceptions, you no longer check for errors at the point of the method call because the exception will guarantee that someone catches it. Ideally, you only handle the problem in one place, in the so-called *exception handler*. This can save code, and separate the code that describes your goal during normal execution from the code executed when things go awry. Used carefully, reading,

[99] The C programmer can look up the return value of `printf()` for an example of this.

writing, and debugging code can be clearer with exceptions than when using the old ways of error handling.

Basic Exceptions

An *exceptional condition* is a problem that prevents the continuation of the current method or scope. It's important to distinguish an exceptional condition from a normal problem, when you have enough information in the current context to somehow cope with the difficulty. With an exceptional condition, you cannot continue processing because you don't have the information necessary to deal with the problem *in the current context*. All you can do is jump out of the current context and relegate that problem to a higher context. This is what happens when you throw an exception.

Division is a simple example. If you're about to divide by zero, it's worth checking for that condition. But what does it mean that the denominator is zero? Maybe you know, in the context of the problem you're trying to solve in that particular method, how to deal with a zero denominator. But if it's an unexpected value, you can't deal with it and so must throw an exception rather than continuing along that execution path.

When you throw an exception, several things happen. First, the exception object is created in the same way as any Java object: on the heap, with `new`. Then the current path of execution (the one you can't continue) is stopped and the reference for the exception object is ejected from the current context. Now the exception-handling mechanism takes over and begins to look for an appropriate place to continue executing the program. This appropriate place is the *exception handler*, whose job is to recover from the problem so the program can either try another tack or just continue.

As a simple example of throwing an exception, consider an object reference called `t`. You might receive a reference that hasn't been initialized, so you check before using that reference. You can send information about the error into a larger context by creating an object representing your information and "throwing" it out of your current context. This is called *throwing an exception*. Here's what it looks like:

```
if(t == null)
  throw new NullPointerException();
```

This throws the exception, which allows you—in the current context—to abdicate responsibility for the issue. It's just magically handled somewhere else. Precisely *where* is shown shortly.

Exceptions allow you to think of everything you do as a transaction, and the exceptions guard those transactions: "...the fundamental premise of transactions is that we needed exception handling in distributed computations. Transactions are the computer equivalent of contract law. If anything goes wrong, we'll just blow away the whole computation."[100] You can also think about exceptions as a built-in "undo" system, because (with some care) you can have various recovery points in your program. If a part of the program fails, the exception will "undo" back to a known stable point in the program.

If something bad happens, exceptions don't allow a program to continue along its ordinary path. This is a real problem in languages like C and C++; especially C, which had no way to force a program to stop going down a path if a problem occurred, so it was possible to ignore problems for a long time and get into a completely inappropriate state. Exceptions allow you to (if nothing else) force the program to stop and tell you what went wrong, or (ideally) force the program to deal with the problem and return to a stable state.

Exception Arguments

As with any object in Java, you always create exceptions on the heap using `new`, which allocates storage and calls a constructor. There are two constructors in all standard exceptions: The first is the zero-argument constructor, and the second takes a `String` argument to place pertinent information in the exception:

```
throw new NullPointerException("t = null");
```

This `String` can later be extracted using various methods, as you'll see.

The keyword `throw` produces several interesting consequences. After creating an exception object with `new`, you give the resulting reference to `throw`. The object is,

[100] Jim Gray, Turing Award winner for his team's contributions on transactions, in an interview on *www.acmqueue.org*.

in effect, "returned" from the method, even though that object type isn't normally what the method is designed to return. A simplistic way to think about exception handling is as a different kind of return mechanism, although you'll have trouble if you take that analogy too far. You can also exit from ordinary scopes by throwing an exception. In either case, an exception object is returned, and the method or scope exits.

Any similarity to an ordinary return from a method ends here, because *where* you return to is someplace completely different from where you return for a normal method call. You end up in an appropriate exception handler that might be far away—many levels on the call stack—from where the exception was thrown.

In addition, you can throw any type of `Throwable`, the exception root class. Typically, you'll throw a different class of exception for each different type of error. The information about the error is represented both inside the exception object and implicitly in the name of the exception class, so someone in the bigger context can figure out what to do with your exception. (Often, the only information is the type name of the exception, and nothing meaningful is stored within the exception object itself.)

Catching an Exception

To see how an exception is caught, you must first understand the concept of a *guarded region*. This is a section of code that might produce exceptions, and is followed by the code to handle those exceptions.

The `try` Block

If you're inside a method and you throw an exception (or another method you call within this method throws an exception), that method will exit in the process of throwing. If you don't want a `throw` to exit the method, you can set up a special block within that method to capture the exception. This is called the *try block* because you "try" your various method calls there. The try block is an ordinary scope preceded by the keyword `try`:

```
try {
  // Code that might generate exceptions
}
```

If you are carefully checking for errors in a programming language that doesn't support exception handling, you surround every method call with setup and error-testing code, even if you call the same method several times. With exception handling, you put everything in a `try` block and capture all the exceptions in one place. This means your code can be much easier to write and read, because the goal of the code is not confused with the error checking.

Exception Handlers

The thrown exception ends up someplace. This "place" is the exception handler, and you write one for every exception type. Exception handlers immediately follow the `try` block and are denoted by the keyword `catch`:

```
try {
  // Code that might generate exceptions
} catch(Type1 id1) {
  // Handle exceptions of Type1
} catch(Type2 id2) {
  // Handle exceptions of Type2
} catch(Type3 id3) {
  // Handle exceptions of Type3
}

// etc.
```

Each `catch` clause (exception handler) is like a little method that takes one and only one argument of a particular type. The identifier (`id1`, `id2`, and so on) can be used inside the handler, just like a method argument. Sometimes you never use the identifier because the type of the exception gives you enough information to deal with the exception, but the identifier must still be there.

The handlers must appear directly after the `try` block. If an exception is thrown, the exception-handling mechanism goes hunting for the first handler with an argument that matches the type of the exception. Then it enters that `catch` clause, and the exception is considered handled. The search for handlers stops once the `catch`

clause is finished. Only the matching `catch` clause executes; it's not like a `switch` statement when you need a `break` after each `case` to prevent the remaining ones from executing.

Note that within the `try` block, a number of different method calls might generate the same exception, but you need only one handler.

Termination vs. Resumption

There are two basic models in exception-handling theory. Java supports *termination*,[101] where you assume that the error is so critical there's no way to get back to where the exception occurred. Whoever threw the exception decided there was no way to salvage the situation, and they don't *want* to come back.

The alternative is called *resumption*. It means the exception handler is expected to do something to rectify the situation, then the faulting method is retried, presuming success the second time. If you want resumption, it means you still hope to continue execution after the exception is handled.

If you want resumption-like behavior in Java, don't throw an exception when you encounter an error. Instead, call a method that fixes the problem. Alternatively, place your `try` block inside a `while` loop that keeps reentering the `try` block until the result is satisfactory.

Historically, programmers using operating systems that supported resumptive exception handling eventually ended up using termination-like code and skipping resumption. So although resumption sounds attractive at first, it isn't so useful in practice. The dominant reason is probably the coupling that results: A resumptive handler would be aware of where the exception is thrown, and contain non-generic code specific to the throwing location. This makes the code difficult to write and maintain, especially for large systems where the exception can be generated from many points.

Creating Your Own Exceptions

You're not stuck using the existing Java exceptions. The Java exception hierarchy can't foresee all the errors you might report, so you can create your own to denote a

[101] As do most languages, including C++, C#, Python, D, etc.

special problem that your library might encounter.

To create your own exception class, inherit from an existing exception class, preferably one that is close in meaning to your new exception (although this is often not possible). The most trivial way to create a new type of exception is just to let the compiler create the zero-argument constructor for you, so it requires almost no code at all:

```
// exceptions/InheritingExceptions.java
// Creating your own exceptions

class SimpleException extends Exception {}

public class InheritingExceptions {
  public void f() throws SimpleException {
    System.out.println(
      "Throw SimpleException from f()");
    throw new SimpleException();
  }
  public static void main(String[] args) {
    InheritingExceptions sed =
      new InheritingExceptions();
    try {
      sed.f();
    } catch(SimpleException e) {
      System.out.println("Caught it!");
    }
  }
}
/* Output:
Throw SimpleException from f()
Caught it!
*/
```

The compiler creates a zero-argument constructor, which automatically (and invisibly) calls the base-class zero-argument constructor. Here you don't get a `SimpleException(String)` constructor, but in practice that isn't used much. As you'll see, the most important thing about an exception is the class name, so most of the time an exception like the one shown here is satisfactory.

Here, the result is displayed on the console. You can also send error output to the *standard error* stream by writing to `System.err`. This is usually a better place to

send error information than `System.out`, which can be redirected. If you send output to `System.err`, it is not redirected along with `System.out` so the user is more likely to notice it.

You can also create an exception class that has a constructor with a `String` argument:

```
// exceptions/FullConstructors.java

class MyException extends Exception {
  MyException() {}
  MyException(String msg) { super(msg); }
}

public class FullConstructors {
  public static void f() throws MyException {
    System.out.println("Throwing MyException from f()");
    throw new MyException();
  }
  public static void g() throws MyException {
    System.out.println("Throwing MyException from g()");
    throw new MyException("Originated in g()");
  }
  public static void main(String[] args) {
    try {
      f();
    } catch(MyException e) {
      e.printStackTrace(System.out);
    }
    try {
      g();
    } catch(MyException e) {
      e.printStackTrace(System.out);
    }
  }
}
/* Output:
Throwing MyException from f()
MyException
        at FullConstructors.f(FullConstructors.java:11)
        at
FullConstructors.main(FullConstructors.java:19)
Throwing MyException from g()
```

```
MyException: Originated in g()
        at FullConstructors.g(FullConstructors.java:15)
        at
FullConstructors.main(FullConstructors.java:24)
*/
```

The added code is small: two constructors that define the way you create `MyException`. In the second constructor, the base-class constructor with a `String` argument is explicitly invoked using the `super` keyword.

In the handlers, one of the `Throwable` (from which `Exception` is inherited) methods is called: `printStackTrace()`. As shown in the output, this produces information about the sequence of method calls to get to the point where the exception happened. Here, the information is sent to `System.out`, and automatically captured and displayed in the output. However, if you call the default version:

```
e.printStackTrace();
```

the information goes to the standard error stream.

Exceptions and Logging

You might also *log* the output using the `java.util.logging` facility. Basic logging is straightforward enough to show here.

```
// exceptions/LoggingExceptions.java
// An exception that reports through a Logger
// {ErrorOutputExpected}
import java.util.logging.*;
import java.io.*;

class LoggingException extends Exception {
  private static Logger logger =
    Logger.getLogger("LoggingException");
  LoggingException() {
    StringWriter trace = new StringWriter();
    printStackTrace(new PrintWriter(trace));
    logger.severe(trace.toString());
  }
}
```

```
public class LoggingExceptions {
  public static void main(String[] args) {
    try {
      throw new LoggingException();
    } catch(LoggingException e) {
      System.err.println("Caught " + e);
    }
    try {
      throw new LoggingException();
    } catch(LoggingException e) {
      System.err.println("Caught " + e);
    }
  }
}
/* Output:
___[ Error Output ]___
Jan 24, 2021 8:48:54 AM LoggingException <init>
SEVERE: LoggingException
        at
LoggingExceptions.main(LoggingExceptions.java:20)

Caught LoggingException
Jan 24, 2021 8:48:54 AM LoggingException <init>
SEVERE: LoggingException
        at
LoggingExceptions.main(LoggingExceptions.java:25)

Caught LoggingException
*/
```

The `static Logger.getLogger()` method creates a `Logger` object associated with the `String` argument (usually the name of the package and class that the errors are about) which sends its output to `System.err`. The easiest way to write to a `Logger` is just to call the method associated with the level of logging message; here, `severe()` is used. To produce the `String` for the logging message, we'd like the stack trace to appear where the exception is thrown, but `printStackTrace()` doesn't produce a `String` by default. To get a `String`, we use the overloaded `printStackTrace()` that takes a `java.io.PrintWriter` object as an argument (`PrintWriter` is fully explained in the Appendix: I/O Streams). If we hand the `PrintWriter` constructor a `java.io.StringWriter` object, the output can be extracted as a `String` by calling

`toString()`.

The approach used by `LoggingException` is convenient because it builds all the logging infrastructure into the exception itself, and thus it works automatically without client programmer intervention. However, it's more common to catch and log someone else's exception, so you must generate the log message in the exception handler:

```
// exceptions/LoggingExceptions2.java
// Logging caught exceptions
// {ErrorOutputExpected}
import java.util.logging.*;
import java.io.*;

public class LoggingExceptions2 {
  private static Logger logger =
    Logger.getLogger("LoggingExceptions2");
  static void logException(Exception e) {
    StringWriter trace = new StringWriter();
    e.printStackTrace(new PrintWriter(trace));
    logger.severe(trace.toString());
  }
  public static void main(String[] args) {
    try {
      throw new NullPointerException();
    } catch(NullPointerException e) {
      logException(e);
    }
  }
}
/* Output:
___[ Error Output ]___
Jan 24, 2021 8:48:54 AM LoggingExceptions2 logException
SEVERE: java.lang.NullPointerException
        at
LoggingExceptions2.main(LoggingExceptions2.java:17)
*/
```

The process of creating your own exceptions can be taken further. You can add extra constructors and members:

```
// exceptions/ExtraFeatures.java
// Further embellishment of exception classes

class MyException2 extends Exception {
  private int x;
  MyException2() {}
  MyException2(String msg) { super(msg); }
  MyException2(String msg, int x) {
    super(msg);
    this.x = x;
  }
  public int val() { return x; }
  @Override public String getMessage() {
    return "Detail Message: "+ x
      + " "+ super.getMessage();
  }
}

public class ExtraFeatures {
  public static void f() throws MyException2 {
    System.out.println(
      "Throwing MyException2 from f()");
    throw new MyException2();
  }
  public static void g() throws MyException2 {
    System.out.println(
      "Throwing MyException2 from g()");
    throw new MyException2("Originated in g()");
  }
  public static void h() throws MyException2 {
    System.out.println(
      "Throwing MyException2 from h()");
    throw new MyException2("Originated in h()", 47);
  }
  public static void main(String[] args) {
    try {
      f();
    } catch(MyException2 e) {
      e.printStackTrace(System.out);
    }
    try {
      g();
```

```
    } catch(MyException2 e) {
      e.printStackTrace(System.out);
    }
    try {
      h();
    } catch(MyException2 e) {
      e.printStackTrace(System.out);
      System.out.println("e.val() = " + e.val());
    }
  }
}
/* Output:
Throwing MyException2 from f()
MyException2: Detail Message: 0 null
        at ExtraFeatures.f(ExtraFeatures.java:23)
        at ExtraFeatures.main(ExtraFeatures.java:37)
Throwing MyException2 from g()
MyException2: Detail Message: 0 Originated in g()
        at ExtraFeatures.g(ExtraFeatures.java:28)
        at ExtraFeatures.main(ExtraFeatures.java:42)
Throwing MyException2 from h()
MyException2: Detail Message: 47 Originated in h()
        at ExtraFeatures.h(ExtraFeatures.java:33)
        at ExtraFeatures.main(ExtraFeatures.java:47)
e.val() = 47
*/
```

A field `x` is added, along with a method to reads that value and an additional constructor that sets it. In addition, `Throwable.getMessage()` is overridden to produce a more helpful detail message. `getMessage()` is something like `toString()` for exception classes.

An exception is just another kind of object so you can continue this process of embellishing the power of your exception classes. Keep in mind, however, that all this dressing-up might be lost on the client programmers using your packages because they might simply look for the exception to be thrown and nothing more. (That's the way most of the Java library exceptions are used.)

The Exception Specification

In Java, you're encouraged to inform the client programmer, who calls your method, of the exceptions that might be thrown from your method. This is civilized, because the caller can then know exactly what code to write to catch all potential exceptions. If the source code is available, the client programmer can hunt through and look for `throw` statements, but a library might not come with sources. To prevent this from being a problem, Java requires syntax that politely tells the client programmer what exceptions this method throws, so the client programmer can handle them. This is the *exception specification* and it's part of the method declaration, appearing after the argument list.

The exception specification uses an additional keyword, `throws`, followed by a list of all the potential exception types. So your method definition might look like this:

```
void f() throws TooBig, TooSmall, DivZero { // ...
```

However, if you say

```
void f() { // ...
```

it means no exceptions are thrown from the method (*except* for the exceptions inherited from `RuntimeException`, which can be thrown anywhere without exception specifications—these are described later).

You can't lie about an exception specification. If the code within your method causes exceptions, but your method doesn't handle them, the compiler will detect this and declare you must either handle the exception or indicate with an exception specification that it might be thrown from your method. By enforcing exception specifications from top to bottom, Java guarantees that a certain level of exception correctness can be ensured at compile time.

There is one place you can lie: You can claim to throw an exception you really don't. The compiler takes your word for it, and forces the users of your method to treat it as if it really does throw that exception. This has the beneficial effect of being a placeholder for that exception, so you can actually start throwing the exception later without requiring changes to existing code. It's also important for creating `abstract` base classes and interfaces whose derived classes or implementations might throw exceptions.

Exceptions that are checked and enforced at compile time are called *checked exceptions*.

Catching Any Exception

You can create a handler that catches any type of exception by catching the base-class exception type `Exception`. There are other types of base exceptions, but `Exception` is the base that's pertinent to virtually all programming activities:

```
catch(Exception e) {
  System.out.println("Caught an exception");
}
```

This will catch any exception, so if you use it, put it at the *end* of your list of handlers to avoid preempting any exception handlers that might otherwise follow.

The `Exception` class is the base of all the exception classes important to the programmer so you don't get much specific information about the exception, but you can call the methods that come from *its* base type `Throwable`:

```
String getMessage()
String getLocalizedMessage()
```

: Gets the detail message, or a message adjusted for this particular locale.

`String toString()`
: Returns a short description of the `Throwable`, including the detail message if there is one.

```
void printStackTrace()
void printStackTrace(PrintStream)
void printStackTrace(java.io.PrintWriter)
```

: Prints the `Throwable` and the `Throwable` call stack trace. The call stack shows the sequence of method calls that brought you to the point at which the exception was thrown. The first version prints to standard error, the second and third print to a stream of your choice (in the Appendix: I/O Streams, you'll understand why there are two types of streams).

`Throwable fillInStackTrace()`
: Records information within this `Throwable` object about the current state of the stack frames. Useful when an application is rethrowing an error or exception (more about this shortly).

In addition, you get some other methods from `Throwable`'s base type `Object` (everybody's base type). The one that might come in handy for exceptions is `getClass()`, which returns an object representing the class of this object. You can in turn query this `Class` object for its name with `getName()`, which includes package information, or `getSimpleName()`, which produces the class name alone.

Here's an example that shows the basic `Exception` methods:

```
// exceptions/ExceptionMethods.java
// Demonstrating the Exception Methods

public class ExceptionMethods {
  public static void main(String[] args) {
    try {
      throw new Exception("My Exception");
    } catch(Exception e) {
      System.out.println("Caught Exception");
      System.out.println(
        "getMessage():" + e.getMessage());
      System.out.println("getLocalizedMessage():" +
        e.getLocalizedMessage());
      System.out.println("toString():" + e);
      System.out.println("printStackTrace():");
      e.printStackTrace(System.out);
    }
  }
}
/* Output:
Caught Exception
getMessage():My Exception
getLocalizedMessage():My Exception
toString():java.lang.Exception: My Exception
printStackTrace():
java.lang.Exception: My Exception
        at
ExceptionMethods.main(ExceptionMethods.java:7)
*/
```

The methods provide successively more information—each is effectively a superset of the previous one.

Multi-Catch

If there are a set of exceptions you want to handle the same way and they have a common base type, you just catch that. But if those exceptions don't have a base type in common, before Java 7 you must write a catch for each one:

```
// exceptions/SameHandler.java

class EBase1 extends Exception {}
class Except1 extends EBase1 {}
class EBase2 extends Exception {}
class Except2 extends EBase2 {}
class EBase3 extends Exception {}
class Except3 extends EBase3 {}
class EBase4 extends Exception {}
class Except4 extends EBase4 {}

public class SameHandler {
  void x() throws Except1, Except2, Except3, Except4 {}
  void process() {}
  void f() {
    try {
      x();
    } catch(Except1 e) {
      process();
    } catch(Except2 e) {
      process();
    } catch(Except3 e) {
      process();
    } catch(Except4 e) {
      process();
    }
  }
}
```

With Java 7 multi-catch handlers, you can "OR" together different types of exceptions in a single `catch`:

```
// exceptions/MultiCatch.java

public class MultiCatch {
  void x() throws Except1, Except2, Except3, Except4 {}
  void process() {}
  void f() {
    try {
      x();
    } catch(Except1 | Except2 | Except3 | Except4 e) {
      process();
    }
  }
}
```

Or in other combinations:

```
// exceptions/MultiCatch2.java

public class MultiCatch2 {
  void x() throws Except1, Except2, Except3, Except4 {}
  void process1() {}
  void process2() {}
  void f() {
    try {
      x();
    } catch(Except1 | Except2 e) {
      process1();
    } catch(Except3 | Except4 e) {
      process2();
    }
  }
}
```

This is a nice contribution to clearer code.

The Stack Trace

The information provided by `printStackTrace()` can also be accessed directly using `getStackTrace()`. This method returns an array of stack trace elements, each representing one stack frame. Element zero is the top of the stack, and is the

last method invocation in the sequence (the point this `Throwable` was created and thrown). The last element of the array and the bottom of the stack is the first method invocation in the sequence. This program provides a simple demonstration:

```
// exceptions/WhoCalled.java
// Programmatic access to stack trace information

public class WhoCalled {
  static void f() {
    // Generate an exception to fill in the stack trace
    try {
      throw new Exception();
    } catch(Exception e) {
      for(StackTraceElement ste : e.getStackTrace())
        System.out.println(ste.getMethodName());
    }
  }
  static void g() { f(); }
  static void h() { g(); }
  public static void main(String[] args) {
    f();
    System.out.println("*******");
    g();
    System.out.println("*******");
    h();
  }
}
/* Output:
f
main
*******
f
g
main
*******
f
g
h
main
*/
```

Here, we just print the method name, but you can also print the entire `Stack-`

`TraceElement`, which contains additional information.

Rethrowing an Exception

Sometimes you'll rethrow the exception you just caught, particularly when you use `Exception` to catch any exception. You already have the reference to the current exception so you can rethrow that reference:

```
catch(Exception e) {
  System.out.println("An exception was thrown");
  throw e;
}
```

Rethrowing an exception causes it to go to the exception handlers in the next-higher context. Any further `catch` clauses for the same `try` block are still ignored. In addition, everything about the exception object is preserved, so the handler at the higher context that catches the specific exception type can extract all the information from that object.

If you rethrow the current exception, the information you print about that exception in `printStackTrace()` will pertain to the exception's origin, not the place where you rethrow it. To install new stack trace information, you can call `fillInStackTrace()`, which returns a `Throwable` object it creates by stuffing the current stack information into the old exception object. Here's what it looks like:

```
// exceptions/Rethrowing.java
// Demonstrating fillInStackTrace()

public class Rethrowing {
  public static void f() throws Exception {
    System.out.println(
      "originating the exception in f()");
    throw new Exception("thrown from f()");
  }
  public static void g() throws Exception {
    try {
      f();
    } catch(Exception e) {
      System.out.println(
        "Inside g(), e.printStackTrace()");
```

```
      e.printStackTrace(System.out);
      throw e;
    }
  }
  public static void h() throws Exception {
    try {
      f();
    } catch(Exception e) {
      System.out.println(
        "Inside h(), e.printStackTrace()");
      e.printStackTrace(System.out);
      throw (Exception)e.fillInStackTrace();
    }
  }
  public static void main(String[] args) {
    try {
      g();
    } catch(Exception e) {
      System.out.println("main: printStackTrace()");
      e.printStackTrace(System.out);
    }
    try {
      h();
    } catch(Exception e) {
      System.out.println("main: printStackTrace()");
      e.printStackTrace(System.out);
    }
  }
}
/* Output:
originating the exception in f()
Inside g(), e.printStackTrace()
java.lang.Exception: thrown from f()
        at Rethrowing.f(Rethrowing.java:8)
        at Rethrowing.g(Rethrowing.java:12)
        at Rethrowing.main(Rethrowing.java:32)
main: printStackTrace()
java.lang.Exception: thrown from f()
        at Rethrowing.f(Rethrowing.java:8)
        at Rethrowing.g(Rethrowing.java:12)
        at Rethrowing.main(Rethrowing.java:32)
originating the exception in f()
```

```
Inside h(), e.printStackTrace()
java.lang.Exception: thrown from f()
        at Rethrowing.f(Rethrowing.java:8)
        at Rethrowing.h(Rethrowing.java:22)
        at Rethrowing.main(Rethrowing.java:38)
main: printStackTrace()
java.lang.Exception: thrown from f()
        at Rethrowing.h(Rethrowing.java:27)
        at Rethrowing.main(Rethrowing.java:38)
*/
```

The line where `fillInStackTrace()` is called becomes the new point of origin of the exception.

It's also possible to rethrow a different exception from the one you caught. If you do this, you get a similar effect as when you use `fillInStackTrace()`—the information about the original site of the exception is lost, and what you're left with is the information pertaining to the new `throw`:

```
// exceptions/RethrowNew.java
// Rethrow a different object from the one you caught

class OneException extends Exception {
  OneException(String s) { super(s); }
}

class TwoException extends Exception {
  TwoException(String s) { super(s); }
}

public class RethrowNew {
  public static void f() throws OneException {
    System.out.println(
      "originating the exception in f()");
    throw new OneException("thrown from f()");
  }
  public static void main(String[] args) {
    try {
      try {
        f();
      } catch(OneException e) {
        System.out.println(
```

```
          "Caught in inner try, e.printStackTrace()");
        e.printStackTrace(System.out);
        throw new TwoException("from inner try");
      }
    } catch(TwoException e) {
      System.out.println(
        "Caught in outer try, e.printStackTrace()");
      e.printStackTrace(System.out);
    }
  }
}
/* Output:
originating the exception in f()
Caught in inner try, e.printStackTrace()
OneException: thrown from f()
        at RethrowNew.f(RethrowNew.java:16)
        at RethrowNew.main(RethrowNew.java:21)
Caught in outer try, e.printStackTrace()
TwoException: from inner try
        at RethrowNew.main(RethrowNew.java:26)
*/
```

The final exception knows only it came from the inner `try` block and not from `f()`.

Don't worry about cleaning up the previous exception, or any exceptions. They're all heap-based objects created with `new`, so the garbage collector automatically cleans them all up.

Precise Rethrow

Before Java 7, if you caught an exception, you could only rethrow that type of exception. This caused imprecisions in code that were fixed in Java 7. So before, this would not compile:

```
// exceptions/PreciseRethrow.java

class BaseException extends Exception {}
class DerivedException extends BaseException {}

public class PreciseRethrow {
  void catcher() throws DerivedException {
    try {
      throw new DerivedException();
    } catch(BaseException e) {
      throw e;
    }
  }
}
```

Because the `catch` caught a `BaseException`, the compiler forced you to declare that `catcher() throws BaseException`, even though it's actually throwing the more specific `DerivedException`. Since Java 7, this code *does* compile, which is a small but useful fix.

Exception Chaining

Sometimes you catch one exception and throw another, but still keep the information about the originating exception—this is called *exception chaining*. Prior to Java 1.4, programmers wrote their own code to preserve the original exception information, but now all `Throwable` subclasses have the option to take a *cause* object in their constructor. The *cause* is intended as the originating exception, and by passing it in you maintain the stack trace back to its origin, even though you're creating and throwing a new exception.

The only `Throwable` subclasses providing the *cause* argument in the constructor are the three fundamental exception classes `Error` (used by the JVM to report system errors), `Exception`, and `RuntimeException`. To chain any other exception types, use the `initCause()` method rather than the constructor.

Here's an example that dynamically adds fields to a `DynamicFields` object at run time:

```
// exceptions/DynamicFields.java
// A Class that dynamically adds fields to itself to
// demonstrate exception chaining

class DynamicFieldsException extends Exception {}

public class DynamicFields {
  private Object[][] fields;
  public DynamicFields(int initialSize) {
    fields = new Object[initialSize][2];
    for(int i = 0; i < initialSize; i++)
      fields[i] = new Object[] { null, null };
  }
  @Override public String toString() {
    StringBuilder result = new StringBuilder();
    for(Object[] obj : fields) {
      result.append(obj[0]);
      result.append(": ");
      result.append(obj[1]);
      result.append("\n");
    }
    return result.toString();
  }
  private int hasField(String id) {
    for(int i = 0; i < fields.length; i++)
      if(id.equals(fields[i][0]))
        return i;
    return -1;
  }
  private int getFieldNumber(String id)
  throws NoSuchFieldException {
    int fieldNum = hasField(id);
    if(fieldNum == -1)
      throw new NoSuchFieldException();
    return fieldNum;
  }
  private int makeField(String id) {
    for(int i = 0; i < fields.length; i++)
      if(fields[i][0] == null) {
        fields[i][0] = id;
        return i;
      }
```

```
    // No empty fields. Add one:
    Object[][] tmp = new Object[fields.length + 1][2];
    for(int i = 0; i < fields.length; i++)
      tmp[i] = fields[i];
    for(int i = fields.length; i < tmp.length; i++)
      tmp[i] = new Object[] { null, null };
    fields = tmp;
    // Recursive call with expanded fields:
    return makeField(id);
  }
  public Object
  getField(String id) throws NoSuchFieldException {
    return fields[getFieldNumber(id)][1];
  }
  public Object setField(String id, Object value)
  throws DynamicFieldsException {
    if(value == null) {
      // Most exceptions don't have a "cause"
      // constructor. In these cases you must use
      // initCause(), available in all
      // Throwable subclasses.
      DynamicFieldsException dfe =
        new DynamicFieldsException();
      dfe.initCause(new NullPointerException());
      throw dfe;
    }
    int fieldNumber = hasField(id);
    if(fieldNumber == -1)
      fieldNumber = makeField(id);
    Object result = null;
    try {
      result = getField(id); // Get old value
    } catch(NoSuchFieldException e) {
      // Use constructor that takes "cause":
      throw new RuntimeException(e);
    }
    fields[fieldNumber][1] = value;
    return result;
  }
  public static void main(String[] args) {
    DynamicFields df = new DynamicFields(3);
    System.out.println(df);
```

```
    try {
      df.setField("d", "A value for d");
      df.setField("number", 47);
      df.setField("number2", 48);
      System.out.println(df);
      df.setField("d", "A new value for d");
      df.setField("number3", 11);
      System.out.println("df: " + df);
      System.out.println("df.getField(\"d\") : "
        + df.getField("d"));
      Object field =
        df.setField("d", null); // Exception
    } catch(NoSuchFieldException |
            DynamicFieldsException e) {
      e.printStackTrace(System.out);
    }
  }
}
/* Output:
null: null
null: null
null: null

d: A value for d
number: 47
number2: 48

df: d: A new value for d
number: 47
number2: 48
number3: 11

df.getField("d") : A new value for d
DynamicFieldsException
        at
DynamicFields.setField(DynamicFields.java:64)
        at DynamicFields.main(DynamicFields.java:96)
Caused by: java.lang.NullPointerException
        at
DynamicFields.setField(DynamicFields.java:66)
        ... 1 more
*/
```

Each `DynamicFields` object contains an array of `Object-Object` pairs. The first object is the field identifier (a `String`), and the second is the field value, which can be any type except an unwrapped primitive. When you create the object, you make an educated guess about how many fields you need. When you call `setField()`, it either finds the existing field by that name or creates a new one, and puts in your value. If it runs out of space, it adds new space by creating an array of length one longer and copying the old elements in. If you try to put in a `null` value, it throws a `DynamicFieldsException` by creating one and using `initCause()` to insert a `NullPointerException` as the cause.

As a return value, `setField()` also fetches out the old value at that field location using `getField()`, which could throw a `NoSuchFieldException`. If the client programmer calls `getField()`, they are responsible for handling `NoSuchFieldException`, but if this exception is thrown inside `setField()`, it's a programming error, so the `NoSuchFieldException` is converted to a `RuntimeException` using the constructor that takes a *cause* argument.

You'll notice that `toString()` uses a `StringBuilder` to create its result. You'll learn more about `StringBuilder` in the Strings chapter, but in general you'll use it whenever you're writing a `toString()` that involves looping, as is the case here.

The `catch` clause in `main()` looks different—it handles *two* different types of exceptions with the same clause, combined with an "OR" symbol. This Java 7 feature helps reduce code duplication and makes it easier to specify the exact types you are catching, rather than simply catching a base type. You can combine numerous exception types this way.

Standard Java Exceptions

The Java class `Throwable` describes anything that can be thrown as an exception. There are two general types of `Throwable` objects ("types of" = "inherited from"). `Error` represents compile-time and system errors you don't worry about catching (except in very special cases). `Exception` is the basic type that can be thrown from any of the standard Java library class methods and from your methods and runtime accidents. So the Java programmer's base type of interest is usually `Exception`.

The best way to get an overview of the exceptions is to browse the JDK documentation. It's worth doing this once just to get a feel for the various exceptions, but you'll

soon see there isn't anything special between one exception and the next except for the name. Also, the number of exceptions in Java keeps expanding; basically, it's pointless to put them in a book. Any new library you get from a third-party vendor will probably have its own exceptions as well. The important thing to understand is the concept and what to do with the exceptions.

The basic idea is that the name of the exception represents the problem that occurred. The exception name should be relatively self-explanatory. The exceptions are not all defined in `java.lang`; some are created to support other libraries such as `util`, `net`, and `io`, which you see from their full class names or their base classes. For example, all I/O exceptions are inherited from `java.io.IOException`.

Special Case: `RuntimeException`

The first example in this chapter was

```
if(t == null)
  throw new NullPointerException();
```

It can be a bit horrifying to think you must check for `null` on every reference that is passed into a method (you can't know if the caller has passed you a valid reference). Fortunately, you don't—this is part of the standard runtime checking that Java performs for you, and if any call is made to a `null` reference, Java will automatically throw a `NullPointerException`. So the above bit of code is always superfluous, although you might perform other checks to guard against the appearance of a `NullPointerException`.

There's a whole group of exception types in this category. They're always thrown automatically by Java and you don't include them in your exception specifications. Conveniently enough, they're all grouped together by putting them under a single base class called `RuntimeException`, a perfect example of inheritance: It establishes a family of types that have some characteristics and behaviors in common.

A `RuntimeException` represents a programming error, which is:

1. An error you cannot anticipate. For example, a `null` reference that is outside of your control.

2. An error that you, as a programmer, should have checked for in your code (such as `ArrayIndexOutOfBoundsException` where you should pay attention to the size of the array). An exception that happens from point #1 often becomes an issue for point #2.

Exceptions are a tremendous benefit here because they help in the debugging process.

You never write an exception specification saying that a method might throw a `RuntimeException` (or any type inherited from `RuntimeException`), because they are *unchecked exceptions*. Because they indicate bugs, you don't usually catch a `RuntimeException`—it's dealt with automatically. If you were forced to check for `RuntimeExceptions`, your code could get too messy. Even though you don't typically catch `RuntimeExceptions`, in your own packages you might choose to throw some `RuntimeExceptions`.

What happens when you don't catch such exceptions? The compiler doesn't enforce exception specifications for these so it's plausible that a `RuntimeException` could percolate all the way out to your `main()` method without being caught:

```
// exceptions/NeverCaught.java
// Ignoring RuntimeExceptions
// {ThrowsException}

public class NeverCaught {
  static void f() {
    throw new RuntimeException("From f()");
  }
  static void g() {
    f();
  }
  public static void main(String[] args) {
    g();
  }
}
/* Output:
___[ Error Output ]___
Exception in thread "main" java.lang.RuntimeException:
From f()
        at NeverCaught.f(NeverCaught.java:7)
        at NeverCaught.g(NeverCaught.java:10)
        at NeverCaught.main(NeverCaught.java:13)
*/
```

If a `RuntimeException` gets all the way out to `main()` without being caught, `printStackTrace()` is called for that exception as the program exits.

You can see that a `RuntimeException` (or anything inherited from it) is a special case because the compiler doesn't require an exception specification for these types. The output is reported to `System.err`.

Keep in mind that only exceptions of type `RuntimeException` (and subclasses) can be ignored in your coding because the compiler carefully enforces the handling of all checked exceptions.

You cannot classify Java exception handling as a single-purpose tool. Yes, it is designed to handle those pesky runtime errors that occur from forces outside your code's control, but it's also essential for certain types of programming bugs that the compiler cannot detect.

Performing Cleanup with `finally`

There's often a piece of code you must execute whether or not an exception is thrown within a `try` block. This usually pertains to an operation other than memory recovery (that's handled by the garbage collector). To achieve this effect, you use a `finally` clause[102] at the end of all the exception handlers. The full picture of exception-handling is thus:

```
try {
  // The guarded region: Dangerous activities
  // that might throw A, B, or C
} catch(A a1) {
  // Handler for situation A
} catch(B b1) {
  // Handler for situation B
} catch(C c1) {
  // Handler for situation C
} finally {
  // Activities that happen every time
}
```

This program demonstrate that the `finally` clause always runs:

[102] C++ exception handling does not have the `finally` clause because it relies on destructors to accomplish this sort of cleanup.

```
// exceptions/FinallyWorks.java
// The finally clause is always executed

class ThreeException extends Exception {}

public class FinallyWorks {
  static int count = 0;
  public static void main(String[] args) {
    while(true) {
      try {
        // Post-increment is zero first time:
        if(count++ == 0)
          throw new ThreeException();
        System.out.println("No exception");
      } catch(ThreeException e) {
        System.out.println("ThreeException");
      } finally {
        System.out.println("In finally clause");
        if(count == 2) break; // out of "while"
      }
    }
  }
}
/* Output:
ThreeException
In finally clause
No exception
In finally clause
*/
```

From the output you see that the `finally` clause is executed whether or not an exception is thrown. There's also a hint for dealing with the fact that exceptions in Java do not allow you to resume back to where the exception was thrown, as discussed earlier. If you place your `try` block in a loop, you can establish a condition that must be met before you continue the program. You can also add a `static` counter or some other device to allow the loop to try several different approaches before giving up. This way you can build a greater level of robustness into your programs.

What's `finally` for?

In a language without garbage collection *and* without automatic destructor calls,[103] `finally` is important because it allows the programmer to guarantee the release of memory regardless of what happens in the `try` block. But Java has garbage collection, so releasing memory is virtually never a problem. Also, it has no destructors to call. So when do you use `finally` in Java?

The `finally` clause is necessary to clean up something *other* than memory. Examples include an open file or network connection, something you've drawn on the screen, or even a switch in the outside world:

```
// exceptions/Switch.java

public class Switch {
  private boolean state = false;
  public boolean read() { return state; }
  public void on() {
    state = true;
    System.out.println(this);
  }
  public void off() {
    state = false;
    System.out.println(this);
  }
  @Override public String toString() {
    return state ? "on" : "off";
  }
}


// exceptions/OnOffException1.java
public class OnOffException1 extends Exception {}


// exceptions/OnOffException2.java
public class OnOffException2 extends Exception {}
```

[103] A destructor is a function that's always called when an object becomes unused. You always know exactly where and when the destructor gets called. C++ has automatic destructor calls, and C# (which is much more like Java) has a way that automatic destruction can occur.

```
// exceptions/OnOffSwitch.java
// Why use finally?

public class OnOffSwitch {
  private static Switch sw = new Switch();
  public static void f()
  throws OnOffException1, OnOffException2 {}
  public static void main(String[] args) {
    try {
      sw.on();
      // Code that can throw exceptions...
      f();
      sw.off();
    } catch(OnOffException1 e) {
      System.out.println("OnOffException1");
      sw.off();
    } catch(OnOffException2 e) {
      System.out.println("OnOffException2");
      sw.off();
    }
  }
}
/* Output:
on
off
*/
```

The goal here is to make sure the switch is off when `main()` completes, so `sw.off()` is placed at the end of the `try` block and at the end of each exception handler. But an exception might be thrown that isn't caught here, so `sw.off()` would be missed. However, with `finally` you can place the cleanup code from a `try` block in just one place:

```
// exceptions/WithFinally.java
// Finally Guarantees cleanup

public class WithFinally {
  static Switch sw = new Switch();
  public static void main(String[] args) {
    try {
      sw.on();
      // Code that can throw exceptions...
      OnOffSwitch.f();
    } catch(OnOffException1 e) {
      System.out.println("OnOffException1");
    } catch(OnOffException2 e) {
      System.out.println("OnOffException2");
    } finally {
      sw.off();
    }
  }
}
/* Output:
on
off
*/
```

Here `sw.off()` is guaranteed to run no matter what happens.

Even in cases when the exception is not caught in the current set of `catch` clauses, `finally` is executed before the exception-handling mechanism continues its search for a handler at the next higher level:

```
// exceptions/AlwaysFinally.java
// Finally is always executed

class FourException extends Exception {}

public class AlwaysFinally {
  public static void main(String[] args) {
    System.out.println("Entering first try block");
    try {
      System.out.println("Entering second try block");
      try {
        throw new FourException();
```

```
      } finally {
        System.out.println("finally in 2nd try block");
      }
    } catch(FourException e) {
      System.out.println(
        "Caught FourException in 1st try block");
    } finally {
      System.out.println("finally in 1st try block");
    }
  }
}
/* Output:
Entering first try block
Entering second try block
finally in 2nd try block
Caught FourException in 1st try block
finally in 1st try block
*/
```

The `finally` statement is also executed when `break` and `continue` statements are involved. Together with the labeled `break` and labeled `continue`, `finally` eliminates the need for a `goto` statement in Java.

Using `finally` During `return`

Because a `finally` clause is always executed, it's possible to return from multiple points within a method and still guarantee that important cleanup is performed:

```
// exceptions/MultipleReturns.java

public class MultipleReturns {
  public static void f(int i) {
    System.out.println(
      "Initialization that requires cleanup");
    try {
      System.out.println("Point 1");
      if(i == 1) return;
      System.out.println("Point 2");
      if(i == 2) return;
      System.out.println("Point 3");
```

```
      if(i == 3) return;
      System.out.println("End");
      return;
    } finally {
      System.out.println("Performing cleanup");
    }
  }
  public static void main(String[] args) {
    for(int i = 1; i <= 4; i++)
      f(i);
  }
}
/* Output:
Initialization that requires cleanup
Point 1
Performing cleanup
Initialization that requires cleanup
Point 1
Point 2
Performing cleanup
Initialization that requires cleanup
Point 1
Point 2
Point 3
Performing cleanup
Initialization that requires cleanup
Point 1
Point 2
Point 3
End
Performing cleanup
*/
```

The output shows it doesn't matter where you return, the `finally` clause always runs.

Pitfall: the Lost Exception

Unfortunately, there's a flaw in Java's exception implementation. Although exceptions are an indication of a crisis in your program and should never be ignored, it's

possible for an exception to be lost. This happens with a particular configuration using a `finally` clause:

```
// exceptions/LostMessage.java
// How an exception can be lost

class VeryImportantException extends Exception {
  @Override public String toString() {
    return "A very important exception!";
  }
}

class HoHumException extends Exception {
  @Override public String toString() {
    return "A trivial exception";
  }
}

public class LostMessage {
  void f() throws VeryImportantException {
    throw new VeryImportantException();
  }
  void dispose() throws HoHumException {
    throw new HoHumException();
  }
  public static void main(String[] args) {
    try {
      LostMessage lm = new LostMessage();
      try {
        lm.f();
      } finally {
        lm.dispose();
      }
    } catch(VeryImportantException |
            HoHumException e) {
      System.out.println(e);
    }
  }
}
/* Output:
A trivial exception
*/
```

The output shows no evidence of the `VeryImportantException`, which is replaced by the `HoHumException` in the `finally` clause. This is a rather serious pitfall because it means an exception can be completely lost, and in a far more subtle and difficult-to-detect fashion than the preceding example. In contrast, C++ treats the situation where a second exception is thrown before the first one is handled as a dire programming error. Perhaps a future version of Java will repair this problem (on the other hand, you typically wrap any method that throws an exception, such as `dispose()` in the example above, inside a `try-catch` clause).

An even simpler way to lose an exception is just to `return` from inside a `finally` clause:

```
// exceptions/ExceptionSilencer.java

public class ExceptionSilencer {
  public static void main(String[] args) {
    try {
      throw new RuntimeException();
    } finally {
      // Using 'return' inside the finally block
      // will silence any thrown exception.
      return;
    }
  }
}
```

If you run this program you'll see it produces no output, even though an exception is thrown.

Exception Restrictions

When you override a method, you can throw only the exceptions specified in the base-class version of the method. This is a useful restriction because it means code that works with the base class will automatically work with any object derived it (a fundamental OOP concept), including exceptions.

This example demonstrates the kinds of restrictions imposed (at compile time) for exceptions:

```
// exceptions/StormyInning.java
// Overridden methods can throw only the exceptions
// specified in their base-class versions, or exceptions
// derived from the base-class exceptions

class BaseballException extends Exception {}
class Foul extends BaseballException {}
class Strike extends BaseballException {}

abstract class Inning {
  Inning() throws BaseballException {}
  public void event() throws BaseballException {
    // Doesn't actually have to throw anything
  }
  public abstract void atBat() throws Strike, Foul;
  public void walk() {} // Throws no checked exceptions
}

class StormException extends Exception {}
class RainedOut extends StormException {}
class PopFoul extends Foul {}

interface Storm {
  void event() throws RainedOut;
  void rainHard() throws RainedOut;
}

public
class StormyInning extends Inning implements Storm {
  // OK to add new exceptions for constructors, but you
  // must deal with the base constructor exceptions:
  public StormyInning()
    throws RainedOut, BaseballException {}
  public StormyInning(String s)
    throws BaseballException {}
  // Regular methods must conform to base class:
  //- void walk() throws PopFoul {} //Compile error
  // Interface CANNOT add exceptions to existing
  // methods from the base class:
  //- public void event() throws RainedOut {}
  // If the method doesn't already exist in the
  // base class, the exception is OK:
```

```
  @Override public void rainHard() throws RainedOut {}
  // You can choose to not throw any exceptions,
  // even if the base version does:
  @Override public void event() {}
  // Overridden methods can throw inherited exceptions:
  @Override public void atBat() throws PopFoul {}
  public static void main(String[] args) {
    try {
      StormyInning si = new StormyInning();
      si.atBat();
    } catch(PopFoul e) {
      System.out.println("Pop foul");
    } catch(RainedOut e) {
      System.out.println("Rained out");
    } catch(BaseballException e) {
      System.out.println("Generic baseball exception");
    }
    // Strike not thrown in derived version.
    try {
      // What happens if you upcast?
      Inning i = new StormyInning();
      i.atBat();
      // You must catch the exceptions from the
      // base-class version of the method:
    } catch(Strike e) {
      System.out.println("Strike");
    } catch(Foul e) {
      System.out.println("Foul");
    } catch(RainedOut e) {
      System.out.println("Rained out");
    } catch(BaseballException e) {
      System.out.println("Generic baseball exception");
    }
  }
}
```

In `Inning`, you see that both the constructor and the `event()` method say they will throw an exception, but they never do. This is legal because it forces the user to catch any exceptions that might be added in overridden versions of `event()`. The same idea holds for `abstract` methods, as seen in `atBat()`.

The interface `Storm` contains one method (`event()`) that is defined in `Inning`,

and one method that isn't. Both methods throw a new type of exception, `RainedOut`. When `StormyInning extends Inning` and `implements Storm`, the `event()` method in `Storm` *cannot* change the exception interface of `event()` in `Inning`. Again, this makes sense because otherwise you'd never know if you were catching the correct thing when working with the base class. However, if a method described in an interface is not in the base class, such as `rainHard()`, there's no problem if it throws exceptions.

The restriction on exceptions does not apply to constructors. `StormyInning` shows that a constructor can throw anything it wants, regardless of what the base-class constructor throws. Because a base-class constructor must always be called one way or another (here, the zero-argument constructor is called automatically), the derived-class constructor must declare any base-class constructor exceptions in its exception specification.

A derived-class constructor cannot catch exceptions thrown by its base-class constructor.

The reason `StormyInning.walk()` will not compile is that it throws an exception, but `Inning.walk()` does not. If this were allowed, you could write code that called `Inning.walk()` without handling any exceptions. However, when you substituted an object of a class derived from `Inning`, exceptions would be thrown so your code would break. By forcing the derived-class methods to conform to the exception specifications of the base-class methods, substitutability of objects is maintained.

The overridden `event()` method shows that a derived-class version of a method can choose not to throw any exceptions, even if the base-class version does. Again, this is fine because it doesn't break code that is written assuming the base-class version throws exceptions. Similar logic applies to `atBat()`, which throws `PopFoul`, an exception that is derived from `Foul` thrown by the base-class version of `atBat()`. This way, if you write code that works with `Inning` and calls `atBat()`, you must catch the `Foul` exception. `PopFoul` is derived from `Foul` so the exception handler will also catch `PopFoul`.

The last point of interest is in `main()`. Notice that if you're dealing with exactly a `StormyInning` object, the compiler forces you to catch only the exceptions specific to that class, but if you upcast to the base type, the compiler (correctly) forces you to catch the exceptions for the base type. All these constraints produce much more

robust exception-handling code.[104]

Although exception specifications are enforced by the compiler during inheritance, the exception specifications are not part of the type of a method, which comprises only the method name and argument types. Therefore, you cannot overload methods based on exception specifications. In addition, just because an exception specification exists in a base-class version of a method doesn't mean it must exist in the derived-class version of the method. This is different from inheritance rules, where a method in the base class must also exist in the derived class. Put another way, the "exception specification interface" for a particular method can narrow during inheritance and overriding, but it cannot widen—this is precisely the opposite of the rule for the class interface during inheritance.

Constructors

It's important that you always ask, "If an exception occurs, will everything be properly cleaned up?" Most of the time you're fairly safe, but with constructors there's a problem. The constructor puts the object into a safe starting state, but it might perform some operation—such as opening a file—that doesn't get cleaned up until the user is finished with the object and calls a special cleanup method. If you throw an exception from inside a constructor, these cleanup behaviors might not occur properly. This means you must be especially vigilant when writing a constructor.

You might think `finally` is the solution. But it's not that simple, because `finally` performs the cleanup code *every time*. If a constructor fails partway through its execution, it might not have successfully created some part of the object that is cleaned up in the `finally` clause.

In the following example, a class called `InputFile` opens a file and reads it one line at a time. It uses the classes `FileReader` and `BufferedReader` from the Java standard I/O library discussed in the Appendix: I/O Streams. These classes are simple enough you probably won't have any trouble understanding their basic use:

[104] ISO C++ added similar constraints that require derived-method exceptions to be the same as, or derived from, the exceptions thrown by the base-class method. This is one case where C++ is actually able to check exception specifications at compile time.

```
// exceptions/InputFile.java
// Paying attention to exceptions in constructors
import java.io.*;

public class InputFile {
  private BufferedReader in;
  public InputFile(String fname) throws Exception {
    try {
      in = new BufferedReader(new FileReader(fname));
      // Other code that might throw exceptions
    } catch(FileNotFoundException e) {
      System.out.println("Could not open " + fname);
      // Wasn't open, so don't close it
      throw e;
    } catch(Exception e) {
      // All other exceptions must close it
      try {
        in.close();
      } catch(IOException e2) {
        System.out.println("in.close() unsuccessful");
      }
      throw e; // Rethrow
    } finally {
      // Don't close it here!!!
    }
  }
  public String getLine() {
    String s;
    try {
      s = in.readLine();
    } catch(IOException e) {
      throw new RuntimeException("readLine() failed");
    }
    return s;
  }
  public void dispose() {
    try {
      in.close();
      System.out.println("dispose() successful");
    } catch(IOException e2) {
      throw new RuntimeException("in.close() failed");
    }
```

```
  }
}
```

The constructor for `InputFile` takes a `String` argument: the name of the file to open. Inside a `try` block, it creates a `FileReader` using the file name. A `FileReader` isn't particularly useful until you use it to create a `BufferedReader`. One of the benefits of `InputFile` is that it combines these two actions.

If the `FileReader` constructor is unsuccessful, it throws a `FileNotFoundException`. This is the one case when you don't close the file, because it wasn't successfully opened. Any *other* `catch` clauses must close the file because it *was* opened by the time those `catch` clauses are entered. (This gets trickier if more than one method can throw a `FileNotFoundException`. In that case, you'll usually break things into several `try` blocks.) The `close()` method might throw an exception so it is tried and caught even though it's within the block of another `catch` clause—it's just another pair of curly braces to the Java compiler. After performing local operations, the exception is rethrown, which is appropriate because this constructor failed, and you don't want the calling method to assume that the object was properly created and is valid.

In this example, the `finally` clause is definitely *not* the place to `close()` the file because that would close it every time the constructor completed. We want the file to be open for the useful lifetime of the `InputFile` object.

The `getLine()` method returns a `String` containing the next line in the file. It calls `readLine()`, which can throw an exception, but that exception is caught so `getLine()` doesn't throw any exceptions. One of the design issues with exceptions is whether to handle an exception completely at this level, to handle it partially and pass the same exception (or a different one) on, or whether to simply pass it on. Passing it on, when appropriate, can certainly simplify coding. In this situation, the `getLine()` method *converts* the exception to a `RuntimeException` to indicate a programming error.

The `dispose()` method must be called by the user when the `InputFile` object is no longer needed. This will release the system resources (such as file handles) used by the `BufferedReader` and/or `FileReader` objects. You don't do this until you're finished with the `InputFile` object. You might think of putting such functionality into a `finalize()` method, but as mentioned in the Housekeeping chapter, you can't always be sure that `finalize()` is called (even if you *can* be sure it gets called,

you don't know *when*). This is one of the downsides to Java: All cleanup—other than memory cleanup—doesn't happen automatically, so you must inform the client programmers they are responsible.

The safest way to use a class which might throw an exception during construction and which requires cleanup is to use nested `try` blocks:

```
// exceptions/Cleanup.java
// Guaranteeing proper cleanup of a resource

public class Cleanup {
  public static void main(String[] args) {
    try {
      InputFile in = new InputFile("Cleanup.java");
      try {
        String s;
        int i = 1;
        while((s = in.getLine()) != null)
          ; // Perform line-by-line processing here...
      } catch(Exception e) {
        System.out.println("Caught Exception in main");
        e.printStackTrace(System.out);
      } finally {
        in.dispose();
      }
    } catch(Exception e) {
      System.out.println(
        "InputFile construction failed");
    }
  }
}
/* Output:
dispose() successful
*/
```

Look carefully at the logic here: The construction of the `InputFile` object is effectively in its own `try` block. If that construction fails, the outer `catch` clause is entered and `dispose()` is not called. However, if construction succeeds then you must ensure the object is cleaned up, so immediately after construction you create a new `try` block. The `finally` that performs cleanup is associated with the *inner* `try` block; this way, the `finally` clause is not executed if construction fails, and it is *always* executed if construction succeeds.

This general cleanup idiom should still be used if the constructor throws no exceptions. The basic rule is: Right after you create an object that requires cleanup, begin a `try-finally`:

```
// exceptions/CleanupIdiom.java
// Disposable objects must be followed by a try-finally

class NeedsCleanup { // Construction can't fail
  private static long counter = 1;
  private final long id = counter++;
  public void dispose() {
    System.out.println(
      "NeedsCleanup " + id + " disposed");
  }
}

class ConstructionException extends Exception {}

class NeedsCleanup2 extends NeedsCleanup {
  // Construction can fail:
  NeedsCleanup2() throws ConstructionException {}
}

public class CleanupIdiom {
  public static void main(String[] args) {
    // [1]:
    NeedsCleanup nc1 = new NeedsCleanup();
    try {
      // ...
    } finally {
      nc1.dispose();
    }

    // [2]:
    // If construction cannot fail,
    // you can group objects:
    NeedsCleanup nc2 = new NeedsCleanup();
    NeedsCleanup nc3 = new NeedsCleanup();
    try {
      // ...
    } finally {
      nc3.dispose(); // Reverse order of construction
```

```
      nc2.dispose();
    }

    // [3]:
    // If construction can fail you must guard each one:
    try {
      NeedsCleanup2 nc4 = new NeedsCleanup2();
      try {
        NeedsCleanup2 nc5 = new NeedsCleanup2();
        try {
          // ...
        } finally {
          nc5.dispose();
        }
      } catch(ConstructionException e) { // nc5 const.
        System.out.println(e);
      } finally {
        nc4.dispose();
      }
    } catch(ConstructionException e) { // nc4 const.
      System.out.println(e);
    }
  }
}
/* Output:
NeedsCleanup 1 disposed
NeedsCleanup 3 disposed
NeedsCleanup 2 disposed
NeedsCleanup 5 disposed
NeedsCleanup 4 disposed
*/
```

- **[1]** This is fairly straightforward: You follow a disposable object with a `try-finally`. If the object construction cannot fail, no `catch` is necessary.
- **[2]** Here you see objects with constructors that cannot fail grouped together for both construction and cleanup.
- **[3]** This shows how to deal with objects whose constructors can fail *and* which need cleanup. To properly handle this situation, things get messy, because you must surround each construction with its own `try-catch`, and each object construction must be followed by a `try-finally` to guarantee cleanup.

The messiness of exception handling here is a strong argument for creating constructors that cannot fail, although this is not always possible.

Note that if `dispose()` can throw an exception you might need additional `try` blocks. Basically, you must think carefully about all the possibilities and guard for each one.

Try-With-Resources

The last section might have made your head spin a bit. Figuring out where to put all the `try-catch-finally` blocks becomes intimidating when considering all the ways something can fail. It seems quite challenging to ensure that none of the failure paths leave your system in an unstable state.

`InputFile.java` is a particularly thorny case, because the file is opened (with all the possible exceptions from that), then it is *left open* for the lifetime of the object. Each call to `getLine()` can cause an exception, and so can `dispose()`. This is a good example only because it shows how messy things can be. It also demonstrates that you should try your best *not* to design your code that way (of course, you often get into situations when it's not your choice how the code is designed, so you must still understand it).

A better design for `InputFile.java` is if the constructor reads the file and buffers it internally—that way, the opening, reading, and closing of the file all happen in the constructor. Or, if reading and storing the file is impractical, you can instead produce a `Stream`. Ideally you'd design it more like this:

```
// exceptions/InputFile2.java
import java.io.*;
import java.nio.file.*;
import java.util.stream.*;

public class InputFile2 {
  private String fname;
  public InputFile2(String fname) {
    this.fname = fname;
  }
  public
  Stream<String> getLines() throws IOException {
```

```
    return Files.lines(Paths.get(fname));
  }
  public static void
  main(String[] args) throws IOException {
    new InputFile2("InputFile2.java").getLines()
      .skip(15)
      .limit(1)
      .forEach(System.out::println);
  }
}
/* Output:
  main(String[] args) throws IOException {
*/
```

Now `getLines()` is solely responsible for opening the file and creating the `Stream`.

You can't always sidestep the problem this easily. Sometimes there are objects that:

1. Need cleanup.
2. Need cleanup at a particular moment, when you go out of a scope (by normal means or via an exception).

A common example is `java.io.FileInputStream` (described in the Appendix: I/O Streams). To use it properly, you must write some tricky boilerplate code:

```
// exceptions/MessyExceptions.java
import java.io.*;

public class MessyExceptions {
  public static void main(String[] args) {
    InputStream in = null;
    try {
      in = new FileInputStream(
        new File("MessyExceptions.java"));
      int contents = in.read();
      // Process contents
    } catch(IOException e) {
      // Handle the error
    } finally {
      if(in != null) {
        try {
          in.close();
```

```
        } catch(IOException e) {
          // Handle the close() error
        }
      }
    }
  }
}
```

When the `finally` clause has its own `try` block, it feels like things have become overcomplicated.

Fortunately Java 7 introduced the *try-with-resources* syntax, which cleans up the above code remarkably well:

```
// exceptions/TryWithResources.java
import java.io.*;

public class TryWithResources {
  public static void main(String[] args) {
    try(
      InputStream in = new FileInputStream(
        new File("TryWithResources.java"))
    ) {
      int contents = in.read();
      // Process contents
    } catch(IOException e) {
      // Handle the error
    }
  }
}
```

Before Java 7, a `try` was always followed by a `{`, but now it can be followed by a parenthesized definition—here our creation of the `FileInputStream` object. The part within parentheses is called the *resource specification header*. Now `in` is available throughout the rest of the `try` block. More importantly, no matter how you exit the `try` block (normally or via exception), the equivalent of the previous `finally` clause is executed, but without writing that messy and tricky code. This is an important improvement.

How does it work? The objects created in the try-with-resources definition clause (within the parentheses) must implement the `interface java.lang.AutoCloseable`,

which contains a single method, `close()`. When `AutoCloseable` was introduced in Java 7, many interfaces and classes were modified to implement it. Check the `AutoCloseable` Javadocs to see a list, which includes `Stream` objects:

```
// exceptions/StreamsAreAutoCloseable.java
import java.io.*;
import java.nio.file.*;
import java.util.stream.*;

public class StreamsAreAutoCloseable {
  public static void
  main(String[] args) throws IOException{
    try(
      Stream<String> in = Files.lines(
        Paths.get("StreamsAreAutoCloseable.java"));
      PrintWriter outfile = new PrintWriter(
        "Results.txt");                          // [1]
    ) {
      in.skip(5)
        .limit(1)
        .map(String::toLowerCase)
        .forEachOrdered(outfile::println);
    }                                            // [2]
  }
}
```

- **[1]** You can see another feature here: the resource specification header can contain multiple definitions, separated by semicolons (the final semicolon is accepted but optional). Each object defined in the header will have its `close()` called at the end of the `try` block.
- **[2]** The `try` block for try-with-resources can stand alone, without a `catch` or `finally`. Here, the `IOException` is passed out through `main()` so it doesn't have to be caught at the end of the `try`.

The Java 5 `Closeable` interface was modified to inherit from `AutoCloseable`, so anything that historically supports `Closeable` is also supported by try-with-resources.

Revealing the Details

To investigate the underlying mechanisms of try-with-resources, we'll create our own `AutoCloseable` classes:

```
// exceptions/AutoCloseableDetails.java

class Reporter implements AutoCloseable {
  String name = getClass().getSimpleName();
  Reporter() {
    System.out.println("Creating " + name);
  }
  @Override public void close() {
    System.out.println("Closing " + name);
  }
}

class First extends Reporter {}
class Second extends Reporter {}

public class AutoCloseableDetails {
  public static void main(String[] args) {
    try(
      First f = new First();
      Second s = new Second()
    ) {
    }
  }
}
/* Output:
Creating First
Creating Second
Closing Second
Closing First
*/
```

Exiting the `try` block calls `close()` for both objects *and* it closes them in reverse order of creation. The order is important because in this configuration the `Second` object might depend on the `First` object, so if `First` is already closed by the time `Second` closes, `Second`'s `close()` might try to access some feature of `First` which is no longer available.

Suppose we define an object within the resource specification header that is *not* `AutoCloseable`:

```
// exceptions/TryAnything.java
// {WillNotCompile}

class Anything {}

public class TryAnything {
  public static void main(String[] args) {
    try(
      Anything a = new Anything()
    ) {
    }
  }
}
```

As we hope and expect, Java won't let us do it and emits a compile-time error.

What if one of the constructors throws an exception?

```
// exceptions/ConstructorException.java

class CE extends Exception {}

class SecondExcept extends Reporter {
  SecondExcept() throws CE {
    super();
    throw new CE();
  }
}

public class ConstructorException {
  public static void main(String[] args) {
    try(
      First f = new First();
      SecondExcept s = new SecondExcept();
      Second s2 = new Second()
    ) {
      System.out.println("In body");
    } catch(CE e) {
      System.out.println("Caught: " + e);
```

```
    }
  }
}
/* Output:
Creating First
Creating SecondExcept
Closing First
Caught: CE
*/
```

Now there are three objects defined in the resource specification header, and the middle one throws an exception. Because of this, the compiler forces us to have a `catch` clause to catch the constructor exception. This means the resource specification header is actually enclosed by the `try` block.

As expected, `First` is created without incident, and `SecondExcept` throws an exception during creation. Notice that `close()` is *not* called for `SecondExcept`, because if your constructor fails you can't assume you can do *anything* safely with that object, including close it. Because of the exception from `SecondExcept`, the `Second` object `s2` is never created, so it is not cleaned up.

If no constructors throw exceptions but you might get them in the body of the `try`, you are again forced to provide a `catch` clause:

```
// exceptions/BodyException.java

class Third extends Reporter {}

public class BodyException {
  public static void main(String[] args) {
    try(
      First f = new First();
      Second s2 = new Second()
    ) {
      System.out.println("In body");
      Third t = new Third();
      new SecondExcept();
      System.out.println("End of body");
    } catch(CE e) {
      System.out.println("Caught: " + e);
    }
```

```
  }
}
/* Output:
Creating First
Creating Second
In body
Creating Third
Creating SecondExcept
Closing Second
Closing First
Caught: CE
*/
```

Notice that the `Third` object never gets cleaned up. That's because it was not created inside the resource specification header, so it is not guarded. This is important, because Java provides no guidance here in the form of warnings or errors, so a mistake like this can easily slip through. Indeed, if you rely on some of the integrated development environments to automatically rewrite code to use try-with-resources, they will (at the time of this writing) typically only guard the first object they come across, and miss the rest.

Finally, let's look at `close()` methods that throw exceptions:

```
// exceptions/CloseExceptions.java

class CloseException extends Exception {}

class Reporter2 implements AutoCloseable {
  String name = getClass().getSimpleName();
  Reporter2() {
    System.out.println("Creating " + name);
  }
  @Override
  public void close() throws CloseException {
    System.out.println("Closing " + name);
  }
}

class Closer extends Reporter2 {
  @Override
  public void close() throws CloseException {
```

```
    super.close();
    throw new CloseException();
  }
}

public class CloseExceptions {
  public static void main(String[] args) {
    try(
      First f = new First();
      Closer c = new Closer();
      Second s = new Second()
    ) {
      System.out.println("In body");
    } catch(CloseException e) {
      System.out.println("Caught: " + e);
    }
  }
}
/* Output:
Creating First
Creating Closer
Creating Second
In body
Closing Second
Closing Closer
Closing First
Caught: CloseException
*/
```

Technically we're not forced to provide a `catch` clause here; you can instead report that `main() throws CloseException`. But the `catch` clause is the typical place to put the error-handling code.

Notice that, because all three objects were created, they are all closed—in reverse order—even though `Closer.close()` throws an exception. When you think about it, this is what you want to happen, but if you must code all that logic yourself, you might miss something and get it wrong. Imagine all the code that's out there where the programmers didn't think through all the implications of cleanup, and did it wrong. For that reason you should always use try-with-resources whenever you can. It helps a lot that the feature also makes the resulting code much cleaner and easier to understand.

Exception Matching

When an exception is thrown, the exception-handling system looks through the "nearest" handlers in the order they are written. When it finds a match, the exception is considered handled, and no further searching occurs.

Matching an exception doesn't require a perfect match between the exception and its handler. A derived-class object will match a handler for the base class:

```
// exceptions/Human.java
// Catching exception hierarchies

class Annoyance extends Exception {}
class Sneeze extends Annoyance {}

public class Human {
  public static void main(String[] args) {
    // Catch the exact type:
    try {
      throw new Sneeze();
    } catch(Sneeze s) {
      System.out.println("Caught Sneeze");
    } catch(Annoyance a) {
      System.out.println("Caught Annoyance");
    }
    // Catch the base type:
    try {
      throw new Sneeze();
    } catch(Annoyance a) {
      System.out.println("Caught Annoyance");
    }
  }
}
/* Output:
Caught Sneeze
Caught Annoyance
*/
```

The `Sneeze` exception is caught by the first `catch` clause it matches, which is the first one in the sequence. However, if you remove the first `catch` clause, leaving only the `catch` clause for `Annoyance`, the code still works because it's catching the base

class of `Sneeze`. Put another way, `catch(Annoyance a)` will catch an `Annoyance` *or any class derived from it.* This is useful because if you decide to add more derived exceptions to a method, the client programmer's code won't need changing as long as the client catches the base-class exceptions.

If you try to "mask" the derived-class exceptions by putting the base-class `catch` clause first, like this:

```
try {
  throw new Sneeze();
} catch(Annoyance a) {
  // ...
} catch(Sneeze s) {
  // ...
}
```

The compiler produces an error message because it sees that the `Sneeze` `catch` clause can never be reached.

Alternative Approaches

An exception-handling system is a trapdoor that allows your program to abandon execution of the normal sequence of statements. The trapdoor is used when an "exceptional condition" occurs, such that normal execution is no longer possible or desirable. Exceptions represent conditions that the current method is unable to handle. The reason exception-handling systems were developed is because the approach of dealing with each possible error condition produced by each function call was too onerous, and programmers simply weren't doing it. As a result, they were ignoring the errors. It's worth observing that the issue of programmer convenience in handling errors was a prime motivation for exceptions in the first place.

One of the important guidelines in exception handling is "Don't catch an exception unless you know what to do with it." In fact, one of the important goals of exception handling is to move the error-handling code away from the point where the errors occur. This focuses on your objective in one section of your code, and how you're going to deal with problems in a distinct separate section of your code. As a result, your mainline code is not cluttered with error-handling logic, and it's (hypothetically) much easier to understand and maintain. Exception handling also

tends to reduce the amount of error-handling code, by allowing one handler to deal with many error sites.

Checked exceptions complicate this scenario a bit, because they force you to add `catch` clauses in places where you might not be ready to handle an error. This results in the "harmful if swallowed" problem:

```
try {
  // ... to do something useful
} catch(ObligatoryException e) {} // Gulp!
```

Programmers (myself included, in the 1st edition of *Thinking in Java*) would just do the simplest thing, and "swallow" the exception—often unintentionally, but once you do it, the compiler is satisfied, so unless you remember to revisit and correct the code, the exception is lost. The exception happens, but it vanishes completely when swallowed. Because the compiler forces you to write code right away to handle the exception, this seems like the easiest solution even though it's probably the worst thing you can do.

Horrified upon realizing I had done this, in the 2nd edition of *Thinking in Java*, I "fixed" the problem by printing the stack trace inside the handler (as seen—appropriately—in a number of examples in this chapter). While this is useful to trace the behavior of exceptions, it still indicates you don't really know what to do with the exception at that point in your code. In this section you'll learn about some of the issues and complications arising from checked exceptions, and options you have when dealing with them.

This topic seems simple. But it is not only complicated, it is also an issue of some volatility. There are people who are staunchly rooted on either side of the fence and who feel that the correct answer (theirs) is blatantly obvious. I believe the reason for one of these positions is the distinct benefit seen in going from a poorly typed language like pre-ANSI C to a strong, statically typed language (that is, checked at compile time) like C++ or Java. When you make that transition (as I did), the benefits are so dramatic it can seem like static type checking is always the best answer to most problems. My hope is to relate a little bit of my own evolution that has brought the *absolute* value of static type checking into question; clearly, it's very helpful much of the time, but there's a fuzzy line we cross when it begins to get in the way and become a hindrance (one of my favorite quotes is "All models are wrong. Some are useful.").

History

Exception handling originated in systems like PL/1 and Mesa, and later appeared in CLU, SmallTalk, Modula-3, Ada, Eiffel, C++, Python, Java, and the post-Java languages Ruby and C#. The Java design is similar to C++, except in places where the Java designers felt that the C++ approach caused problems.

To provide programmers with a framework they were more likely to use for error handling and recovery, exception handling was added to C++ rather late in the standardization process, promoted by Bjarne Stroustrup, the language's original author. The model for C++ exceptions came primarily from CLU. However, other languages existed at that time that also supported exception handling: Ada, SmallTalk (both of these had exceptions but no exception specifications) and Modula-3 (which included both exceptions and specifications).

In their seminal paper[105] on the subject, Liskov and Snyder observe that a major defect of languages like C, which report errors in a transient fashion, is that:

> "…every invocation must be followed by a conditional test to determine what the outcome was. This requirement leads to programs that are difficult to read, and probably inefficient as well, thus discouraging programmers from signaling and handling exceptions."

Thus, one of the original motivations of exception handling was to prevent this requirement, but with checked exceptions in Java we commonly see exactly this kind of code. They go on to say:

> "…requiring that the text of a handler be attached to the invocation that raises the exception would lead to unreadable programs in which expressions were broken up with handlers."

Following the CLU approach when designing C++ exceptions, Stroustrup stated that the goal was to reduce the amount of code required to recover from errors. I believe that he was observing that programmers were typically not writing error-handling code in C because the amount and placement of such code was daunting

[105] Barbara Liskov and Alan Snyder, *Exception Handling in CLU*, IEEE Transactions on Software Engineering, Vol. SE-5, No. 6, November 1979. Available from IEEE or ACM.

and distracting. As a result, they were used to doing it the C way, ignoring errors in code and using debuggers to track down problems. To use exceptions, these C programmers had to be convinced to write "additional" code they weren't normally writing. Thus, to draw them into a better way of handling errors, the amount of code they must "add" cannot be onerous. I think it's important to keep this goal in mind when looking at the effects of checked exceptions in Java.

C++ brought an additional idea over from CLU: the exception specification, to programmatically state in the method signature the exceptions that could result from calling that method. The exception specification really has two purposes. It can say, "I'm originating this exception in my code; you handle it." But it can also mean, "I'm ignoring this exception that can occur as a result of my code; you handle it." We've been focusing on the "you handle it" part when looking at the mechanics and syntax of exceptions, but here I'm particularly interested in the fact that we often ignore exceptions and that's what the exception specification can state.

In C++ the exception specification is not part of the type information of a function. The only compile-time checking is to ensure that exception specifications are used consistently; for example, if a function or method throws exceptions, the overloaded or derived versions must also throw those exceptions. Unlike Java, however, no compile-time checking occurs to determine whether or not the function or method will actually throw that exception, or whether the exception specification is complete (that is, whether it accurately describes all exceptions that might be thrown). That validation does happen, but only at run time. If an exception is thrown that violates the exception specification, the C++ program will call the standard library function `unexpected()`.

Because of templates, exception specifications are not used at all in the Standard C++ Library. In Java, there are restrictions on the way that Java generics can be used with exception specifications.

Perspectives

First, it's worth noting that Java effectively invented the checked exception (clearly inspired by C++ exception specifications and the fact that C++ programmers typically don't bother with them). However, it was an experiment which no subsequent language has chosen to duplicate.

Secondly, checked exceptions appear to be an "obvious good thing" when seen in introductory examples and in small programs. It is suggested that the subtle difficulties appear when programs begin getting large. This largeness usually doesn't happen overnight; it creeps. Languages that might not be suited for large-scale projects are used for small projects. These projects grow, and at some point we realize that things have gone from "manageable" to "difficult." This, I'm suggesting, might be the case with too much type checking; in particular, with checked exceptions.

The scale of the program seems to be a significant issue. This is a problem because most discussions tend to use small programs as demonstrations. One of the C# designers observed that:

> "Examination of small programs leads to the conclusion that requiring exception specifications could both enhance developer productivity and enhance code quality, but experience with large software projects suggests a different result—decreased productivity and little or no increase in code quality."[106]

In reference to uncaught exceptions, the CLU creators stated:

> "We felt it was unrealistic to require the programmer to provide handlers in situations where no meaningful action can be taken."[107]

When explaining why a function declaration with no specification means it can throw *any* exception, rather than *no* exceptions, Stroustrup states:

> "However, that would require exception specifications for essentially every function, would be a significant cause for recompilation, and would inhibit cooperation with software written in other languages. This would encourage programmers to subvert the exception-handling mechanisms and to write spurious code to suppress exceptions. It would provide a false sense of security to people who failed to notice the exception."[108]

[106] http://discuss.develop.com/archives/wa.e?A2=ind0011A&L=DOTNET&P=R32820
[107] *Exception Handling in CLU*, Liskov & Snyder.
[108] Bjarne Stroustrup, *The C++ Programming Language, 3rd Edition* (Addison-Wesley, 1997), page 376.

We see this very behavior—subverting the exceptions—happening with checked exceptions in Java.

Martin Fowler (author of *UML Distilled*, *Refactoring*, and *Analysis Patterns*) wrote the following to me:

> "…on the whole I think that exceptions are good, but Java checked exceptions are more trouble than they are worth."

I now think Java's important step was to unify the error-reporting model, so all errors are *reported* using exceptions. This wasn't happening with C++, because, for backward compatibility with C, the old model of just ignoring errors was still available. With consistent reporting via exceptions, exceptions can be used if desired, and if not, they will propagate out to the highest level (such as the console). When Java modified the C++ model so exceptions were the only way to report errors, the extra enforcement of checked exceptions might have become less necessary.

In the past, I have believed strongly that both checked exceptions and static type checking were essential to robust program development. However, both anecdotal and direct experience[109] with languages that are more dynamic than static led me to think the great benefits actually come from:

1. A unified error-reporting model via exceptions, regardless of whether the programmer is forced by the compiler to handle them.
2. Type checking, regardless of *when* it takes place. That is, as long as proper use of a type is enforced, it often doesn't matter if it happens at compile time or run time.

On top of this, there are significant productivity benefits to reducing the compile-time constraints upon the programmer. Indeed, *reflection* and *generics* are required to compensate for the overconstraining nature of static typing, as you shall see in a number of examples throughout the book.

I've been told by some that what I say here constitutes blasphemy, and by uttering these words my reputation will be destroyed, civilizations will fall, and a higher percentage of programming projects will fail. The belief that the compiler can save

[109]Indirectly with SmallTalk via conversations with many experienced programmers in that language; directly with Python (*www.Python.org*).

your project by pointing out errors at compile time runs strong, but it's even more important to realize the limitation of what the compiler is able to do. I emphasize the value of an automated build process and unit testing, which give you far more leverage than you get by trying to turn everything into a syntax error. It's worth keeping in mind that:

> "A good programming language is one that helps programmers write good programs. No programming language will prevent its users from writing bad programs." [110]

In any event, the likelihood of checked exceptions ever being removed from Java seems dim. It's too radical of a language change, and proponents appear to be quite strong. Java has a history and policy of absolute backward compatibility—to give you a sense of this, virtually all Sun software ran on all Sun hardware, no matter how old. However, if you find that some checked exceptions are getting in your way, or especially if you find yourself forced to catch exceptions but you don't know what to do with them, there are some alternatives.

Passing Exceptions to the Console

In simple programs, the easiest way to preserve exceptions without writing a lot of code is to pass them out of `main()` to the console. For example, to open a file for reading (something you'll learn about in detail in the Files chapter), you must open and close a `FileInputStream`, which throws exceptions. For a simple program, you can do this (you'll see this approach used in numerous places throughout this book):

[110] Kees Koster, designer of the CDL language, as quoted by Bertrand Meyer, designer of the Eiffel language, *www.elj.com/elj/v1/n1/bm/right/*.

```
// exceptions/MainException.java
import java.util.*;
import java.nio.file.*;

public class MainException {
  // Pass exceptions to the console:
  public static void
  main(String[] args) throws Exception {
    // Open the file:
    List<String> lines = Files.readAllLines(
      Paths.get("MainException.java"));
    // Use the file ...
  }
}
```

`main()` is like any method, which means it can also have an exception specification. Here the type of exception is `Exception`, the root class of all checked exceptions. By passing it out to the console, you are relieved from writing `try-catch` clauses within the body of `main()`. (Unfortunately, some file I/O can be significantly more complex than it would appear from this example. You'll learn more in the Files chapter and the Appendix: I/O Streams).

Converting Checked to Unchecked Exceptions

Throwing an exception from `main()` is convenient when you're writing simple programs for your own consumption, but is not generally useful. The real problem is when you are writing an ordinary method body, and you call another method and realize, "I have no idea what to do with this exception here, but I can't swallow it or print some banal message." With chained exceptions, a simple solution presents itself. You "wrap" a checked exception inside a `RuntimeException` by passing it to the `RuntimeException` constructor, like this:

```
try {
  // ... to do something useful
} catch(IDontKnowWhatToDoWithThisCheckedException e) {
  throw new RuntimeException(e);
}
```

This seems an ideal way to "turn off" the checked exception—you don't swallow it, and you don't put it in your method's exception specification, but because of exception chaining you don't lose any information from the original exception.

This technique provides the option to ignore the exception and let it bubble up the call stack without being required to write `try-catch` clauses and/or exception specifications. However, you can still catch and handle the specific exception by using `getCause()`, as seen here:

```
// exceptions/TurnOffChecking.java
// "Turning off" Checked exceptions
import java.io.*;

class WrapCheckedException {
  void throwRuntimeException(int type) {
    try {
      switch(type) {
        case 0: throw new FileNotFoundException();
        case 1: throw new IOException();
        case 2: throw new
          RuntimeException("Where am I?");
        default: return;
      }
    } catch(IOException | RuntimeException e) {
      // Adapt to unchecked:
      throw new RuntimeException(e);
    }
  }
}

class SomeOtherException extends Exception {}

public class TurnOffChecking {
  public static void main(String[] args) {
    WrapCheckedException wce =
      new WrapCheckedException();
    // You can call throwRuntimeException() without
    // a try block, and let RuntimeExceptions
    // leave the method:
    wce.throwRuntimeException(3);
    // Or you can choose to catch exceptions:
    for(int i = 0; i < 4; i++)
```

```
      try {
        if(i < 3)
          wce.throwRuntimeException(i);
        else
          throw new SomeOtherException();
      } catch(SomeOtherException e) {
          System.out.println(
            "SomeOtherException: " + e);
      } catch(RuntimeException re) {
        try {
          throw re.getCause();
        } catch(FileNotFoundException e) {
          System.out.println(
            "FileNotFoundException: " + e);
        } catch(IOException e) {
          System.out.println("IOException: " + e);
        } catch(Throwable e) {
          System.out.println("Throwable: " + e);
        }
      }
    }
}
/* Output:
FileNotFoundException: java.io.FileNotFoundException
IOException: java.io.IOException
Throwable: java.lang.RuntimeException: Where am I?
SomeOtherException: SomeOtherException
*/
```

`WrapCheckedException.throwRuntimeException()` contains code that generates different types of exceptions. These are caught and wrapped inside `RuntimeException` objects, so they become the "cause" of those exceptions.

In `TurnOffChecking`, you see it's possible to call `throwRuntimeException()` with no `try` block because the method does not throw any checked exceptions. However, when you're ready to catch exceptions, you still have the ability to catch any exception you want by putting your code inside a `try` block. You start by catching all the exceptions you explicitly know might emerge from the code in your `try` block—here, `SomeOtherException` is caught first. Lastly, you catch `RuntimeException` and `throw` the result of `getCause()` (the wrapped exception). This extracts the originating exceptions, which can then be handled in their own `catch` clauses.

The technique of wrapping a checked exception in a `RuntimeException` is used when appropriate throughout the rest of this book. Another solution is to create your own subclass of `RuntimeException`. This way, it doesn't have to be caught, but someone can catch it if they want.

Exception Guidelines

Use exceptions to:

1. Use try-with-resources whenever possible.
2. Handle problems at the appropriate level. (Avoid catching exceptions unless you know what to do with them.)
3. Fix the problem and re-call the method that caused the exception.
4. Patch things up and continue without retrying the method.
5. Calculate some alternative result instead of what the method was supposed to produce.
6. Do whatever you can in the current context and rethrow the *same* exception to a higher context.
7. Do whatever you can in the current context and throw a *different* exception to a higher context.
8. Terminate the program.
9. Simplify. (If your exception scheme makes things more complicated, it is painful and annoying to use.)
10. Make your library and program safer. (This is a short-term investment for debugging, and a long-term investment for application robustness.)

Summary

Exceptions are integral to programming with Java; you can accomplish only so much without knowing how to work with them. Exceptions are introduced now for that reason—there are many libraries you can't use without handling exceptions.

One of the advantages of exception handling is that it concentrates on the problem you're trying to solve in one place, then deals with the errors from that code in

another place. And although exceptions are generally explained as tools that allow you to *report* and *recover from* errors at run time, I have come to wonder how often the "recovery" aspect is implemented, or even possible. My perception is it is less than 10 percent of the time, and even then it probably amounts to unwinding the stack to a known stable state rather than actually performing any kind of resumptive behavior. Whether or not this is true, I have come to believe that the "reporting" function is where the essential value of exceptions lie. The fact that Java effectively insists that all errors be reported in the form of exceptions gives it a great advantage over languages like C++, which allow you to report errors in several different ways, or not at all. A consistent error-reporting system means you no longer ask the question "Are errors slipping through the cracks?" with each piece of code you write (as long as you don't "swallow" the exceptions, that is!).

As you will see in future chapters, by laying this question to rest—even if you do so by throwing a `RuntimeException`—your design and implementation efforts can be focused on more interesting and challenging issues.

Postscript: Exception Bizarro World

(From a blog post in 2011)

My friend James Ward was trying to create some very straightforward teaching examples using JDBC, and kept getting foiled by checked exceptions. He pointed me to Howard Lewis Ship's post The Tragedy of Checked Exceptions[111]. In particular, James was frustrated by the all the hoops he had to jump through to do something that ought to be simple. Even in the `finally` block he's forced to put in more `try-catch` clauses because closing the connection can also cause exceptions. Where does it end? To do something simple you're forced to jump through hoop after hoop (Note that the try-with-resources statement improves this situation significantly).

We started talking about the Go programming language, which I've been fascinated with because Rob Pike et. al. have clearly asked many very incisive and fundamental questions about language design. Basically, they've taken everything we've started to accept about languages and asked, "Why?" about each one. Learning this language really makes you think and wonder.

[111] http://tapestryjava.blogspot.com/2011/05/tragedy-of-checked-exceptions.html

My impression is that the Go team decided not to make any assumptions and to evolve the language only when it is clear that a feature is necessary. They don't seem to worry about making changes that break old code—they created a rewriting tool so if they make such changes it will rewrite the code for you. This frees them to make the language an ongoing experiment to discover what's really necessary rather than doing *Big Upfront Design*.

One of the most interesting decisions they made is to leave out exceptions altogether. You read that right—they aren't just leaving out checked exceptions. They're leaving out *all* exceptions.

The alternative is very simple, and at first it almost seems C-like. Because Go incorporated tuples from the beginning, you can easily return two objects from a function call:

```
result, err := functionCall()
```

(The `:=` tells Go to define `result` and `err` at this point, and to infer their types).

That's it: for each call you get back the result object and an error object. You can check the error right away (which is typical, because if something fails it's unlikely you'll blithely go on to the next step), or check it later if that works.

At first this seems primitive, a regression to ancient times. But so far I've found that the decisions in Go are very well considered, and worth pondering. Am I simply reacting because my brain is exception-addled? How would this affect James's problem?

It occurs to me I've always seen exception handling as kind of a parallel execution path. If you hit an exception, you jump out of the normal path into this parallel execution path, a kind of "bizarro world" where you are no longer doing the things you wrote, and instead jumping around into catch and finally clauses. It's this alternate-execution-path world that causes the problems James is complaining about.

James creates an object. Ideally, object creation does not cause potential exceptions, but if it does you have to catch those. You have to follow creation with a try-finally to make sure cleanup happens (the Python team realized that cleanup is not really an exceptional condition, but a separate problem, so they created a different language construct—`with`—so as to stop conflating the two). Any call that causes an exception

stops the normal execution path and jumps (via parallel bizarro-world) to the catch clause.

One of the fundamental assumptions about exceptions are that we somehow benefit by collecting all the error handling code at the end of the block rather than handling errors when they occur. In both cases we stop normal execution, but exception handling has an automatic mechanism that throws you out of the normal execution path, jumps you into bizarro-parallel-exception world, then pops you back out again in the right handler.

Jumping into bizarro world causes the problems for James, and it adds more work for all programmers: because you can't know when something will happen (you can slip into bizarro world at any moment), you have to add layers of `try` blocks to ensure that nothing slips through the cracks. You end up having to do extra programming to compensate for the exception mechanism (It seems similar to the extra work required to compensate for shared-memory concurrency).

The Go team made the bold move of questioning all this, and saying, "Let's try it without exceptions and see what happens." Yes, this means you're typically going to handle errors where they occur rather than clumping them all at the end of a try block. But that also means code that is about one thing is localized, and maybe that's not so bad. It might also mean you can't easily combine common error-handling code (unless you identified that common code and put it into a function, also not so bad). But it definitely means you don't have to worry about having more than one possible execution path and all that entails.

Validating Your Code

> You can never guarantee your code is correct. You can only prove it's wrong.

Let's pause our learning of language features and look at some programming fundamentals. Specifically, the problem of making sure your code works properly.

Testing

> If it's not tested, it's broken.

Because Java is a (mostly[112]) statically-typed language, programmers often become too comfortable with the apparent safety of the language, thinking "if the compiler accepts it, it's OK." But static type checking is a very limited form of testing. It only means the compiler accepts the syntax and basic type rules of your code. It doesn't mean the code satisfies the goals of your program. As you gain more programming experience, you'll come to understand that your code almost never satisfies those goals. The first step towards code validation is creating tests that check the code behavior against your goals.

Unit Testing

This is the process of building integrated tests into all code you create, and running those tests every time you build your system. That way, the build process checks for more than just syntax errors. You teach it to check for semantic errors as well.

"Unit" refers to the idea of testing small pieces of your code. Typically, each class has tests checking the behavior of all its methods. "System" testing is different, and checks that the finished program satisfies its requirements.

[112] I say "mostly" because it's possible to write code the compiler can't check, as shown in Reflection.

C-style programming languages, and C++ in particular, traditionally valued performance over programming safety. The reason that developing programs in Java is so much faster than in C++ (roughly twice as fast, by most accounts) is because of Java's safety net: features like garbage collection and improved type checking. By integrating unit testing into your build process, you extend this safety net, resulting in faster development. You can also more easily and boldly refactor your code when you discover design or implementation flaws, and in general produce a better product, more quickly.

My own experience with testing began when I realized that, to guarantee the correctness of code in a book, every program in that book must be automatically extracted, then compiled using an appropriate build system. The build system used in this book is Gradle[113], and after you install the JDK, you can just type `gradlew compileJava` to compile all the code for the book. The effect of automatic extraction and compilation on the code quality of the book was so immediate and dramatic it soon became (in my mind) a requisite for any programming book—how can you trust code you didn't compile? I also discovered that I can make sweeping changes using search-and-replace throughout the book. I know that if I introduce a flaw, the code extractor and the build system flushes it out.

As programs became more complex, I found a serious hole in my system. Compiling programs is clearly an important first step, and for a published book it seems a fairly revolutionary one (because of publishing pressures, you can often open a programming book and discover a coding flaw). However, I received messages from readers reporting semantic problems in my code. Of course, these problems could be discovered only by running the code. I took some early faltering steps toward implementing a system to perform automatic execution tests, but succumbed to publishing schedules, all the while knowing there was definitely something wrong with my process that would come back to bite me in the form of embarrassing bug reports.

I had also gotten regular complaints that I didn't show enough program output. I needed to validate the output of a program while showing the validated output in the book. My previous attitude was that the reader should be running the programs while reading the book, and many readers did just that and benefited from it. A hidden reason for that attitude, however, was I didn't have a way to prove the output

[113] https://gradle.org/

shown in the book was correct. From experience, I knew that over time, something would happen so the output was no longer correct (or, I wouldn't get it right in the first place). To solve this problem, I created a tool in the Python language (you will find this tool in the downloaded examples). Most programs in this book produce console output, and this tool compares that output to the expected output shown in the comment at the end of the source-code listing, so readers can see the expected output, and know this output is verified by the build process.

JUnit

The original JUnit, published in 2000, was presumably based on Java 1.0 and thus could not make use of Java's reflection facilities (see Reflection). As a result, writing unit tests with the old JUnit was a rather busy and wordy activity. I found the design unpleasant, and wrote my own unit testing framework as an example for the Annotations chapter. This framework went to the other extreme, "trying the simplest thing that could possibly work" (A key phrase from *Extreme Programming* (XP)). Since then, JUnit has been vastly improved using reflection and annotations, which greatly simplifies the process of writing unit test code. For Java 8, they even added support for lambdas. This book uses *JUnit5*, the latest version at the time.

In the simplest use of JUnit, you tag each method that represents a test with the `@Test` annotation. JUnit identifies these methods as individual tests and sets up and runs them one at a time, taking measures to avoid side effects between tests.

Let's try a simple example. `CountedList` inherits `ArrayList`, with added information to keep track of how many `CountedLists` are created:

```
// validating/CountedList.java
// Keeps track of how many of itself are created.
package validating;
import java.util.*;

public class CountedList extends ArrayList<String> {
  private static int counter = 0;
  private int id = counter++;
  public CountedList() {
    System.out.println("CountedList #" + id);
  }
  public int getId() { return id; }
}
```

Standard practice is to put tests in their own subdirectory. Tests must also be in packages so JUnit can discover them:

```
// validating/tests/CountedListTest.java
// Simple use of JUnit to test CountedList.
package validating;
import java.util.*;
import org.junit.jupiter.api.*;
import static org.junit.jupiter.api.Assertions.*;

public class CountedListTest {
  private CountedList list;
  @BeforeAll
  static void beforeAllMsg() {
    System.out.println(">>> Starting CountedListTest");
  }
  @AfterAll
  static void afterAllMsg() {
    System.out.println(">>> Finished CountedListTest");
  }
  @BeforeEach
  public void initialize() {
    list = new CountedList();
    System.out.println("Set up for " + list.getId());
    for(int i = 0; i < 3; i++)
      list.add(Integer.toString(i));
  }
  @AfterEach
  public void cleanup() {
    System.out.println("Cleaning up " + list.getId());
  }
  @Test
  public void insert() {
    System.out.println("Running testInsert()");
    assertEquals(list.size(), 3);
    list.add(1, "Insert");
    assertEquals(list.size(), 4);
    assertEquals(list.get(1), "Insert");
  }
  @Test
  public void replace() {
    System.out.println("Running testReplace()");
```

```
    assertEquals(list.size(), 3);
    list.set(1, "Replace");
    assertEquals(list.size(), 3);
    assertEquals(list.get(1), "Replace");
  }
  // A helper method to simplify the code. As
  // long as it's not annotated with @Test, it will
  // not be automatically executed by JUnit.
  private
  void compare(List<String> lst, String[] strs) {
    assertArrayEquals(lst.toArray(new String[0]), strs);
  }
  @Test
  public void order() {
    System.out.println("Running testOrder()");
    compare(list, new String[] { "0", "1", "2" });
  }
  @Test
  public void remove() {
    System.out.println("Running testRemove()");
    assertEquals(list.size(), 3);
    list.remove(1);
    assertEquals(list.size(), 2);
    compare(list, new String[] { "0", "2" });
  }
  @Test
  public void addAll() {
    System.out.println("Running testAddAll()");
    list.addAll(Arrays.asList(new String[] {
      "An", "African", "Swallow"}));
    assertEquals(list.size(), 6);
    compare(list, new String[] { "0", "1", "2",
       "An", "African", "Swallow" });
  }
}
/* Output:
>>> Starting CountedListTest
CountedList #0
Set up for 0
Running testRemove()
Cleaning up 0
CountedList #1
```

```
Set up for 1
Running testReplace()
Cleaning up 1
CountedList #2
Set up for 2
Running testAddAll()
Cleaning up 2
CountedList #3
Set up for 3
Running testInsert()
Cleaning up 3
CountedList #4
Set up for 4
Running testOrder()
Cleaning up 4
>>> Finished CountedListTest
*/
```

`@BeforeAll` annotates a method that runs once before any other test operations. `@AfterAll` is for a method that runs once after all other test operations. Both methods must be `static`.

`@BeforeEach` annotates a method typically used to create and initialize a common set of objects and runs before each test. Alternatively, you can put all such initialization in the constructor for the test class, although I think `@BeforeEach` makes it clearer. JUnit creates an object for each test to ensure there are no side effects between test runs. However, all objects for all tests are created at once (rather than creating the object right before the test), so the only difference between using `@BeforeEach` and the constructor is `@BeforeEach` is called directly before the test. In most situations this is not an issue, and you can use the constructor approach if you prefer.

If you must perform cleanup after each test (if you modify any `statics` that need restoring, open files that need closing, open databases or network connections, etc.), annotate a method with `@AfterEach`.

Each test creates a new `CountedListTest` object, thus any non-`static` members are also created at that time. `initialize()` is then called for that test, so `list` is assigned a new `CountedList` object which is then initialized with the `Strings` `"0"`, `"1"`, and `"2"`.

To observe the behavior of `@BeforeEach` and `@AfterEach`, those methods display

information about the test as it is initialized and cleaned up.

`insert()` and `replace()` demonstrate typical test methods. JUnit discovers these methods using the `@Test` annotation and runs each one as a test. Inside the methods, you perform any desired operations and use JUnit assertion methods (which all start with the name "assert") to verify the correctness of your tests (the full range of "assert" statements is found in the JUnit docs). If the assertion fails, the expression and values that caused the failure are displayed. This is often enough, but you can also use the overloaded version of each JUnit assertion statement and include a `String` for display when the assertion fails.

The assertion statements are not required; you can also run the test without assertions and consider it a success if there are no exceptions.

`compare()` is an example of a "helper method" that is not executed by JUnit but instead is used by other tests in the class. As long as there's no `@Test` annotation, JUnit doesn't run it or expect a particular signature. Here, `compare()` is `private`, emphasizing it is used only within the test class, but it can also be `public`. The remaining test methods eliminate duplicate code by refactoring it into the `compare()` method.

The `build.gradle` file for this book controls testing. To run the tests for this chapter, the command is:

```
gradlew validating:test
```

Gradle doesn't run tests that have already run for that build, so if you get no test results, first run:

```
gradlew validating:clean
```

You can run all tests in the book with the command:

```
gradlew test
```

Although you can probably survive with the simplest approach to JUnit as shown in `CountedListTest.java`, JUnit contains numerous additional testing structures you can learn about at junit.org[114].

JUnit is the most popular unit testing framework for Java, but there are alternatives. You can explore others via the Internet in case one of those better suits your needs.

[114] http://junit.org/

The Illusion of Test Coverage

Test coverage, also called *code coverage*, measures the test percentage for your code base. The higher the percentage, the greater the test coverage. There are a number of approaches[115] for calculating coverage, and a helpful article describing Java Code Coverage Tools[116].

It is far too easy for persons without knowledge but in positions of control to decide that 100% coverage is the only acceptable value. This is a problem because the number is not really a good measure of testing effectiveness. You might test everything that needs it, but only measure 65% test coverage. If someone demands 100% coverage, you waste a lot of time generating the rest, and more as you add code to the project.

Test coverage as a rough measure is useful when you're analyzing an unknown code base. If a coverage tool reports an especially low value (say, less than 40%), that tells you the coverage is probably insufficient. However, a very high value is equally suspicious, suggesting that someone with inadequate knowledge of the programming field has forced an arbitrary decision on the team. The best use of coverage tools is to discover untested parts of your codebase. However, don't rely on coverage to tell you anything about the quality of your tests.

Preconditions

The concept of the *precondition* comes from *Design By Contract* (DbC) and is implemented using the basic *assertion* mechanism. We'll start by looking at assertions in Java, then cover DbC, and finally use the Google Guava library for preconditions.

Assertions

Assertions increase program robustness by verifying that certain conditions are satisfied during the execution of your program.

For example, suppose you have a numerical field in an object that represents the month on the Julian calendar. You know this value must always be in the range 1-12. An assertion can check this and report an error if it somehow falls outside that

[115] https://en.wikipedia.org/wiki/Code_coverage
[116] https://en.wikipedia.org/wiki/Java_Code_Coverage_Tools

range. If you're inside a method, you can check the validity of an argument with an assertion. These are important tests to make sure your program is correct, but they cannot be checked at compile-time, and they do not fall under the purview of unit testing.

Java Assertion Syntax

You can simulate the effect of assertions using other programming constructs, so the point of including assertions in Java is that they are easy to write. Assertion statements come in two forms:

```
assert boolean-expression;
assert boolean-expression: information-expression;
```

Both say "I assert this `boolean-expression` produces a `true` value." If this is not the case, the assertion produces an `AssertionError` exception. This is a `Throwable` subclass, and as such doesn't require an exception specification.

Unfortunately, an exception from the first form of assertion does *not* produce any information containing the `boolean-expression` (in contrast with most other languages' assertion mechanisms). Here's an example showing the use of the first form:

```
// validating/Assert1.java
// Non-informative style of assert
// Must run using -ea flag:
// {java -ea Assert1}
// {ThrowsException}

public class Assert1 {
  public static void main(String[] args) {
    assert false;
  }
}
/* Output:
___[ Error Output ]___
Exception in thread "main" java.lang.AssertionError
        at Assert1.main(Assert1.java:9)
*/
```

If you run the program normally, without any special assertion flags, nothing happens. You must explicitly enable assertions when you run the program. The easiest way to do this is with the `-ea` flag, which can also be spelled out: `-enableassertions`. This runs the program and executes any assertion statements.

The output doesn't contain much in the way of useful information. On the other hand, if you use the *information-expression*, you'll produce a helpful message as part of the exception stack trace. The most useful *information-expression* is typically a `String` of text directed at the programmer:

```
// validating/Assert2.java
// Assert with an information-expression
// {java Assert2 -ea}
// {ThrowsException}

public class Assert2 {
  public static void main(String[] args) {
    assert false:
      "Here's a message saying what happened";
  }
}
/* Output:
___[ Error Output ]___
Exception in thread "main" java.lang.AssertionError:
Here's a message saying what happened
        at Assert2.main(Assert2.java:8)
*/
```

The *information-expression* can produce any kind of object, so you typically construct a more complex `String` containing the value(s) of objects that were involved with the failed assertion.

You can also turn assertions on and off based on the class name or package name; that is, you can enable or disable assertions for an entire package. Details for doing this are in the JDK documentation on assertions. This feature is useful for a large project instrumented with assertions when you want to turn some of them on or off. However, Logging or Debugging, described in their respective sections, are probably better tools for capturing that kind of information.

There's one other way you can control assertions: programmatically, by hooking into the `ClassLoader` object. There are several methods in `ClassLoader` that allow

the dynamic enabling and disabling of assertions, including `setDefaultAssertionStatus()`, which sets the assertion status for all the classes loaded afterward. So you might think you can silently turn on assertions like this:

```
// validating/LoaderAssertions.java
// Using the class loader to enable assertions
// {ThrowsException}

public class LoaderAssertions {
  public static void main(String[] args) {
    ClassLoader.getSystemClassLoader()
      .setDefaultAssertionStatus(true);
    new Loaded().go();
  }
}

class Loaded {
  public void go() {
    assert false: "Loaded.go()";
  }
}
/* Output:
___[ Error Output ]___
Exception in thread "main" java.lang.AssertionError:
Loaded.go()
        at Loaded.go(LoaderAssertions.java:15)
        at
LoaderAssertions.main(LoaderAssertions.java:9)
*/
```

This eliminates the need for the `-ea` flag on the command line when running the program. It may be just as straightforward to enable assertions using the `-ea` flag. When delivering a standalone product, you probably have to set up an execution script enabling the user to start the program anyway, to configure other startup parameters.

It does make sense, however, to decide to *require* assertions enabled when the program runs. You can accomplish this with the following `static` clause, placed in the main class of your system:

```
static {
  boolean assertionsEnabled = false;
  // Note intentional side effect of assignment:
  assert assertionsEnabled = true;
  if(!assertionsEnabled)
    throw new RuntimeException("Assertions disabled");
}
```

If assertions are enabled, then the `assert` statement executes and `assertionsEnabled` becomes `true`. The assertion never fails because the return value of the assignment is the assigned value. If assertions are not enabled, the `assert` statement doesn't execute and `assertionsEnabled` remains `false`, resulting in the exception.

Guava Assertions

Because turning on native Java assertions is a bother, the Guava team added a `Verify` class with replacement assertions always enabled. The team recommends importing the `Verify` methods statically:

```
// validating/GuavaAssertions.java
// Assertions that are always enabled.
import com.google.common.base.*;
import static com.google.common.base.Verify.*;

public class GuavaAssertions {
  public static void main(String[] args) {
    verify(2 + 2 == 4);
    try {
      verify(1 + 2 == 4);
    } catch(VerifyException e) {
      System.out.println(e);
    }
    try {
      verify(1 + 2 == 4, "Bad math");
    } catch(VerifyException e) {
      System.out.println(e.getMessage());
    }
    try {
      verify(1 + 2 == 4, "Bad math: %s", "not 4");
    } catch(VerifyException e) {
      System.out.println(e.getMessage());
```

```
    }
    String s = "";
    s = verifyNotNull(s);
    s = null;
    try {
      verifyNotNull(s);
    } catch(VerifyException e) {
      System.out.println(e.getMessage());
    }
    try {
      verifyNotNull(
        s, "Shouldn't be null: %s", "arg s");
    } catch(VerifyException e) {
      System.out.println(e.getMessage());
    }
  }
}
/* Output:
com.google.common.base.VerifyException
Bad math
Bad math: not 4
expected a non-null reference
Shouldn't be null: arg s
*/
```

There are two methods, `verify()` and `verifyNotNull()` with variations to support useful error messages. Note that `verifyNotNull()`'s built-in error message is typically enough, while `verify()` is too general to have a useful default error message.

Using Assertions for *Design By Contract*

Design by Contract (DbC) is a concept developed by Bertrand Meyer, inventor of the Eiffel programming language, to help create robust programs by guaranteeing that objects follow certain rules.[117] These rules are determined by the nature of the problem being solved, which is outside the scope that the compiler can validate.

Although assertions do not directly implement DBC (as does the Eiffel language), they create an informal style of DbC programming.

[117] Design by contract is described in detail in Chapter 11 of *Object-Oriented Software Construction, 2nd Edition*, by Bertrand Meyer (Prentice Hall, 1997).

DbC presumes a clearly-specified contract between the supplier of a service and the consumer or client of that service. In object-oriented programming, services are usually supplied by objects, and the boundary of the object—the division between the supplier and consumer—is the interface of an object's class. When clients call a particular public method, they expect certain behavior from that call: a state change in the object, and a predictable return value. Meyer's thesis is that:

1. This behavior can be clearly specified, as if it were a contract.
2. This behavior can be guaranteed by implementing certain run-time checks, which he calls *preconditions*, *postconditions* and *invariants*.

Whether or not you agree that point 1 is always true, it does appear true for enough situations to make DbC a useful approach. (I believe that, like any solution, there are boundaries to its usefulness. But if you know these boundaries, you know when to try it.) In particular, a valuable part of the design process is the expression of DbC constraints for a particular class; if you are unable to specify the constraints, you probably don't know enough about what you're trying to build.

Check Instructions

Before looking at DbC in detail, consider the simplest use for assertions, which Meyer calls the *check instruction*. A check instruction states your conviction that a particular property is satisfied at this point in the code. The idea of the check instruction is to express non-obvious conclusions in code, not only to verify the test, but also as documentation for future readers of the code.

In chemistry, you might titrate one clear liquid into another, and when you reach a certain point, everything turns blue. This is not obvious from the color of the two liquids; it is part of a complex reaction. A useful check instruction at the completion of the titration process would assert that the resulting liquid is blue.

Check instructions are a valuable addition to your code, and should be used whenever you can test and illuminate the state of your object or program.

Precondition Testing

Preconditions ensure the client (the code calling this method) fulfills its part of the contract. This almost always means checking the arguments at the very beginning

of a method call (before you do anything else in that method) to guarantee they are appropriate for use in the method. You never know what a client will hand you so precondition checks are always a good idea.

Postconditions

Postconditions test the results of what you did in the method. This code is placed at the end of the method call, before the `return` statement (if there is one). For long, complex methods where calculation results should be verified before returning them (that is, in situations where for some reason you cannot always trust the results), postcondition checks are essential, but any time you can describe constraints on the result of the method, it's wise to express those constraints in code as a postcondition.

Invariants

An invariant gives guarantees about the state of the object that must be maintained between method calls. However, it doesn't restrain a method from temporarily diverging from those guarantees during the execution of the method. It just says that the state information of the object will always obey the stated rules:

1. Upon entry to the method.
2. Before leaving the method.

In addition, the invariant is a guarantee about the state of the object after construction.

According to this description, an effective invariant is defined as a method, probably named `invariant()`, which is invoked after construction, and at the beginning and end of each method. The method could be invoked as:

```
assert invariant();
```

This way, if you disable assertions for performance reasons, there's no overhead.

Relaxing DbC

Although he emphasizes the value of expressing preconditions, postconditions, and invariants, and the importance of using these during development, Meyer admits it is not always practical to include all DbC code in a shipping product. You can relax DbC checking based on the amount of trust you can place in the code at a particular point. Here is the order of relaxation, from safest to least safe:

1. The invariant check at the beginning of each method is disabled first because the invariant check at the end of each method guarantees that the object's state is valid at the beginning of every method call. That is, you can generally trust that the state of the object will not change in between method calls. This one is such a safe assumption that you might choose to write code with invariant checks only at the end.
2. The postcondition check is disabled next, when you have reasonable unit testing to verify that your methods are returning appropriate values. The invariant check is watching the state of the object so the postcondition check is only validating the results of the calculation during the method, and therefore may be discarded in favor of unit testing. The unit testing will not be as safe as a run-time postcondition check, but it may be enough, especially if you have confidence in the code.
3. The invariant check at the end of a method call can be disabled if you are confident the method body does not put the object into an invalid state. It might be possible to verify this with *white-box* unit testing (that is, unit tests that have access to private fields, to validate the object state). Thus, although it may not be as robust as calls to `invariant()`, it is possible to "migrate" the invariant checking from run-time tests to build-time tests (via unit testing), just as with postconditions.
4. Finally, as a last resort, disable precondition checks. This is the least safe and least advisable option, because although you know and have control over your own code, you have no control over what arguments the client may pass to a method. However, in a situation where (A) performance is desperately needed and profiling has pointed at precondition checks as a bottleneck and (B) you have some kind of reasonable assurance that the client will not violate preconditions (as in the case where you've written the client code yourself) it may be acceptable to disable precondition checks.

You shouldn't remove the code that performs the checks described here as you disable the checks (just comment it out). If a bug is discovered, you can easily recover the checks to rapidly discover the problem.

DbC + Unit Testing

The following example demonstrates the potency of combining concepts from Design by Contract with unit testing. It shows a small first-in, first-out (FIFO) queue implemented as a "circular" array—that is, an array used in a circular fashion. When the end of the array is reached, the class wraps back around to the beginning.

We can make a number of contractual definitions for this queue:

1. **Precondition (for a `put()`)**: Null elements are not allowed to be added to the queue.
2. **Precondition (for a `put()`)**: It is illegal to put elements into a full queue.
3. **Precondition (for a `get()`)**: It is illegal to try to get elements from an empty queue.
4. **Postcondition (for a `get()`)**: Null elements cannot be produced from the array.
5. **Invariant**: The region that contains objects cannot contain any null elements.
6. **Invariant**: The region that doesn't contain objects must have only null values.

Here is one way to implement these rules, using explicit method calls for each type of DbC element. First, we create a dedicated `Exception`:

```
// validating/CircularQueueException.java
package validating;

public class
CircularQueueException extends RuntimeException {
  public CircularQueueException(String why) {
    super(why);
  }
}
```

This is used to report errors with the `CircularQueue` class:

```
// validating/CircularQueue.java
// Demonstration of Design by Contract (DbC)
package validating;
import java.util.*;

public class CircularQueue {
  private Object[] data;
  private int
    in = 0, // Next available storage space
    out = 0; // Next gettable object
  // Has it wrapped around the circular queue?
  private boolean wrapped = false;
  public CircularQueue(int size) {
    data = new Object[size];
    // Must be true after construction:
    assert invariant();
  }
  public boolean empty() {
    return !wrapped && in == out;
  }
  public boolean full() {
    return wrapped && in == out;
  }
  public boolean isWrapped() { return wrapped; }
  public void put(Object item) {
    precondition(item != null, "put() null item");
    precondition(!full(),
      "put() into full CircularQueue");
    assert invariant();
    data[in++] = item;
    if(in >= data.length) {
      in = 0;
      wrapped = true;
    }
    assert invariant();
  }
  public Object get() {
    precondition(!empty(),
      "get() from empty CircularQueue");
    assert invariant();
    Object returnVal = data[out];
    data[out] = null;
```

```
    out++;
    if(out >= data.length) {
      out = 0;
      wrapped = false;
    }
    assert postcondition(
      returnVal != null,
        "Null item in CircularQueue");
    assert invariant();
    return returnVal;
  }
  // Design-by-contract support methods:
  private static void
  precondition(boolean cond, String msg) {
    if(!cond) throw new CircularQueueException(msg);
  }
  private static boolean
  postcondition(boolean cond, String msg) {
    if(!cond) throw new CircularQueueException(msg);
    return true;
  }
  private boolean invariant() {
    // Guarantee that no null values are in the
    // region of 'data' that holds objects:
    for(int i = out; i != in; i = (i + 1) % data.length)
      if(data[i] == null)
        throw new CircularQueueException(
          "null in CircularQueue");
    // Guarantee that only null values are outside the
    // region of 'data' that holds objects:
    if(full()) return true;
    for(int i = in; i != out; i = (i + 1) % data.length)
      if(data[i] != null)
        throw new CircularQueueException(
          "non-null outside of CircularQueue range: "
          + dump());
    return true;
  }
  public String dump() {
    return "in = " + in +
      ", out = " + out +
      ", full() = " + full() +
```

```
      ", empty() = " + empty() +
      ", CircularQueue = " + Arrays.asList(data);
  }
}
```

The `in` counter indicates the place in the array where the next object goes. The `out` counter indicates where the next object comes from. The `wrapped` flag means `in` has gone "around the circle" and is now coming up from behind `out`. When `in` and `out` coincide, the queue is empty (if `wrapped` is false) or full (if `wrapped` is true).

The `put()` and `get()` methods call `precondition()`, `postcondition()`, and `invariant()`, which are `private` methods defined further down in the class. `precondition()` and `postcondition()` are helper methods designed to clarify the code. Note that `precondition()` returns `void`, because it is not used with `assert`. As previously noted, you'll generally keep preconditions in your code. By wrapping them in a `precondition()` method call, you have better options if you are reduced to the dire move of turning them off.

`postcondition()` and `invariant()` both return a Boolean value so they can be used in `assert` statements. Then, if you disable assertions for performance reasons, there are no method calls at all.

`invariant()` performs internal validity checks on the object. This is an expensive operation if you do it at both the start and end of every method call, as Meyer suggests. However, it's valuable to clearly represent in code, and it helped me debug the implementation. In addition, if you make any changes to the implementation, the `invariant()` ensures you haven't broken the code. But it's fairly trivial to move the invariant tests from the method calls into the unit test code. If your unit tests are thorough, you have a reasonable level of confidence that invariants are respected.

The `dump()` helper method returns a `String` with all the data, rather than printing the data directly. This allows more options for using the information.

Now we can create JUnit tests for the class:

```
// validating/tests/CircularQueueTest.java
package validating;
import org.junit.jupiter.api.*;
import static org.junit.jupiter.api.Assertions.*;

public class CircularQueueTest {
  private CircularQueue queue = new CircularQueue(10);
  private int i = 0;
  @BeforeEach
  public void initialize() {
    while(i < 5) // Pre-load with some data
      queue.put(Integer.toString(i++));
  }
  // Support methods:
  private void showFullness() {
    assertTrue(queue.full());
    assertFalse(queue.empty());
    System.out.println(queue.dump());
  }
  private void showEmptiness() {
    assertFalse(queue.full());
    assertTrue(queue.empty());
    System.out.println(queue.dump());
  }
  @Test
  public void full() {
    System.out.println("testFull");
    System.out.println(queue.dump());
    System.out.println(queue.get());
    System.out.println(queue.get());
    while(!queue.full())
      queue.put(Integer.toString(i++));
    String msg = "";
    try {
      queue.put("");
    } catch(CircularQueueException e) {
      msg = e.getMessage();
      System.out.println(msg);
    }
    assertEquals(msg, "put() into full CircularQueue");
    showFullness();
  }
```

```
  @Test
  public void empty() {
    System.out.println("testEmpty");
    while(!queue.empty())
      System.out.println(queue.get());
    String msg = "";
    try {
      queue.get();
    } catch(CircularQueueException e) {
      msg = e.getMessage();
      System.out.println(msg);
    }
    assertEquals(msg, "get() from empty CircularQueue");
    showEmptiness();
  }
  @Test
  public void nullPut() {
    System.out.println("testNullPut");
    String msg = "";
    try {
      queue.put(null);
    } catch(CircularQueueException e) {
      msg = e.getMessage();
      System.out.println(msg);
    }
    assertEquals(msg, "put() null item");
  }
  @Test
  public void circularity() {
    System.out.println("testCircularity");
    while(!queue.full())
      queue.put(Integer.toString(i++));
    showFullness();
    assertTrue(queue.isWrapped());
    while(!queue.empty())
      System.out.println(queue.get());
    showEmptiness();
    while(!queue.full())
      queue.put(Integer.toString(i++));
    showFullness();
    while(!queue.empty())
      System.out.println(queue.get());
```

```
    showEmptiness();
  }
}
/* Output:
testNullPut
put() null item
testCircularity
in = 0, out = 0, full() = true, empty() = false,
CircularQueue =
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
0
1
2
3
4
5
6
7
8
9
in = 0, out = 0, full() = false, empty() = true,
CircularQueue =
[null, null, null, null, null, null, null, null, null,
null]
in = 0, out = 0, full() = true, empty() = false,
CircularQueue =
[10, 11, 12, 13, 14, 15, 16, 17, 18, 19]
10
11
12
13
14
15
16
17
18
19
in = 0, out = 0, full() = false, empty() = true,
CircularQueue =
[null, null, null, null, null, null, null, null, null,
null]
testFull
```

```
in = 5, out = 0, full() = false, empty() = false,
CircularQueue =
[0, 1, 2, 3, 4, null, null, null, null, null]
0
1
put() into full CircularQueue
in = 2, out = 2, full() = true, empty() = false,
CircularQueue =
[10, 11, 2, 3, 4, 5, 6, 7, 8, 9]
testEmpty
0
1
2
3
4
get() from empty CircularQueue
in = 5, out = 5, full() = false, empty() = true,
CircularQueue =
[null, null, null, null, null, null, null, null, null,
null]
*/
```

The `initialize()` method adds some data so the `CircularQueue` is partially full for each test. The support methods `showFullness()` and `showEmptiness()` indicate that the `CircularQueue` is full or empty, respectively. Each of the four test methods ensures that a different aspect of the `CircularQueue` functions correctly.

Note that by combining DbC with unit testing, you not only get the best of both worlds, but you also have a migration path—you can move some DbC tests to unit tests rather than simply disabling them, so you still have some level of testing.

Using Guava Preconditions

In Relaxing DbC, I pointed out that the precondition is the one part of DbC you don't want to remove, as it checks the validity of method arguments. That's something you have no control over, so you do want to check them. Because Java disables assertions by default, it's usually better to use a different library that's always validating method arguments.

Google's *Guava* library incorporates a nice set of precondition tests that are not only easy to use, but also descriptively well-named. Here you see simple usages of all of them. The library designers recommend you import the preconditions statically:

```
// validating/GuavaPreconditions.java
// Demonstrating Guava Preconditions
import java.util.function.*;
import static com.google.common.base.Preconditions.*;

public class GuavaPreconditions {
  static void test(Consumer<String> c, String s) {
    try {
      System.out.println(s);
      c.accept(s);
      System.out.println("Success");
    } catch(Exception e) {
      String type = e.getClass().getSimpleName();
      String msg = e.getMessage();
      System.out.println(type +
        (msg == null ? "" : ": " + msg));
    }
  }
  public static void main(String[] args) {
    test(s -> s = checkNotNull(s), "X");
    test(s -> s = checkNotNull(s), null);
    test(s -> s = checkNotNull(s, "s was null"), null);
    test(s -> s = checkNotNull(
      s, "s was null, %s %s", "arg2", "arg3"), null);

    test(s -> checkArgument(s == "Fozzie"), "Fozzie");
    test(s -> checkArgument(s == "Fozzie"), "X");
    test(s -> checkArgument(s == "Fozzie"), null);
    test(s -> checkArgument(
      s == "Fozzie", "Bear Left!"), null);
    test(s -> checkArgument(
      s == "Fozzie", "Bear Left! %s Right!", "Frog"),
      null);

    test(s -> checkState(s.length() > 6), "Mortimer");
    test(s -> checkState(s.length() > 6), "Mort");
    test(s -> checkState(s.length() > 6), null);
```

```
    test(s ->
      checkElementIndex(6, s.length()), "Robert");
    test(s ->
      checkElementIndex(6, s.length()), "Bob");
    test(s ->
      checkElementIndex(6, s.length()), null);

    test(s ->
      checkPositionIndex(6, s.length()), "Robert");
    test(s ->
      checkPositionIndex(6, s.length()), "Bob");
    test(s ->
      checkPositionIndex(6, s.length()), null);

    test(s -> checkPositionIndexes(
      0, 6, s.length()), "Hieronymus");
    test(s -> checkPositionIndexes(
      0, 10, s.length()), "Hieronymus");
    test(s -> checkPositionIndexes(
      0, 11, s.length()), "Hieronymus");
    test(s -> checkPositionIndexes(
      -1, 6, s.length()), "Hieronymus");
    test(s -> checkPositionIndexes(
      7, 6, s.length()), "Hieronymus");
    test(s -> checkPositionIndexes(
      0, 6, s.length()), null);
  }
}
/* Output:
X
Success
null
NullPointerException
null
NullPointerException: s was null
null
NullPointerException: s was null, arg2 arg3
Fozzie
Success
X
IllegalArgumentException
null
```

```
IllegalArgumentException
null
IllegalArgumentException: Bear Left!
null
IllegalArgumentException: Bear Left! Frog Right!
Mortimer
Success
Mort
IllegalStateException
null
NullPointerException
Robert
IndexOutOfBoundsException: index (6) must be less than
size (6)
Bob
IndexOutOfBoundsException: index (6) must be less than
size (3)
null
NullPointerException
Robert
Success
Bob
IndexOutOfBoundsException: index (6) must not be
greater than size (3)
null
NullPointerException
Hieronymus
Success
Hieronymus
Success
Hieronymus
IndexOutOfBoundsException: end index (11) must not be
greater than size (10)
Hieronymus
IndexOutOfBoundsException: start index (-1) must not be
negative
Hieronymus
IndexOutOfBoundsException: end index (6) must not be
less than start index (7)
null
NullPointerException
*/
```

Although Guava preconditions work with all types, I only demonstrate `Strings` here. The `test()` method expects a `Consumer<String>` so we can pass a lambda expression as the first argument, and the `String` to pass to the lambda as the second argument. It displays the `String` to orient you when looking at the output, then passes the `String` to the lambda expression. The second `println()` in the `try` block is only displayed if the lambda expression succeeds; otherwise the `catch` clause displays the error information. Notice how much duplicate code the `test()` method eliminates.

Each precondition has three different overloaded forms: a test with no message, a test with a simple `String` message, and a test with a `String` and a variable argument list of replacement values. For efficiency, only `%s` (`String` type) replacement tags are allowed. In the above example, the two forms of `String` message are only demonstrated for `checkNotNull()` and `checkArgument()`, but they are the same for all the rest of the precondition methods.

Note that `checkNotNull()` returns its argument, so you can use it inline in an expression. Here's how to use it in a constructor to prevent object construction containing `null` values:

```
// validating/NonNullConstruction.java
import static com.google.common.base.Preconditions.*;

public class NonNullConstruction {
  private Integer n;
  private String s;
  NonNullConstruction(Integer n, String s) {
    this.n = checkNotNull(n);
    this.s = checkNotNull(s);
  }
  public static void main(String[] args) {
    NonNullConstruction nnc =
      new NonNullConstruction(3, "Trousers");
  }
}
```

`checkArgument()` takes a `boolean` expression for a more specific test of an argument, and throws `IllegalArgumentException` upon failure. `checkState()` is for testing the state of the object (for example, an invariant check), rather than checking the arguments, and throws `IllegalStateException` upon failure.

The last three methods throw `IndexOutOfBoundsException` upon failure. `checkElementIndex()` ensures that its first argument is a valid element index into a list, string, or array with a size specified by its second argument. `checkPositionIndex()` ensures that its first argument is in the range from zero to its second argument, inclusive. `checkPositionIndexes()` checks that `[first_arg, second_arg)` is a valid subrange of a `List`, `String`, or array with the size specified by the third argument.

All Guava precondition methods have the necessary overloads for primitive types as well as `Objects`.

Test-Driven Development

The premise of *Test-Driven Development* (TDD) is that if you design and write your code with testing in mind, you not only create testable code, it will also be better-designed. In general, this seems to hold true. If I'm thinking "how will I test this?" it makes my code different, and oftentimes "testable" translates to "usable."

TDD purists write tests for a new feature before implementing that feature; this is called *Test-First Development*. To demonstrate, consider a toy example utility that inverts the case of characters in a `String`. Let's add some arbitrary constraints: the `String` must be less than or equal to 30 characters, and must contain only letters, spaces, commas and periods.

This example is different from standard TDD because it's designed to accept different implementations of the `StringInverter`, showing the evolution of the class as we satisfy the tests step-by-step. To enable this, the `StringInverter` is represented as an `interface`:

```
// validating/StringInverter.java
package validating;

interface StringInverter {
  String invert(String str);
}
```

Now we can write tests to express our requirements. The following is not typically the way you'd write your tests, but we have a special constraint here: we want to

test multiple versions of the `StringInverter` implementation. To achieve this, we exploit one of the most sophisticated new features in JUnit5: *dynamic test generation*. This is exactly what it sounds like—instead of each test being coded explicitly, you can write code that generates tests at runtime. This opens many new possibilities, especially in situations where writing a full set of tests explicitly might otherwise be prohibitive.

JUnit5 provides several ways to dynamically generate tests, but the one used here might be the most complex. The `DynamicTest.stream()` method takes:

- An iterator over the set of objects that vary from one set of tests to another. The object produced by that iterator can be of any type, but there's only a single object produced so for multiple items that vary, you must artificially package them into a single type.
- A `Function` that takes the object from the iterator and produces a `String` describing the test.
- A `Consumer` that accepts the object from the iterator and contains the test code based on that object.

In this example, all code that would otherwise be duplicated is combined in `testVersions()`. The objects that represent change and are produced by the iterator are different implementations of `DynamicTest`:

```
// validating/tests/DynamicStringInverterTests.java
package validating;
import java.util.*;
import java.util.function.*;
import java.util.stream.*;
import org.junit.jupiter.api.*;
import static org.junit.jupiter.api.Assertions.*;
import static org.junit.jupiter.api.DynamicTest.*;

class DynamicStringInverterTests {
  // Combine operations to prevent code duplication:
  Stream<DynamicTest> testVersions(String id,
    Function<StringInverter, String> test) {
    List<StringInverter> versions = Arrays.asList(
      new Inverter1(), new Inverter2(),
      new Inverter3(), new Inverter4());
    return DynamicTest.stream(
```

```
      versions.iterator(),
      inverter -> inverter.getClass().getSimpleName(),
      inverter -> {
        System.out.println(
          inverter.getClass().getSimpleName() +
            ": " + id);
        try {
          if(test.apply(inverter) != "fail")
            System.out.println("Success");
        } catch(Exception | Error e) {
          System.out.println(
            "Exception: " + e.getMessage());
        }
      }
    );
  }
  String isEqual(String lval, String rval) {
    if(lval.equals(rval))
      return "success";
    System.out.println("FAIL: " + lval + " != " + rval);
    return "fail";
  }
  @BeforeAll
  static void startMsg() {
    System.out.println(
      ">>> Starting DynamicStringInverterTests <<<");
  }
  @AfterAll
  static void endMsg() {
    System.out.println(
      ">>> Finished DynamicStringInverterTests <<<");
  }
  @TestFactory
  Stream<DynamicTest> basicInversion1() {
    String in =  "Exit, Pursued by a Bear.";
    String out = "eXIT, pURSUED BY A bEAR.";
    return testVersions(
      "Basic inversion (should succeed)",
      inverter -> isEqual(inverter.invert(in), out)
    );
  }
  @TestFactory
```

```
  Stream<DynamicTest> basicInversion2() {
    return testVersions(
      "Basic inversion (should fail)",
      inverter -> isEqual(inverter.invert("X"), "X"));
  }
  @TestFactory
  Stream<DynamicTest> disallowedCharacters() {
    String disallowed = ";-_()*&^%$#@!~`0123456789";
    return testVersions(
      "Disallowed characters",
      inverter -> {
        String result = disallowed.chars()
          .mapToObj(c -> {
            String cc = Character.toString((char)c);
            try {
              inverter.invert(cc);
              return "";
            } catch(RuntimeException e) {
              return cc;
            }
          }).collect(Collectors.joining(""));
        if(result.length() == 0)
          return "success";
        System.out.println("Bad characters: " + result);
        return "fail";
      }
    );
  }
  @TestFactory
  Stream<DynamicTest> allowedCharacters() {
    String lowcase = "abcdefghijklmnopqrstuvwxyz ,.";
    String upcase =  "ABCDEFGHIJKLMNOPQRSTUVWXYZ ,.";
    return testVersions(
      "Allowed characters (should succeed)",
      inverter -> {
        assertEquals(inverter.invert(lowcase), upcase);
        assertEquals(inverter.invert(upcase), lowcase);
        return "success";
      }
    );
  }
  @TestFactory
```

```
  Stream<DynamicTest> lengthNoGreaterThan30() {
    String str = "xxxxxxxxxxxxxxxxxxxxxxxxxxxxxx";
    assertTrue(str.length() > 30);
    return testVersions(
      "Length must be less than 31 (throws exception)",
      inverter -> inverter.invert(str)
    );
  }
  @TestFactory
  Stream<DynamicTest> lengthLessThan31() {
    String str = "xxxxxxxxxxxxxxxxxxxxxxxxxxxxx";
    assertTrue(str.length() < 31);
    return testVersions(
      "Length must be less than 31 (should succeed)",
      inverter -> inverter.invert(str)
    );
  }
}
```

In ordinary testing, you'd want a failed test to stop the build. Here, however, we only want the system to report the issue, but then still continue so you can see the effects of the different versions of `StringInverter`.

Each method annotated with `@TestFactory` produces a `Stream` of `DynamicTest` objects (via `testVersions()`), each of which JUnit executes just like regular `@Test` methods.

Now that the tests are in place, we can begin implementing the `StringInverter`. We start with a dummy class that returns its argument:

```
// validating/Inverter1.java
package validating;

public class Inverter1 implements StringInverter {
  @Override
  public String invert(String str) { return str; }
}
```

Next we implement the inversion operation:

```
// validating/Inverter2.java
package validating;
import static java.lang.Character.*;

public class Inverter2 implements StringInverter {
  @Override public String invert(String str) {
    String result = "";
    for(int i = 0; i < str.length(); i++) {
      char c = str.charAt(i);
      result += isUpperCase(c) ?
                toLowerCase(c) :
                toUpperCase(c);
    }
    return result;
  }
}
```

Now add code to ensure there are no more than 30 characters:

```
// validating/Inverter3.java
package validating;
import static java.lang.Character.*;

public class Inverter3 implements StringInverter {
  @Override public String invert(String str) {
    if(str.length() > 30)
      throw new RuntimeException("argument too long!");
    String result = "";
    for(int i = 0; i < str.length(); i++) {
      char c = str.charAt(i);
      result += isUpperCase(c) ?
                toLowerCase(c) :
                toUpperCase(c);
    }
    return result;
  }
}
```

Finally, we exclude disallowed characters:

```
// validating/Inverter4.java
package validating;
import static java.lang.Character.*;

public class Inverter4 implements StringInverter {
  static final String ALLOWED =
    "abcdefghijklmnopqrstuvwxyz ,." +
    "ABCDEFGHIJKLMNOPQRSTUVWXYZ";
  @Override public String invert(String str) {
    if(str.length() > 30)
      throw new RuntimeException("argument too long!");
    String result = "";
    for(int i = 0; i < str.length(); i++) {
      char c = str.charAt(i);
      if(ALLOWED.indexOf(c) == -1)
        throw new RuntimeException(c + " Not allowed");
      result += isUpperCase(c) ?
                toLowerCase(c) :
                toUpperCase(c);
    }
    return result;
  }
}
```

You'll see from the test output that each version of `Inverter` is closer to passing all the tests. This duplicates your experience while performing test-first development.

`DynamicStringInverterTests.java` was only used to show the development of the different `StringInverter` implementations during the TDD process. Ordinarily, you just write a set of tests like the following, and modify a single `StringInverter` class until it satisfies all tests:

```
// validating/tests/StringInverterTests.java
package validating;
import java.util.*;
import java.util.stream.*;
import org.junit.jupiter.api.*;
import static org.junit.jupiter.api.Assertions.*;

public class StringInverterTests {
  StringInverter inverter = new Inverter4();
  @BeforeAll
  static void startMsg() {
    System.out.println(">>> StringInverterTests <<<");
  }
  @Test
  void basicInversion1() {
    String in = "Exit, Pursued by a Bear.";
    String out = "eXIT, pURSUED BY A bEAR.";
    assertEquals(inverter.invert(in), out);
  }
  @Test
  void basicInversion2() {
    assertThrows(Error.class, () -> {
      assertEquals(inverter.invert("X"), "X");
    });
  }
  @Test
  void disallowedCharacters() {
    String disallowed = ";-_()*&^%$#@!~`0123456789";
    String result = disallowed.chars()
      .mapToObj(c -> {
        String cc = Character.toString((char)c);
        try {
          inverter.invert(cc);
          return "";
        } catch(RuntimeException e) {
          return cc;
        }
      }).collect(Collectors.joining(""));
    assertEquals(result, disallowed);
  }
  @Test
  void allowedCharacters() {
```

```
    String lowcase = "abcdefghijklmnopqrstuvwxyz ,.";
    String upcase =  "ABCDEFGHIJKLMNOPQRSTUVWXYZ ,.";
    assertEquals(inverter.invert(lowcase), upcase);
    assertEquals(inverter.invert(upcase), lowcase);
  }
  @Test
  void lengthNoGreaterThan30() {
    String str = "xxxxxxxxxxxxxxxxxxxxxxxxxxxxxx";
    assertTrue(str.length() > 30);
    assertThrows(RuntimeException.class, () -> {
      inverter.invert(str);
    });
  }
  @Test
  void lengthLessThan31() {
    String str = "xxxxxxxxxxxxxxxxxxxxxxxxxxxxx";
    assertTrue(str.length() < 31);
    inverter.invert(str);
  }
}
```

By establishing all the desired characteristics within tests as a starting point, you can add functionality until all the tests pass. Once you're finished, you also have the tests to tell you (or anyone else) when you've broken the code in the future, while fixing bugs or adding features. TDD tends to produce better, more thoughtful tests, whereas attempting to achieve full test coverage after the fact often produces hurried or meaningless tests.

Test-Driven vs. Test-First

Although I myself have not achieved the Test-First level of consciousness, I am most intrigued by the concept of "failing test as bookmark" that comes from Test-First. When you walk away from your work for awhile, it can sometimes be challenging to get back into the groove, or even find where you left off. A failing test, however, brings you right back to where you stopped. This seems like it makes it easier to step away without worrying about losing your place.

The main problem with pure test-first programming is it assumes you know everything about the problem you're solving, up front. In my own experience, I usually

start by experimenting, and only when I've worked with the problem for awhile do I understand it well enough to write tests. Certainly, there are occasional problems that are perfectly defined before you start, but I personally don't run across such problems very often. Indeed, it might be worth coining the phrase *Test-Oriented Development* to describe the practice of writing code that tests well.

Logging

> *Logging* reports information about a running program.

In a debugged program, this can be ordinary status data showing the progress of the program (for example, an installation program may log the steps taken during installation, the directories where you stored files, startup values for the program, etc.).

Logging is also helpful during debugging. Without logging, you might try to decipher the behavior of a program by inserting `println()` statements. Some examples in this book use that very technique, and in the absence of a debugger (a topic introduced shortly), it's about all you have. However, once you decide the program is working properly, you'll probably take the `println()` statements out. Then if you run into more bugs, you may need to put them back in. It's much nicer if you include output statements that are only used when necessary.

Prior to the availability of logging packages, programmers relied on the fact that the Java compiler optimizes away code that is never called. If `debug` is a `static final boolean`, you can say:

```
if(debug) {
  System.out.println("Debug info");
}
```

Then, when `debug` is `false`, the compiler removes the code within the braces. Thus, the code has no run-time impact when it isn't used. With this approach, you can place trace code throughout your program and easily turn it on and off. One drawback to the technique, however, is you must recompile your code to turn your trace statements on and off. It's more convenient to turn on the trace without recompiling the program, via a configuration file you change to modify the logging properties.

The logging package from the standard Java distribution (`java.util.logging`) is almost universally considered a poor design. Most people choose an alternative logging package instead. The *Simple Logging Facade for Java* (SLF4J) provides a facade for multiple logging frameworks, such as `java.util.logging`, `logback` and `log4j`. SLF4J allows the end-user to plug in the desired logging framework at deployment time.

SLF4J provides a sophisticated facility to report information about your program with almost the same efficiency of the technique in the preceding example. For very simple informational logging, you can do something like this:

```
// validating/SLF4JLogging.java
import org.slf4j.*;

public class SLF4JLogging {
  private static Logger log =
    LoggerFactory.getLogger(SLF4JLogging.class);
  public static void main(String[] args) {
    log.info("hello logging");
  }
}
/* Output:
2021-01-24T08:49:38.496
[main] INFO  SLF4JLogging - hello logging
*/
```

The format and information in the output, and even whether the output is considered normal or "error," depends on the back-end package connected to SLF4J. In the above example it's connected to the `logback` library (via this book's `build.gradle` file) and appears as standard output.

If we modify `build.gradle` to instead use the logging package that comes built into the JDK as the back end, the output appears as error output, and looks like this:

```
Aug 16, 2016 5:40:31 PM InfoLogging main
INFO: hello logging
```

The logging system detects the class name and method name where the log message originated. It's not guaranteed that these names are correct, so don't rely on their accuracy.

Logging Levels

SLF4J provides multiple levels of reporting. This example shows them all, in increasing order of "seriousness":

```
// validating/SLF4JLevels.java
import org.slf4j.*;

public class SLF4JLevels {
  private static Logger log =
    LoggerFactory.getLogger(SLF4JLevels.class);
  public static void main(String[] args) {
    log.trace("Hello");
    log.debug("Logging");
    log.info("Using");
    log.warn("the SLF4J");
    log.error("Facade");
  }
}
/* Output:
2021-01-24T08:49:37.658
[main] TRACE SLF4JLevels - Hello
2021-01-24T08:49:37.661
[main] DEBUG SLF4JLevels - Logging
2021-01-24T08:49:37.661
[main] INFO  SLF4JLevels - Using
2021-01-24T08:49:37.661
[main] WARN  SLF4JLevels - the SLF4J
2021-01-24T08:49:37.661
[main] ERROR SLF4JLevels - Facade
*/
```

These different levels let you look for messages of a certain level. The level is typically set inside a separate configuration file, so you can reconfigure without recompiling. The configuration file format depends on which back-end logging implementation you are using. Here is one for `logback`, which uses XML:

```
<!-- validating/logback.xml -->
<?xml version="1.0" encoding="UTF-8"?>
<configuration>
  <appender name="STDOUT"
    class="ch.qos.logback.core.ConsoleAppender">
    <encoder>
      <pattern>
%d{yyyy-MM-dd'T'HH:mm:ss.SSS}
[%thread] %-5level %logger - %msg%n
      </pattern>
    </encoder>
  </appender>
  <root level="TRACE">
    <appender-ref ref="STDOUT" />
  </root>
</configuration>
```

Try changing the `<root level="TRACE">` line to a different level and re-run the program to see how the output changes. If you provide no `logback.xml` file you'll still get a default configuration.

This has only been the briefest introduction to SLF4J and logging in general, enough to give you the basics of logging—which will actually take you a long way. Visit the SLF4J Documentation[118] for more depth.

Debugging

> Although judicious use of `System.out` statements or logging information produces valuable insight into the behavior of a program, for difficult problems this approach becomes cumbersome and time-consuming.

You might also need to peek more deeply into the program than print statements allow. For this, you need a *debugger*.

In addition to more quickly and easily displaying information than you could produce with print statements, a debugger will also set *breakpoints* and stop the program when it reaches those breakpoints. A debugger can display the state of the

[118] http://www.slf4j.org/manual.html

program at any instant, view the values of variables, step through the program line-by-line, connect to a remotely running program, and more. Especially when you start building larger systems (where bugs can easily become buried), it pays to become familiar with debuggers.

Debugging with JDB

The *Java Debugger* (JDB) is a command-line tool that ships with the JDK. JDB is at least conceptually a descendant of the Gnu Debugger (GDB, which was inspired by the original Unix DB), in terms of the instructions for debugging and its command-line interface. JDB is useful for learning about debugging and performing simple debugging tasks, and it's helpful to know it's always available wherever the JDK is installed. However, for larger projects you'll want a graphical debugger, described later.

Suppose you've written the following program:

```
// validating/SimpleDebugging.java
// {ThrowsException}

public class SimpleDebugging {
  private static void foo1() {
    System.out.println("In foo1");
    foo2();
  }
  private static void foo2() {
    System.out.println("In foo2");
    foo3();
  }
  private static void foo3() {
    System.out.println("In foo3");
    int j = 1;
    j--;
    int i = 5 / j;
  }
  public static void main(String[] args) {
    foo1();
  }
}
/* Output:
```

```
In foo1
In foo2
In foo3
___[ Error Output ]___
Exception in thread "main"
java.lang.ArithmeticException: / by zero
        at
SimpleDebugging.foo3(SimpleDebugging.java:17)
        at
SimpleDebugging.foo2(SimpleDebugging.java:11)
        at SimpleDebugging.foo1(SimpleDebugging.java:7)
        at
SimpleDebugging.main(SimpleDebugging.java:20)
*/
```

If you look at `foo3()`, the problem is obvious: You're dividing by zero. But suppose this code is buried in a large program (as is implied here by the sequence of calls) and you don't know where to start looking for the problem. As it turns out, the exception gives enough information for you to locate the problem. But let's suppose things are more difficult than that, and you must drill into it more deeply and get more information than what an exception provides.

To run JDB, you first tell the compiler to generate debugging information by compiling `SimpleDebugging.java` with the `–g` flag. Then you start debugging the program with the command line:

```
jdb SimpleDebugging
```

This brings up JDB and gives you a command prompt. You can view the list of available JDB commands by typing `?` at the prompt.

Here's an interactive debugging trace that shows how to chase down a problem:

```
Initializing jdb ...
> catch Exception
```

The `>` indicates that JDB is waiting for a command. The command `catch Exception` sets a breakpoint any place an exception is thrown (however, the debugger will stop anyway, even if you don't explicitly give this directive—exceptions appear to be default breakpoints in JDB).

```
Deferring exception catch Exception.
It will be set after the class is loaded.
> run
```

Now the program will run until the next breakpoint, which in this case is where the exception occurs. Here's the result of the `run` command:

```
run SimpleDebugging
Set uncaught java.lang.Throwable
Set deferred uncaught java.lang.Throwable
>
VM Started: In foo1
In foo2
In foo3
Exception occurred: java.lang.ArithmeticException
(uncaught)"thread=main",
SimpleDebugging.foo3(), line=16 bci=15
16        int i = 5 / j;
```

The program runs until line 16 where the exception occurs, but JDB does not exit when it hits the exception. The debugger also displays the line of code that caused the exception. You can list the point where execution stopped in the program source using the `list` command:

```
main[1] list
12      private static void foo3() {
13        System.out.println("In foo3");
14        int j = 1;
15        j--;
16 =>     int i = 5 / j;
17      }
18      public static void main(String[] args) {
19        foo1();
20      }
21    }
/* Output:
```

The pointer ("=>") in this listing shows the current point from where the execution will resume. You *could* resume the execution with the `cont` (continue) command, but that makes JDB exit at the exception, printing the stack trace.

The `locals` command dumps the value of all local variables:

```
main[1] locals
Method arguments:
Local variables:
j = 0
```

The `wherei` command prints the stack frames pushed in the method stack of the current thread:

```
main[1] wherei
[1] SimpleDebugging.foo3 (SimpleDebugging.java:16), pc = 15
[2] SimpleDebugging.foo2 (SimpleDebugging.java:10), pc = 8
[3] SimpleDebugging.foo1 (SimpleDebugging.java:6), pc = 8
[4] SimpleDebugging.main (SimpleDebugging.java:19), pc = 0
```

Each line after `wherei` represents a method call and the point where the call returns (which is shown by the value of the *program counter* `pc`). Here the calling sequence is `main(), foo1(), foo2()`, and `foo3()`.

Because the `list` command shows where the execution stopped, you often get a good enough idea of what happened that you can fix it. The `help` command will tell you more of what you can do with `jdb`, but before you spend too much time learning it, keep in mind that command-line debuggers tend to require far too much work to get results. Use `jdb` to learn the basics of debugging, then move to a graphical debugger.

Graphical Debuggers

Using a command-line debugger like JDB can be inconvenient. It requires explicit commands to look at the state of the variables (`locals`, `dump`), list the point of execution in the source code (`list`), find out the threads in the system (`threads`), set breakpoints (`stop in`, `stop at`), etc. A graphical debugger provides these features with a few clicks, and also displays the latest details of the program being debugged without using explicit commands.

Thus, although you may get started by experimenting with JDB, you'll probably find it much more productive to learn to use a graphical debugger to quickly track down your bugs. The IBM *Eclipse*, Oracle *NetBeans* and JetBrains *IntelliJ* development environments all contain good graphical debuggers for Java.

Benchmarking

> "We should forget about small efficiencies, say about 97% of the time: Premature optimization is the root of all evil."—*Donald Knuth*

If you find yourself on the slippery slope of premature optimization, you can lose months of time if you get ambitious. Usually a simple, straightforward coding approach is good enough. If you unnecessarily optimize, you make your code needlessly complicated and hard to understand.

Benchmarking means timing pieces of code or algorithms to see which runs faster, as opposed to see Profiling and Optimizing which looks at an entire program and finds sections where that program spends most of its time.

Can't you simply time the execution of a piece of code? In a straightforward language like C, this approach actually works. In a language with a complex runtime system like Java, benchmarking becomes far more challenging. To produce reliable data, the experimental setup must control variables such as CPU frequency, power saving features, other processes running on the same machine, optimizer options, and more.

Microbenchmarking

It's tempting to write a timing utility to compare speeds of different pieces of code. It seems like this might produce some useful data.

For example, here is a simple `Timer` class that can be used two ways:

1. Create a `Timer` object, perform your operations and then call the `duration()` method on your `Timer` to produce the time elapsed in milliseconds.
2. Pass a `Runnable` to the static `duration()` method. Anything conforming to the `Runnable interface` has a functional method `run()` with a function signature that takes no arguments and returns nothing.

```
// onjava/Timer.java
package onjava;
import static java.util.concurrent.TimeUnit.*;

public class Timer {
  private long start = System.nanoTime();
  public long duration() {
    return NANOSECONDS.toMillis(
      System.nanoTime() - start);
  }
  public static long duration(Runnable test) {
    Timer timer = new Timer();
    test.run();
    return timer.duration();
  }
}
```

This is a straightforward way to time something. Can't we just run some code and see how long it takes?

There are many factors that impact your results, even to the point of producing indicators that are upside-down. Here's a seemingly-innocent example using the standard Java `Arrays` library (described more fully in Arrays):

```
// validating/BadMicroBenchmark.java
// {ExcludeFromTravisCI}
import java.util.*;
import onjava.Timer;

public class BadMicroBenchmark {
  static final int SIZE = 250_000_000;
  public static void main(String[] args) {
    try { // For machines with insufficient memory
      long[] la = new long[SIZE];
      System.out.println("setAll: " +
        Timer.duration(() ->
          Arrays.setAll(la, n -> n)));
      System.out.println("parallelSetAll: " +
        Timer.duration(() ->
          Arrays.parallelSetAll(la, n -> n)));
    } catch(OutOfMemoryError e) {
      System.out.println("Insufficient memory");
```

```
      System.exit(0);
    }
  }
}
/* Output:
Insufficient memory
*/
```

The body of `main()` is inside a `try` block because one machine[119] ran out of memory and this would stop the build.

For an array of 250 million `longs` (just short of producing an "out of memory" exception on most machines), we "compare" the performance of `Arrays.setAll()` and `Arrays.parallelSetAll()`. The parallel version attempts to use multiple processors to get the job done faster. (Although I refer to some parallel ideas in this section, these concepts are not explained in detail until the Concurrent Programming chapter). Despite that, the non-parallel version appears to run faster, although the results may vary across machines.

Each operation in *BadMicroBenchmark.java* is independent, but if your operation depends on a common resource, the parallel version can end up being much slower, as the separate processes contend for that resource:

```
// validating/BadMicroBenchmark2.java
// Relying on a common resource
import java.util.*;
import onjava.Timer;

public class BadMicroBenchmark2 {
  // SIZE reduced to make it run faster:
  static final int SIZE = 5_000_000;
  public static void main(String[] args) {
    long[] la = new long[SIZE];
    Random r = new Random();
    System.out.println("parallelSetAll: " +
      Timer.duration(() ->
        Arrays.parallelSetAll(la, n -> r.nextLong())));
    System.out.println("setAll: " +
      Timer.duration(() ->
        Arrays.setAll(la, n -> r.nextLong())));
```

[119] An off-the-shelf Mac Mini with 8Gb of memory.

```
    SplittableRandom sr = new SplittableRandom();
    System.out.println("parallelSetAll: " +
      Timer.duration(() ->
        Arrays.parallelSetAll(la, n -> sr.nextLong())));
    System.out.println("setAll: " +
      Timer.duration(() ->
        Arrays.setAll(la, n -> sr.nextLong())));
  }
}
/* Output:
parallelSetAll: 1008
setAll: 294
parallelSetAll: 78
setAll: 88
*/
```

`SplittableRandom` is designed for parallel algorithms, and it certainly does seem to run faster than plain `Random` in `parallelSetAll()`. But it still appears to take longer than the non-parallel `setAll()`, which seems unlikely (And yet might be true. We just can't tell by using bad microbenchmarking).

This only touches the microbenchmark problem. The JVM Hotspot technologies significantly affect performance. If you don't "warm up" the JVM by running your code awhile before you run the test, you can get "cold" results that don't reflect the speed after it's been running awhile (And what if your running application doesn't end up using it enough to "warm up" the JVM? You won't get the performance you expect, and might even decrease speed).

The optimizer can sometimes detect when you create something and don't use it, or when the result of running some code has no effect on the program. If it optimizes away your tests then you get bad results.

A good microbenchmarking system automatically compensates for issues like this (and many others) to produce reasonable results, but creating such a system is quite tricky and requires deep knowledge.

Introducing JMH

At this writing, the only microbenchmark system for Java that produces decent results is The *Java Microbenchmarking Harness* JMH[120]. This book's `build.gradle` automates JMH setup so you can use it easily.

It's possible to write JMH code to run it from the command line, but the recommended approach is to let the JMH system run the tests for you; the `build.gradle` file is configured to run JMH tests with a single command.

JMH attempts to make benchmarks as easy as possible. For example, we'll rewrite `BadMicroBenchmark.java` to use JMH. The only annotations necessary here are `@State` and `@Benchmark`. The remaining annotations are included to either produce more understandable output or to make the benchmarks run faster for this example (JMH benchmarks often take a long time to run):

```
// validating/jmh/JMH1.java
package validating.jmh;
import java.util.*;
import org.openjdk.jmh.annotations.*;
import java.util.concurrent.TimeUnit;

@State(Scope.Thread)
@BenchmarkMode(Mode.AverageTime)
@OutputTimeUnit(TimeUnit.MICROSECONDS)
// Increase these three for more accuracy:
@Warmup(iterations = 5)
@Measurement(iterations = 5)
@Fork(1)
public class JMH1 {
  private long[] la;
  @Setup
  public void setup() {
    la = new long[250_000_000];
  }
  @Benchmark
  public void setAll() {
    Arrays.setAll(la, n -> n);
  }
  @Benchmark
```

[120] http://openjdk.java.net/projects/code-tools/jmh/

```
  public void parallelSetAll() {
    Arrays.parallelSetAll(la, n -> n);
  }
}
```

The default number of "forks" is ten, which means each test set runs ten times. To speed things up, I've used the `@Fork` annotation to reduce this to one. I've also reduced the number of warmup iterations and measurement iterations from the default of twenty down to five using the `@Warmup` and `@Measurement` annotations. Although this reduces the overall accuracy, the results are nearly identical to those with the default values. Try commenting out the `@Warmup`, `@Measurement` and `@Fork` annotations to see the default values and whether the tests show any noticeable variation; normally you should only see the error factors go down using the longer-running tests, and not changes in the results.

Running the benchmark requires an explicit gradle command (executed from the root directory of the example code). This prevents time-consuming benchmarking from running for any other `gradlew` commands:

```
gradlew validating:jmh
```

It takes several minutes, depending on your machine (without the annotation adjustments, it takes hours). The console output displays the path to a `results.txt` file, which summarizes the results. Note that `results.txt` contains the results for all `jmh` tests in this chapter: `JMH1.java`, `JMH2.java`, and `JMH3.java`.

Because the output is in absolute time, results vary between machines and operating systems. The important factor is not the absolute times. What we're really looking for is how one algorithm compares to another; in particular, how much faster or slower it is. If you run the tests on your machine, you'll see different numbers but the same patterns.

I've tested this code on numerous machines, and while the absolute numbers vary from machine to machine, the relative values stay reasonably consistent. I only display the appropriate snippets from `results.txt`, and I edit the output to make it easier to understand, and to fit on the page. The `Mode` for all tests is displayed as `avgt` for "Average Time." The `Cnt` (number of tests) is 200, although you'll see a `Cnt` of 5 when you run the examples as configured here. The `Units` are `us/op` for "Microseconds per operation", thus smaller numbers indicate faster performance.

I'm also showing the output from the default number of warmups, measurements and forks. I deleted the corresponding annotations from the examples in order to run my tests with greater accuracy (this takes hours). The pattern of the numbers should still look the same regardless of how you run the tests.

Here are the results for `JMH1.java`:

```
Benchmark              Score
JMH1.setAll            196280.2
JMH1.parallelSetAll    195412.9
```

Even for a sophisticated benchmarking tool like JMH, the process of benchmarking is nontrivial and you must exercise care. Here, the test produces counterintuitive results: the parallel version takes about the same time as the non-parallel `setAll()`, and both seem to take a rather long time.

My assumption when I created the example was that, if we are testing array initialization, it makes sense to use very large arrays. So I chose the largest array I could; if you experiment you'll see that when you start making the array larger than 250 million[121] you begin to get out-of-memory exceptions. However, it's possible that performing mass operations on an array this large is thrashing the memory system and producing these unexpected results. Whether or not that's a correct hypothesis, it *does* seem that we're not actually testing what we think we're testing.

Consider other factors:

- C: The number of client threads performing operations.
- P: The amount of parallelism used by a parallel algorithm.
- N: The size of the array: `10^(2*k)`, where `k=1..7` is usually enough to exercise different cache footprints.
- Q: The cost of the setter operation.

This C/P/N/Q model surfaced during early JDK 8 Lambda development, and most parallel `Stream` operations (`parallelSetAll()` is quite similar) agree with these conclusions:

- `N*Q` (basically, the amount of work) is critical to parallel performance. With less work, the parallel algorithm may actually run slower.

[121] For limited machine configurations this number can be much lower.

- There are cases where operations are so contended that parallelism is no help at all, no matter how large `N*Q` is.
- When `C` is high, `P` is much less relevant (an abundance of external parallelism makes internal parallelism redundant). Moreover, in some cases, the cost of parallel decomposition makes `C` clients running a parallel algorithm run slower than the same `C` clients running sequential code.

Based on this information, we re-run the test using different array sizes (values of `N`):

```
// validating/jmh/JMH2.java
package validating.jmh;
import java.util.*;
import org.openjdk.jmh.annotations.*;
import java.util.concurrent.TimeUnit;

@State(Scope.Thread)
@BenchmarkMode(Mode.AverageTime)
@OutputTimeUnit(TimeUnit.MICROSECONDS)
@Warmup(iterations = 5)
@Measurement(iterations = 5)
@Fork(1)
public class JMH2 {
  private long[] la;
  @Param({
    "1",
    "10",
    "100",
    "1000",
    "10000",
    "100000",
    "1000000",
    "10000000",
    "100000000",
    "250000000"
  })
  int size;

  @Setup
  public void setup() {
    la = new long[size];
```

```
  }
  @Benchmark
  public void setAll() {
    Arrays.setAll(la, n -> n);
  }
  @Benchmark
  public void parallelSetAll() {
    Arrays.parallelSetAll(la, n -> n);
  }
}
```

`@Param` automatically inserts each of its values into the variable it annotates. The values must be `Strings` and are converted to the appropriate type, `int` in this case.

Here are the edited results along with a calculated speedup:

```
JMH2 Benchmark         Size    Score       % Speedup
setAll                    1    0.001
parallelSetAll            1    0.036       0.028
setAll                   10    0.005
parallelSetAll           10    3.965       0.001
setAll                  100    0.031
parallelSetAll          100    3.145       0.010
setAll                 1000    0.302
parallelSetAll         1000    3.285       0.092
setAll                10000    3.152
parallelSetAll        10000    9.669       0.326
setAll               100000    34.971
parallelSetAll       100000    20.153      1.735
setAll              1000000    420.581
parallelSetAll      1000000    165.388     2.543
setAll             10000000    8160.054
parallelSetAll     10000000    7610.190    1.072
setAll            100000000    79128.752
parallelSetAll    100000000    76734.671   1.031
setAll            250000000    199552.121
parallelSetAll    250000000    191791.927  1.040
```

Around 100,000 elements, `parallelSetAll()` starts to pull ahead, but then drops back to about par. Even when it's winning, it doesn't seem to be enough of an improvement to justify its existence.

Does the amount of work in the calculation performed by `setAll()`/`parallelSetAll()` make a difference? In the previous examples, all we did was assign the value of the index into the array location, which is one of the simplest tasks possible. So even when the `N` becomes large, `N*Q` still isn't that great, so it looks like we aren't providing enough opportunity for parallelism. (JMH provides a way to simulate a variable Q; to learn more, search for `Blackhole.consumeCPU`.)

By making the task more complex using the following method `f()`, we produce more parallel opportunities:

```
// validating/jmh/JMH3.java
package validating.jmh;
import java.util.*;
import org.openjdk.jmh.annotations.*;
import java.util.concurrent.TimeUnit;

@State(Scope.Thread)
@BenchmarkMode(Mode.AverageTime)
@OutputTimeUnit(TimeUnit.MICROSECONDS)
@Warmup(iterations = 5)
@Measurement(iterations = 5)
@Fork(1)
public class JMH3 {
  private long[] la;
  @Param({
    "1",
    "10",
    "100",
    "1000",
    "10000",
    "100000",
    "1000000",
    "10000000",
    "100000000",
    "250000000"
  })
  int size;

  @Setup
  public void setup() {
    la = new long[size];
  }
```

```
  public static long f(long x) {
    long quadratic = 42 * x * x + 19 * x + 47;
    return Long.divideUnsigned(quadratic, x + 1);
  }
  @Benchmark
  public void setAll() {
    Arrays.setAll(la, n -> f(n));
  }
  @Benchmark
  public void parallelSetAll() {
    Arrays.parallelSetAll(la, n -> f(n));
  }
}
```

`f()` provides a more complex and time-consuming operation. Now, instead of simply assigning the index into its corresponding location, both `setAll()` and `parallelSetAll()` have more work to do, and this definitely affects the results:

```
JMH3 Benchmark         Size    Score        % Speedup
setAll                    1    0.012
parallelSetAll            1    0.047        0.255
setAll                   10    0.107
parallelSetAll           10    3.894        0.027
setAll                  100    0.990
parallelSetAll          100    3.708        0.267
setAll                 1000    133.814
parallelSetAll         1000    11.747       11.391
setAll                10000    97.954
parallelSetAll        10000    37.259       2.629
setAll               100000    988.475
parallelSetAll       100000    276.264      3.578
setAll              1000000    9203.103
parallelSetAll      1000000    2826.974     3.255
setAll             10000000    92144.951
parallelSetAll     10000000    28126.202    3.276
setAll            100000000    921701.863
parallelSetAll    100000000    266750.543   3.455
setAll            250000000    2299127.273
parallelSetAll    250000000    538173.425   4.272
```

You can see that somewhere around an array size of 1000, `parallelSetAll()` pulls ahead of `setAll()`. It appears that `parallelSetAll()` depends heavily on the

complexity of the calculation combined with the size of the array. This is exactly the value of benchmarking, because we've learned subtle things about how `setAll()` and `parallelSetAll()` work and when to use them. This is not obvious from studying the Javadocs.

Much of the time, simple applications of JMH will produce good results (as you shall see in examples later in the book), but we've learned here that you can't assume this is always the case. The JMH site has samples[122] to help you get started.

Profiling and Optimizing

Sometimes you must detect where your program spends all its time, to see whether you can improve the performance of those sections. A *profiler* finds the slow spots so you can look for the easiest, most obvious way to speed things up.

A profiler gathers information showing which parts of the program consume memory and which methods consume maximum time. Some profilers even disable the garbage collector to help determine patterns of memory allocation.

A profiler is also useful for detecting thread deadlock in your program.

Note the difference between profiling and benchmarking. Profiling looks at your completed program working on your actual data, whereas benchmarking looks at an isolated fragment of a program, typically to optimize an algorithm.

The Java Development Kit (JDK) installation comes with a visual profiler called **VisualVM**. This is automatically installed in the same directory as `javac`, which you should already have in your execution path. To start **VisualVM**, the console command is:

```
› jvisualvm
```

This command opens a window containing links to help information.

Optimization Guidelines

- Avoid sacrificing code readability for performance.

[122] http://hg.openjdk.java.net/code-tools/jmh/file/tip/jmh-samples/src/main/java/org/openjdk/jmh/samples/

- Don't look at performance in isolation. Weigh the amount of effort required versus the advantage gained.
- The size of the program matters. Performance optimization is generally valuable only for large projects that run for a long time. Performance is often not a concern for small projects.
- Making the program work is a higher priority than delving into its performance. Once you have a working program you can, if necessary, use the profiler to make it more efficient. Consider performance during the initial design/development process only if it is a critical factor.
- Do not guess where the bottlenecks are. Run a profiler to get that data.
- Whenever possible, explicitly discard an instance by setting it to null. This can be a useful hint to the garbage collector.
- `static final` variables can be optimized by the JVM to improve program speed. Program constants should thus be declared `static` and `final`.

Style Checking

When you're working on a project in a team (including and especially open-source projects), it's very helpful if everyone follows the same coding style. This way, reading code for the project doesn't produce mental interrupts due to style differences. However, if you're used to coding a different style it can be difficult to remember all the style guidelines for a particular project. Fortunately, tools exist to point out places in your code that don't follow your guidelines.

A popular style checker is Checkstyle[123]. Look at the `gradle.build` and `checkstyle.xml` files in the book's Example Code[124] to see one way to configure Checkstyle. `checkstyle.xml` is a set of common checks, with some of those checks commented out to allow for the style used in this book.

To run all the style checks, the command is:

```
gradlew checkstyleMain
```

[123] http://checkstyle.sourceforge.net/
[124] https://github.com/BruceEckel/OnJava8-Examples

Some files still produce Checkstyle warnings, typically because those examples demonstrate something you don't normally do in production code.

You can also run the style checks for a specific chapter. For example, here's how to check the Annotations chapter:

```
gradlew annotations:checkstyleMain
```

Static Error Analysis

Although Java's static type checking finds basic syntax errors, additional analysis tools can find more sophisticated bugs that elude `javac`. One such tool is Findbugs[125]. This book's `build.gradle` file in the Example Code[126] contains a configuration for Findbugs so you can enter the command:

```
gradlew findbugsMain
```

This produces an HTML report for each chapter, called `main.html`, showing potential issues in the code. The Gradle command output shows you where each report resides.

When you look at the reports you'll see there are many *false positives* suggesting there's a problem when the code is actually fine. Some of the positives are indeed correct for files where I was demonstrating how *not* to do something.

When I initially reviewed the Findbugs output for the book, I found a few things that weren't technically errors but enabled me to improve the code. If you're hunting for a bug, it's worth running Findbugs before starting up your debugger, as it might quickly find something that could otherwise require hours.

Code Reviews

Unit Testing finds significant and important classes of bugs. Checkstyle and Findbugs perform automatic code reviews to find additional problems. Eventually you reach

[125] http://findbugs.sourceforge.net/
[126] https://github.com/BruceEckel/OnJava8-Examples

the point where you must add human eyeballs to the mix. Code reviews are various ways of taking a piece of code written by one person or group and having it read and evaluated by someone else or a group. Initially this can seem intimidating, and it does require emotional trust, but the goal is definitely not to shame or ridicule anyone. The objective is to find program errors, and code reviews are one of the most successful ways to do this. Alas, they are also usually considered "too expensive" (and sometimes that excuse can be a way for programmers to avoid the perceived embarrassment of a review).

Code reviews can be performed as part of *pair programming*, as part of the code check-in process (another programmer is automatically assigned the task of reviewing new code), or with a group, using a *walkthrough*, where everyone reads the code and discusses it. This latter approach has the significant benefit of sharing knowledge and coding culture.

Pair Programming

Pair programming is the practice of two programmers coding together. Typically, one programmer "drives" (sits at the keyboard and types) and the other (the "observer" or "navigator") reviews and analyzes the code, and also considers strategies. This produces a kind of real-time code review. Normally programmers regularly switch roles.

Pair programming has numerous benefits, but the two most compelling ones are sharing knowledge and preventing blockage. One of the best ways to transfer information is by solving problems together, and I have used pair programming in numerous seminars to great effect (also, people in the seminar get to know each other this way). And with two people working together, it's far easier to keep moving forward whereas a single programmer can easily get stuck.

Pair programmers generally report higher satisfaction in their work.

Pair programming can sometimes be a difficult sell to managers who might immediately observe that two programmers working on one problem are less productive than if they are each working on their own projects. While this is often true in the short term, the code produced is of higher quality; along with the other benefits of pair programming this produces greater productivity if you consider a longer view.

Wikipedia's pair programming article[127] is a good start if you want more information.

Refactoring

Technical debt is all those quick-and-dirty solutions that accumulate in your software and make the design impossible to understand and the code unreadable. This is particularly problematic when you must make changes and add features.

Refactoring is the antidote to technical debt. The key to refactoring is that it improves the code design, structure and readability (thus reducing technical debt), but it *doesn't change the behavior of the code.*

This can be a hard sell to management: "We're going to put in a lot of work but we won't be adding any features and from the outside there will be no noticeable changes when we're done. But trust us, things will be a lot better." Unfortunately, management only realizes the value of refactoring at the moment it's too late: when they ask for "just one more feature" and you have to tell them it's impossible because the code base has become a huge accumulation of hacks and it will collapse if you try to add another feature, even if you *could* figure out how to do it.

Foundations For Refactoring

Before you can start refactoring your code, you must have three support systems in place:

1. Tests (typically, JUnit tests as a minimum foundation), so you can ensure your refactorings do not change the behavior of the code.
2. Build automation, so you can easily build the code and run all the tests. This way it's effortless to make a small change and verify it hasn't broken anything. This book uses the Gradle build system, and you can find the `build.gradle` file in the download[128] that contains the examples.
3. Version control, so you can always provide/go back to a working version of the code, and keep track of the steps along the way. This book's example code is hosted on Github[129] and uses the `git` version control system.

[127] https://en.wikipedia.org/wiki/Pair_programming
[128] https://github.com/BruceEckel/OnJava8-Examples
[129] https://github.com/BruceEckel/OnJava8-Examples

Without these three systems, refactoring is virtually impossible. Indeed, without these, building, maintaining and adding to code in the first place is a big challenge. Surprisingly, there are many successful companies that manage to get by for quite awhile without using any of these three systems. However, with such companies it's usually just a matter of time before they run into serious problems.

The Wikipedia article on refactoring[130] provides further details.

Continuous Integration

In the early days of software development, people could only manage one step at a time, so they took on the belief that they were always traveling "the happy path," and each development stage was going to flow seamlessly into the next. This delusion was often called "The Waterfall Model" of software development. I've had people tell me that Waterfall was their method of choice, as if it were actually a selected tool and not just wishful thinking.

In this fairy-tale land, each step finished perfectly and on time according to the made-up schedule, and then the next step could start. By the time you reached the end, all the pieces would slide seamlessly together and voila! A shipping product!

In reality, of course, nothing ever went to plan or to schedule. Believing it should, and then believing harder when it didn't, just made the whole thing worse. Denying evidence doesn't produce good results.

On top of all this, the product itself often wasn't something valuable for customers. Sometimes a whole raft of features were a complete waste of time, because the need for those features was invented by someone other than the customer.

Because it came from assembly-line mentality, each development stage had its own team. Schedule slips from an upstream team were passed to a downstream team, and by the time you got to testing and integration, those teams were expected to somehow catch up to the schedule, and when they inevitably couldn't they were considered "bad team players." The combination of impossible schedules and negative associations created a self-fulfilling prophecy: only the most desperate of developers were willing to do those jobs.

[130] https://en.wikipedia.org/wiki/Code_refactoring

On top of this, business schools continued to produce managers who were only trained to turn the crank on existing processes—processes based on the ideas of industrial-age manufacturing. Business schools that teach creativity rather than conformity continue to be very rare.

Eventually people from within the programming ranks couldn't stand it anymore and started experimenting. Some of these initial experiments were called "extreme programming" because they were very different than industrial-age thinking. As the experiments showed results the ideas began to look like common sense. These experiments evolved the now-obvious idea of putting a working product—albeit very minimal—into the hands of customers and asking them if (A) they even wanted it (B) if they liked the way it worked and (C) what other features they might find useful. Then it's back to development to produce a new version. Version by version, the project evolves into something that truly produces value for the customer.

This is completely upside-down from the waterfall approach. You stop assuming that you handle things like product testing and deployment "as the final step." Instead, everything from beginning to end—even for a product that has virtually no features at the outset (in which case you're probably just testing the installation)—must be in play. Doing so has the huge benefit of finding even more problems early in the development cycle. In addition, rather than doing a lot of big, up-front planning and spending time and money on a lot of useless features, you're in a constant feedback cycle with the customer. When the customer doesn't want any more features, you're done. This saves lots of time and money and produces lots of customer satisfaction.

There have been many different pieces and ideas leading up to this approach, but the current over-arching term is *continuous integration* (CI). The difference between CI and the ideas leading up to it is that CI is a distinct mechanical process that encompasses those ideas; it is a well-defined way of doing things. So well-defined, in fact, that the whole process is automated.

The current apex of CI technology is the *continuous integration server*. This is a separate machine or virtual machine, and is commonly a completely separate service hosted by a third-party company. They typically provide a basic service for free, and if you need additional features like more processors or memory or specialized tools or systems, you pay for those. The CI server starts out as a completely blank slate with only the most minimal operating system available. This is important because it's easy to forget if you have already installed some piece on your development

machine, and then not include that piece in your build and deployment systems.

Just like refactoring, *continuous integration* requires a foundation of distributed version control, build automation, and automated testing. The CI server is typically tied into your version control repository. When the CI server sees a change in the repository, it checks out the latest version and begins the process specified in your CI script. This includes installing all necessary tools and libraries (remember, the CI server starts with nothing but a clean, basic operating system), so if there are any problems in that process you discover them. Then it performs any builds and tests specified in your script; the script typically uses exactly the same commands that a human will use in the installation and testing process. If it succeeds or fails the CI server has numerous ways to report that to you, including a simple badge that appears on your code repository.

Using Continuous Integration, every change you check into the repository is automatically validated from beginning to end. This way you find out immediately if you begin to have problems. Even better, when you are ready to ship a new version of the product there's no delay or any extra steps necessary (being able to deliver at any time is called *Continuous Delivery*).

This book's example code is automatically tested on Travis-CI[131] (for Linux-based systems) and AppVeyor[132] (for Windows). You can see the pass/fail badges on the Readme at the Github repository[133].

Summary

> "It works on my machine." "We are not shipping your machine!"

Code validation is not a single process or technique. Any one approach only finds certain categories of bugs, and as you develop as a programmer you learn that every additional technique adds to the reliability and robustness of your code. Validation helps you discover more errors not only during development but also during the project lifetime, as you add capabilities to your application. Modern development means much more than just writing code, and every testing technique you fold

[131] https://travis-ci.org/BruceEckel/OnJava8-Examples
[132] https://ci.appveyor.com/project/BruceEckel/onjava-examples
[133] https://github.com/BruceEckel/OnJava8-Examples

into your development process—including and especially custom tools you create to fit a particular application—results in better, faster, more pleasant development and improved value and a more satisfying experience for your customer.

Files

> After many years of awkward file I/O programming, Java finally simplified the basic act of reading and writing files.

The full details of "the hard way" are in the Appendix: I/O Streams. If you read that, you might come to the conclusion that the Java designers didn't care about the experience of their users. Opening and reading a file—which is a rather common experience with most programming languages—required such awkward code that no one could remember how to open a file without looking up the code every single time.

Java 7 introduced great improvements, as if the designers had finally heard the years of pain from their users. These new elements are packaged under `java.nio.file`, where the `n` in `nio` formerly meant "new" but now means "non-blocking" (`io` is for *input/output*). The `java.nio.file` library finally brings Java file manipulation into the same arena as other programming languages. On top of that, Java 8 adds streams to the mix, which makes file programming even nicer.

We shall look at the two basic components to manipulating files:

1. The path to the file or directory.
2. The file itself.

File and Directory Paths

A `Path` object represents a path to a file or a directory, abstracted across operating systems (OSes) and file systems. The intent is that you don't have to pay attention to the underlying OS when constructing a path, and your code will work on different OSes without rewriting.

The `java.nio.file.Paths` class contains a single overloaded `static get()` method to take either a sequence of `Strings` or a *Uniform Resource Identifier* (URI) and convert it to a `Path` object:

```
// files/PathInfo.java
import java.nio.file.*;
import java.net.URI;
import java.io.File;
import java.io.IOException;

public class PathInfo {
  static void show(String id, Object p) {
    System.out.println(id + ": " + p);
  }
  static void info(Path p) {
    show("toString", p);
    show("Exists", Files.exists(p));
    show("RegularFile", Files.isRegularFile(p));
    show("Directory", Files.isDirectory(p));
    show("Absolute", p.isAbsolute());
    show("FileName", p.getFileName());
    show("Parent", p.getParent());
    show("Root", p.getRoot());
    System.out.println("********************");
  }
  public static void main(String[] args) {
    System.out.println(System.getProperty("os.name"));
    info(Paths.get(
      "C:", "path", "to", "nowhere", "NoFile.txt"));
    Path p = Paths.get("PathInfo.java");
    info(p);
    Path ap = p.toAbsolutePath();
    info(ap);
    info(ap.getParent());
    try {
      info(p.toRealPath());
    } catch(IOException e) {
      System.out.println(e);
    }
    URI u = p.toUri();
    System.out.println("URI: " + u);
    Path puri = Paths.get(u);
    System.out.println(Files.exists(puri));
    File f = ap.toFile(); // Don't be fooled
  }
}
```

```
/* Output:
Windows 8.1
toString: C:\path\to\nowhere\NoFile.txt
Exists: false
RegularFile: false
Directory: false
Absolute: true
FileName: NoFile.txt
Parent: C:\path\to\nowhere
Root: C:\
******************
toString: PathInfo.java
Exists: true
RegularFile: true
Directory: false
Absolute: false
FileName: PathInfo.java
Parent: null
Root: null
******************
toString:
C:\Git\OnJava8\ExtractedExamples\files\PathInfo.java
Exists: true
RegularFile: true
Directory: false
Absolute: true
FileName: PathInfo.java
Parent: C:\Git\OnJava8\ExtractedExamples\files
Root: C:\
******************
toString: C:\Git\OnJava8\ExtractedExamples\files
Exists: true
RegularFile: false
Directory: true
Absolute: true
FileName: files
Parent: C:\Git\OnJava8\ExtractedExamples
Root: C:\
******************
toString:
C:\Git\OnJava8\ExtractedExamples\files\PathInfo.java
Exists: true
```

```
RegularFile: true
Directory: false
Absolute: true
FileName: PathInfo.java
Parent: C:\Git\OnJava8\ExtractedExamples\files
Root: C:\
******************
URI: file:///C:/Git/OnJava8/ExtractedExamples/files/Pat
hInfo.java
true
*/
```

I've added the first line of `main()` to appropriate programs in this chapter to display the OS name, so you can see what differences there are between one OS and another. Ideally the differences are relatively few and isolated to expected places, such as whether `/` or `\` is the path separator. You can see from the output I'm developing on Windows 10.

While `toString()` produces a full representation of the path, you can see that `getFileName()` always produces the name of the file. Using the `Files` utility class (which we'll see more of) you can test to see whether a file exists, is a "regular" file, is a directory, and more. The "Nofile.txt" example shows it is possible to describe a file which isn't there; this allows you to create a new path. "PathInfo.java" lives in the current directory, and initially it's just the file name without a path, although it still checks out as "existing." Once we convert it to an absolute path, we get the full path starting from the "C:" drive (this was tested on a Windows machine). Now it has a parent, as well.

The definition of the "real" path is a bit vague in the documentation because it depends on the particular file system. If a filename comparison is case-insensitive, for example, you might get a positive match even if the path doesn't look exactly the same because of casing. On such a platform, `toRealPath()` will return the `Path` with the actual case. It also removes any redundant elements.

Here, you see what a URI looks like for a file, but it can be used to describe most things; see Wikipedia[134] for a detailed description. Then we successfully convert the URI back into a `Path`.

[134] https://en.wikipedia.org/wiki/Uniform_Resource_Identifier

Finally, you see something slightly deceiving, which is the call to `toFile()` to produce a `File` object. This sounds like you might get something file-like (it's called `File`, after all) but this method exists for backwards-compatibility to the old-style way of doing things. In that world, `File` actually means either a file or a directory—which sounds like it should have been called "path" instead. Very sloppy and confusing naming, but you can safely ignore it now that `java.nio.file` exists.

Selecting Pieces of a `Path`

`Path` objects can easily yield parts of their path:

```
// files/PartsOfPaths.java
import java.nio.file.*;

public class PartsOfPaths {
  public static void main(String[] args) {
    System.out.println(System.getProperty("os.name"));
    Path p =
      Paths.get("PartsOfPaths.java").toAbsolutePath();
    for(int i = 0; i < p.getNameCount(); i++)
      System.out.println(p.getName(i));
    System.out.println("ends with '.java': " +
      p.endsWith(".java"));
    for(Path pp : p) {
      System.out.print(pp + ": ");
      System.out.print(p.startsWith(pp) + " : ");
      System.out.println(p.endsWith(pp));
    }
    System.out.println("Starts with " + p.getRoot() +
      " " + p.startsWith(p.getRoot()));
  }
}
/* Output:
Windows 8.1
Git
OnJava8
ExtractedExamples
files
PartsOfPaths.java
ends with '.java': false
```

```
Git: false : false
OnJava8: false : false
ExtractedExamples: false : false
files: false : false
PartsOfPaths.java: false : true
Starts with C:\ true
*/
```

You can index into the parts of a `Path` using `getName()`, respecting the upper bound with `getNameCount()`. A `Path` also produces an `Iterator` so you can step through using for-in. Note that, even though my path here *does* end with `.java`, `endsWith()` produces `false`. This is because `endsWith()` is comparing the entire path component, not a substring within the name. This is shown within the for-in body by checking the current piece of the path using `startsWith()` and `endsWith()`. However, we see that iterating through the `Path` does *not* include the root, and only when we check `startsWith()` against the root does it produce `true`.

Analyzing a `Path`

The `Files` utility class contains a full set of tests to discover information about a `Path`:

```
// files/PathAnalysis.java
import java.nio.file.*;
import java.io.IOException;

public class PathAnalysis {
  static void say(String id, Object result) {
    System.out.print(id + ": ");
    System.out.println(result);
  }
  public static void
  main(String[] args) throws IOException {
    System.out.println(System.getProperty("os.name"));
    Path p =
      Paths.get("PathAnalysis.java").toAbsolutePath();
    say("Exists", Files.exists(p));
    say("Directory", Files.isDirectory(p));
    say("Executable", Files.isExecutable(p));
    say("Readable", Files.isReadable(p));
```

```
    say("RegularFile", Files.isRegularFile(p));
    say("Writable", Files.isWritable(p));
    say("notExists", Files.notExists(p));
    say("Hidden", Files.isHidden(p));
    say("size", Files.size(p));
    say("FileStore", Files.getFileStore(p));
    say("LastModified: ", Files.getLastModifiedTime(p));
    say("Owner", Files.getOwner(p));
    say("ContentType", Files.probeContentType(p));
    say("SymbolicLink", Files.isSymbolicLink(p));
    if(Files.isSymbolicLink(p))
      say("SymbolicLink", Files.readSymbolicLink(p));
    if(FileSystems.getDefault()
       .supportedFileAttributeViews().contains("posix"))
      say("PosixFilePermissions",
        Files.getPosixFilePermissions(p));
  }
}
/* Output:
Windows 8.1
Exists: true
Directory: false
Executable: true
Readable: true
RegularFile: true
Writable: true
notExists: false
Hidden: false
size: 1617
FileStore: (C:)
LastModified: : 2021-01-24T15:48:37.461504Z
Owner: GROOT\Bruce (User)
ContentType: null
SymbolicLink: false
*/
```

For the final test, I had to figure out whether the file system supported Posix before calling `getPosixFilePermissions()`, otherwise it produces a runtime exception.

Adding and Subtracting `Path`s

We must be able to construct `Path` objects by adding and subtracting pieces to our `Path`. To subtract the base of a `Path` we use `relativize()` and to add pieces at the end of a `Path` we use `resolve()` (not exactly "discoverable" names).

For this example I use `relativize()` to remove the `base` path from all the output, partly as a demonstration and partly to simplify the output. It turns out you can only `relativize()` a `Path` if it is absolute.

This version of show includes `id` numbers to make it easier to track the output:

```
// files/AddAndSubtractPaths.java
import java.nio.file.*;
import java.io.IOException;

public class AddAndSubtractPaths {
  static Path base = Paths.get("..", "..", "..")
    .toAbsolutePath()
    .normalize();
  static void show(int id, Path result) {
    if(result.isAbsolute())
      System.out.println("(" + id + ")r " +
        base.relativize(result));
    else
      System.out.println("(" + id + ")  " + result);
    try {
      System.out.println("RealPath: "
        + result.toRealPath());
    } catch(IOException e) {
      System.out.println(e);
    }
  }
  public static void main(String[] args) {
    System.out.println(System.getProperty("os.name"));
    System.out.println(base);
    Path p = Paths.get("AddAndSubtractPaths.java")
      .toAbsolutePath();
    show(1, p);
    Path convoluted = p.getParent().getParent()
      .resolve("strings")
      .resolve("..")
```

```
      .resolve(p.getParent().getFileName());
    show(2, convoluted);
    show(3, convoluted.normalize());

    Path p2 = Paths.get("..", "..");
    show(4, p2);
    show(5, p2.normalize());
    show(6, p2.toAbsolutePath().normalize());

    Path p3 = Paths.get(".").toAbsolutePath();
    Path p4 = p3.resolve(p2);
    show(7, p4);
    show(8, p4.normalize());

    Path p5 = Paths.get("").toAbsolutePath();
    show(9, p5);
    show(10, p5.resolveSibling("strings"));
    show(11, Paths.get("nonexistent"));
  }
}
/* Output:
Windows 8.1
C:\Git
(1)r OnJava8\ExtractedExamples\files\AddAndSubtractPath
s.java
RealPath: C:\Git\OnJava8\ExtractedExamples\files\AddAnd
SubtractPaths.java
(2)r OnJava8\ExtractedExamples\strings\..\files
RealPath: C:\Git\OnJava8\ExtractedExamples\files
(3)r OnJava8\ExtractedExamples\files
RealPath: C:\Git\OnJava8\ExtractedExamples\files
(4)  ..\..
RealPath: C:\Git\OnJava8
(5)  ..\..
RealPath: C:\Git\OnJava8
(6)r OnJava8
RealPath: C:\Git\OnJava8
(7)r OnJava8\ExtractedExamples\files\.\..\..
RealPath: C:\Git\OnJava8
(8)r OnJava8
RealPath: C:\Git\OnJava8
(9)r OnJava8\ExtractedExamples\files
```

```
RealPath: C:\Git\OnJava8\ExtractedExamples\files
(10)r OnJava8\ExtractedExamples\strings
RealPath: C:\Git\OnJava8\ExtractedExamples\strings
(11)  nonexistent
java.nio.file.NoSuchFileException:
C:\Git\OnJava8\ExtractedExamples\files\nonexistent
*/
```

I've also added further tests for `toRealPath()`. This always expands and normalizes the `Path`, except when the path doesn't exist, in which case it throws an exception.

Directories

The `Files` utility class contains most of the operations you'll need for directory and file manipulation. For some reason, however, they did not include a tool to delete directory trees, so we'll create one and add it to the `onjava` library.

```
// onjava/RmDir.java
package onjava;
import java.nio.file.*;
import java.nio.file.attribute.BasicFileAttributes;
import java.io.IOException;

public class RmDir {
  public static void rmdir(Path dir)
  throws IOException {
    Files.walkFileTree(dir,
      new SimpleFileVisitor<Path>() {
      @Override public FileVisitResult
      visitFile(Path file, BasicFileAttributes attrs)
      throws IOException {
        Files.delete(file);
        return FileVisitResult.CONTINUE;
      }
      @Override public FileVisitResult
      postVisitDirectory(Path dir, IOException exc)
      throws IOException {
        Files.delete(dir);
        return FileVisitResult.CONTINUE;
      }
```

```
    });
  }
}
```

This relies on `Files.walkFileTree()`; "walking" the tree means looking at every subdirectory and file. The *Visitor* design pattern provides a standard mechanism to visit every object in a collection, then you provide the action you want executed on each of those objects. This action is defined by how you implement the `FileVisitor` argument, which contains:

- `preVisitDirectory()`: Runs on a directory before entries in the directory are visited.
- `visitFile()`: Runs on each file in the directory.
- `visitFileFailed()`: Called for a file that cannot be visited.
- `postVisitDirectory()`: Runs on a directory after entries in the directory—including all the subdirectories beneath it—are visited.

To make things simpler, `java.nio.file.SimpleFileVisitor` provides default definitions for all methods. That way, in our anonymous inner class, we only override the methods with nonstandard behavior: `visitFile()` deletes the file, and `postVisitDirectory()` deletes the directory. Both return flags indicate that the walk should continue (this way you can walk only until you find what you're looking for).

Now we can conditionally delete an existing directory, as part of our exploration of creating and populating directories. In the following example, `makeVariant()` takes a base directory `test` and produces different subdirectory paths by rotating through the `parts` list. These rotations are pasted together with the path separator `sep` using `String.join()`, then the result is returned as a `Path`.

```
// files/Directories.java
import java.util.*;
import java.nio.file.*;
import onjava.RmDir;

public class Directories {
  static Path test = Paths.get("test");
  static String sep =
    FileSystems.getDefault().getSeparator();
  static List<String> parts =
    Arrays.asList("foo", "bar", "baz", "bag");
  static Path makeVariant() {
    Collections.rotate(parts, 1);
    return Paths.get("test", String.join(sep, parts));
  }
  static void refreshTestDir() throws Exception {
    if(Files.exists(test))
      RmDir.rmdir(test);
    if(!Files.exists(test))
      Files.createDirectory(test);
  }
  public static void
  main(String[] args) throws Exception {
    refreshTestDir();
    Files.createFile(test.resolve("Hello.txt"));
    Path variant = makeVariant();
    // Throws exception (too many levels):
    try {
      Files.createDirectory(variant);
    } catch(Exception e) {
      System.out.println("Nope, that doesn't work.");
    }
    populateTestDir();
    Path tempdir =
      Files.createTempDirectory(test, "DIR_");
    Files.createTempFile(tempdir, "pre", ".non");
    Files.newDirectoryStream(test)
      .forEach(System.out::println);
    System.out.println("*********");
    Files.walk(test).forEach(System.out::println);
  }
  static void populateTestDir() throws Exception {
```

```
    for(int i = 0; i < parts.size(); i++) {
      Path variant = makeVariant();
      if(!Files.exists(variant)) {
        Files.createDirectories(variant);
        Files.copy(Paths.get("Directories.java"),
          variant.resolve("File.txt"));
        Files.createTempFile(variant, null, null);
      }
    }
  }
}
/* Output:
Nope, that doesn't work.
test\bag
test\bar
test\baz
test\DIR_8683707748599240459
test\foo
test\Hello.txt
*********
test
test\bag
test\bag\foo
test\bag\foo\bar
test\bag\foo\bar\baz
test\bag\foo\bar\baz\4316127347949967230.tmp
test\bag\foo\bar\baz\File.txt
test\bar
test\bar\baz
test\bar\baz\bag
test\bar\baz\bag\foo
test\bar\baz\bag\foo\1223263495976065729.tmp
test\bar\baz\bag\foo\File.txt
test\baz
test\baz\bag
test\baz\bag\foo
test\baz\bag\foo\bar
test\baz\bag\foo\bar\6666183168609095028.tmp
test\baz\bag\foo\bar\File.txt
test\DIR_8683707748599240459
test\DIR_8683707748599240459\pre63666268047873655 49.non
test\foo
```

```
test\foo\bar
test\foo\bar\baz
test\foo\bar\baz\bag
test\foo\bar\baz\bag\4712324129011589115.tmp
test\foo\bar\baz\bag\File.txt
test\Hello.txt
*/
```

First, `refreshTestDir()` checks to see if `test` already exists. If so, its whole directory is removed using our new `rmdir()` utility. Checking to see whether it `exists()` is then redundant, but I wanted to make the point because if you call `createDirectory()` for a directory that already exists, you'll get an exception.

`createFile()` makes an empty file with the argument `Path`; `resolve()` adds the file name to the end of the `test` `Path`.

We attempt to use `createDirectory()` to create a path with more than one level, but this throws an exception because that method can only create a single directory.

I've made `populateTestDir()` a separate method because it's reused in a later example. For each variant, we create the full directory path using `createDirectories()`, then populate that terminal directory with a copy of this file but using a different destination name. Then we add a temporary file generated with `createTempFile()`. Here we let the method generate the entire temporary file name by passing it `null` for the second two arguments.

After the call to `populateTestDir()`, we create a temporary directory underneath `test`. Note that `createTempDirectory()` only has a prefix option for the name, unlike `createTempFile()` which we again use to put a temporary file in our new temporary directory. You can see from the output that if you don't specify a postfix, the ".tmp" postfix will automatically be used.

To display the results, we first try `newDirectoryStream()` which seems promising, but it turns out to only stream the contents of the `test` directory, and no further down. To get a stream of the entire contents of the directory tree, use `Files.walk()`.

File Systems

For completeness, we need a way to find out the rest of the information about the file system. Here, we get the "default" file system using the `static` `FileSystems` utility,

but you can also call `getFileSystem()` on a `Path` object to get the file system that created that `Path`. You can get a file system given a URI, and you can also construct a new file system (for OSes that support it).

```
// files/FileSystemDemo.java
import java.nio.file.*;

public class FileSystemDemo {
  static void show(String id, Object o) {
    System.out.println(id + ": " + o);
  }
  public static void main(String[] args) {
    System.out.println(System.getProperty("os.name"));
    FileSystem fsys = FileSystems.getDefault();
    for(FileStore fs : fsys.getFileStores())
      show("File Store", fs);
    for(Path rd : fsys.getRootDirectories())
      show("Root Directory", rd);
    show("Separator", fsys.getSeparator());
    show("UserPrincipalLookupService",
      fsys.getUserPrincipalLookupService());
    show("isOpen", fsys.isOpen());
    show("isReadOnly", fsys.isReadOnly());
    show("FileSystemProvider", fsys.provider());
    show("File Attribute Views",
      fsys.supportedFileAttributeViews());
  }
}
/* Output:
Windows 8.1
File Store: (C:)
File Store: System Reserved (E:)
File Store: (F:)
Root Directory: C:\
Root Directory: D:\
Root Directory: E:\
Root Directory: F:\
Separator: \
UserPrincipalLookupService:
sun.nio.fs.WindowsFileSystem$LookupService$1@19e0bfd
isOpen: true
isReadOnly: false
```

```
FileSystemProvider:
sun.nio.fs.WindowsFileSystemProvider@139a55
File Attribute Views: [owner, dos, acl, basic, user]
*/
```

A `FileSystem` can also produce a `WatchService` and a `PathMatcher`.

Watching a `Path`

A `WatchService` enables you to set up a process that reacts to changes within a directory. In this example, `delTxtFiles()` runs as a separate task that goes through the whole directory tree and deletes all files that have names ending with `.txt`, and the `WatchService` reacts to file deletions:

```
// files/PathWatcher.java
// {ExcludeFromGradle}
import java.io.IOException;
import java.nio.file.*;
import static java.nio.file.StandardWatchEventKinds.*;
import java.util.concurrent.*;

public class PathWatcher {
  static Path test = Paths.get("test");
  static void delTxtFiles() {
    try {
      Files.walk(test)
        .filter(f ->
          f.toString().endsWith(".txt"))
        .forEach(f -> {
          try {
            System.out.println("deleting " + f);
            Files.delete(f);
          } catch(IOException e) {
            throw new RuntimeException(e);
          }
        });
    } catch(IOException e) {
      throw new RuntimeException(e);
    }
  }
```

```
  public static void
  main(String[] args) throws Exception {
    Directories.refreshTestDir();
    Directories.populateTestDir();
    Files.createFile(test.resolve("Hello.txt"));
    WatchService watcher =
      FileSystems.getDefault().newWatchService();
    test.register(watcher, ENTRY_DELETE);
    Executors.newSingleThreadScheduledExecutor()
      .schedule(
        PathWatcher::delTxtFiles,
        250, TimeUnit.MILLISECONDS);
    WatchKey key = watcher.take();
    for(WatchEvent evt : key.pollEvents()) {
      System.out.println(
        "evt.context(): " + evt.context() +
        "\nevt.count(): " + evt.count() +
        "\nevt.kind(): " + evt.kind());
      System.exit(0);
    }
  }
}
/* Output:
deleting test\bag\foo\bar\baz\File.txt
deleting test\bar\baz\bag\foo\File.txt
deleting test\baz\bag\foo\bar\File.txt
deleting test\foo\bar\baz\bag\File.txt
deleting test\Hello.txt
evt.context(): Hello.txt
evt.count(): 1
evt.kind(): ENTRY_DELETE
*/
```

The `try` blocks in `delTxtFiles()` look redundant, because they're both catching the same type of exception, and it seems like the outer `try` should be enough. However, Java demands both for some reason (this might be a bug). Also note that in `filter()` I must explicitly convert `f.toString()`, otherwise I'll get the `endsWith()` that compares to an entire `Path` object rather than part of its `String` name.

Once we get a `WatchService` from the `FileSystem`, we register it with the `test Path` along with the variable argument list of items we are interested in—you have

a choice of watching for `ENTRY_CREATE`, `ENTRY_DELETE` or `ENTRY_MODIFY` (creation and deletion doesn't qualify as modification).

Because the upcoming call to `watcher.take()` stops everything until something happens, I want `delTxtFiles()` to start running in parallel so it can generate our event of interest. To do this, I first procure a `ScheduledExecutorService` by calling `Executors.newSingleThreadScheduledExecutor()`, then call `schedule()`, handing it the method reference of the desired function and how long it should wait before running it.

At this point, `watcher.take()` sits and waits. When something happens that fits our target pattern, a `WatchKey` is returned containing `WatchEvents`. The three methods shown are all you can do with a `WatchEvent`.

Look at the output and see what happens. Even though we are deleting files that end with `.txt`, the `WatchService` doesn't get triggered until `Hello.txt` gets deleted. You might *think* that if you say "watch this directory," it would naturally include the entire subtree, but it's very literal: it only watches *that* directory, and *not* everything beneath it. If you want to watch the entire directory tree, you must put a `WatchService` on every subdirectory in the whole tree:

```
// files/TreeWatcher.java
// {ExcludeFromGradle}
import java.io.IOException;
import java.nio.file.*;
import static java.nio.file.StandardWatchEventKinds.*;
import java.util.concurrent.*;

public class TreeWatcher {
  static void watchDir(Path dir) {
    try {
      WatchService watcher =
        FileSystems.getDefault().newWatchService();
      dir.register(watcher, ENTRY_DELETE);
      Executors.newSingleThreadExecutor().submit(() -> {
        try {
          WatchKey key = watcher.take();
          for(WatchEvent evt : key.pollEvents()) {
            System.out.println(
              "evt.context(): " + evt.context() +
              "\nevt.count(): " + evt.count() +
```

```
              "\nevt.kind(): " + evt.kind());
            System.exit(0);
          }
        } catch(InterruptedException e) {
          return;
        }
      });
    } catch(IOException e) {
      throw new RuntimeException(e);
    }
  }
  public static void
  main(String[] args) throws Exception {
    Directories.refreshTestDir();
    Directories.populateTestDir();
    Files.walk(Paths.get("test"))
      .filter(Files::isDirectory)
      .forEach(TreeWatcher::watchDir);
    PathWatcher.delTxtFiles();
  }
}
/* Output:
deleting test\bag\foo\bar\baz\File.txt
deleting test\bar\baz\bag\foo\File.txt
evt.context(): File.txt
evt.count(): 1
evt.kind(): ENTRY_DELETE
*/
```

The `watchDir()` method puts a `WatchService` for `ENTRY_DELETE` on its argument, and also starts an independent process to monitor that `WatchService`. Here, we don't `schedule()` a task to run later, but instead `submit()` it to run right now. We walk the entire directory tree and apply `watchDir()` to each subdirectory. Now when we run `delTxtFiles()`, one of the `WatchServices` detects the very first deletion.

Finding Files

To find files up until now, we've been using the rather crude approach of calling `toString()` on the `Path`, then using `String` operations to look at the result. It turns

out that `java.nio.file` has a better solution: the `PathMatcher`. You get one by calling `getPathMatcher()` on the `FileSystem` object, and you pass in your pattern of interest. There are two options for patterns: `glob` and `regex`. `glob` is simpler and is actually quite powerful so you'll be able to solve many problems using `glob`. If your problem is more complex, you can use `regex`, which is explained in the upcoming `Strings` chapter.

Here we use `glob` to find all the `Paths` that end with `.tmp` or `.txt`:

```
// files/Find.java
// {ExcludeFromGradle}
import java.nio.file.*;

public class Find {
  public static void
  main(String[] args) throws Exception {
    Path test = Paths.get("test");
    Directories.refreshTestDir();
    Directories.populateTestDir();
    // Creating a *directory*, not a file:
    Files.createDirectory(test.resolve("dir.tmp"));

    PathMatcher matcher = FileSystems.getDefault()
      .getPathMatcher("glob:**/*.{tmp,txt}");
    Files.walk(test)
      .filter(matcher::matches)
      .forEach(System.out::println);
    System.out.println("***************");

    PathMatcher matcher2 = FileSystems.getDefault()
      .getPathMatcher("glob:*.tmp");
    Files.walk(test)
      .map(Path::getFileName)
      .filter(matcher2::matches)
      .forEach(System.out::println);
    System.out.println("***************");

    Files.walk(test) // Only look for files
      .filter(Files::isRegularFile)
      .map(Path::getFileName)
      .filter(matcher2::matches)
      .forEach(System.out::println);
```

```
  }
}
/* Output:
test\bag\foo\bar\baz\5208762845883213974.tmp
test\bag\foo\bar\baz\File.txt
test\bar\baz\bag\foo\7918367201207778677.tmp
test\bar\baz\bag\foo\File.txt
test\baz\bag\foo\bar\8016595521026696632.tmp
test\baz\bag\foo\bar\File.txt
test\dir.tmp
test\foo\bar\baz\bag\5832319279813617280.tmp
test\foo\bar\baz\bag\File.txt
***************
5208762845883213974.tmp
7918367201207778677.tmp
8016595521026696632.tmp
dir.tmp
5832319279813617280.tmp
***************
5208762845883213974.tmp
7918367201207778677.tmp
8016595521026696632.tmp
5832319279813617280.tmp
*/
```

In `matcher`, the `**/` at the beginning of the `glob` expression means "all subdirectories," and it's essential if you want to match more than just `Paths` ending in the base directory because it matches the full path up until your desired result. The single `*` is "anything," then a dot, then the curly braces indicate a list of possibilities—we are looking for anything ending with either `.tmp` or `.txt`. You can find further details in the `getPathMatcher()` documentation.

`matcher2` just uses `*.tmp`, which would ordinarily not match anything, but adding the `map()` operation reduces the full path to just the name at the end.

Notice in both cases that `dir.tmp` shows up in the output even though it's a directory and not a file. To only find files, you must filter for them as in the last `Files.walk()`.

Reading & Writing Files

At this point we can do just about anything with paths and directories. Now let's look at manipulating the contents of the files themselves.

If a file is "small," for some definition of "small" (which just means "it runs fast enough for you and doesn't run out of memory"), the `java.nio.file.Files` class contains utilities for easily reading and writing both text and binary files.

`Files.readAllLines()` reads the whole file in at once (thus, the importance of it being a "small" file), producing a `List<String>`. For an example file, we'll reuse `streams/Cheese.dat`:

```
// files/ListOfLines.java
import java.util.*;
import java.nio.file.*;

public class ListOfLines {
  public static void
  main(String[] args) throws Exception {
    Files.readAllLines(
      Paths.get("../streams/Cheese.dat"))
      .stream()
      .filter(line -> !line.startsWith("//"))
      .map(line ->
        line.substring(0, line.length()/2))
      .forEach(System.out::println);
  }
}
/* Output:
Not much of a cheese
Finest in the
And what leads you
Well, it's
It's certainly uncon
*/
```

Comment lines are skipped, and the rest are only printed halfway. Notice how easy it is: you just hand a `Path` to `readAllLines()` (it was far messier in the past). There's an overloaded version of `readAllLines()` that includes a `Charset` argument to establish the Unicode encoding of the file.

`Files.write()` is overloaded to write either an array of `bytes` or anything `Iterable` (which also includes a `Charset` option):

```
// files/Writing.java
import java.util.*;
import java.nio.file.*;

public class Writing {
  static Random rand = new Random(47);
  static final int SIZE = 1000;
  public static void
  main(String[] args) throws Exception {
    // Write bytes to a file:
    byte[] bytes = new byte[SIZE];
    rand.nextBytes(bytes);
    Files.write(Paths.get("bytes.dat"), bytes);
    System.out.println("bytes.dat: " +
      Files.size(Paths.get("bytes.dat")));

    // Write an iterable to a file:
    List<String> lines = Files.readAllLines(
      Paths.get("../streams/Cheese.dat"));
    Files.write(Paths.get("Cheese.txt"), lines);
    System.out.println("Cheese.txt: " +
      Files.size(Paths.get("Cheese.txt")));
  }
}
/* Output:
bytes.dat: 1000
Cheese.txt: 199
*/
```

We use `Random` to create a thousand random `bytes`; you can see the resulting file size is 1000.

A `List` is written to a file here, but anything `Iterable` will work.

What if file size is an issue? Perhaps:

1. The file is so big you might run out of memory if you read the whole thing at once.
2. You only need to work partway through the file to get the results you want, so reading the whole file wastes time.

`Files.lines()` conveniently turns a file into a `Stream` of lines:

```
// files/ReadLineStream.java
import java.nio.file.*;

public class ReadLineStream {
  public static void
  main(String[] args) throws Exception {
    Files.lines(Paths.get("PathInfo.java"))
      .skip(13)
      .findFirst()
      .ifPresent(System.out::println);
  }
}
/* Output:
    show("RegularFile", Files.isRegularFile(p));
*/
```

This streams the first example in this chapter, skips 13 lines, takes the next line and prints it.

`Files.lines()` is very useful for processing a file as an *incoming* `Stream` of lines, but what if you want to read, process, and write, all in a single `Stream`? This requires slightly more complex code:

```
// files/StreamInAndOut.java
import java.io.*;
import java.nio.file.*;
import java.util.stream.*;

public class StreamInAndOut {
  public static void main(String[] args) {
    try(
      Stream<String> input =
        Files.lines(Paths.get("StreamInAndOut.java"));
      PrintWriter output =
        new PrintWriter("StreamInAndOut.txt")
    ) {
      input
        .map(String::toUpperCase)
        .forEachOrdered(output::println);
    } catch(Exception e) {
```

```
      throw new RuntimeException(e);
    }
  }
}
```

Because we're performing all the manipulation within the same block, both files can be opened within the same try-with-resources statement. `PrintWriter` is an old-style `java.io` class that allows you to "print to" a file, so it's ideal for this application. If you look at `StreamInAndOut.txt` you'll see it is indeed in all uppercase.

Summary

Although this has been a fairly thorough introduction to file and directory manipulation, there are still unexplored features in the library—be sure to study the Javadocs for `java.nio.file`, especially `java.nio.file.Files`.

The Java 7 & 8 improvements in libraries for working with files and directories are monumental. If you're just getting started with Java, you're lucky. In the past, it was so unpleasant I was convinced that the Java designers just didn't consider file manipulation important enough to make it easy. It was a definite turnoff for beginner, and for teaching the language to beginners. I don't understand why it took so long to fix this glaring issue but however it happened, I'm glad. Working with files is now easy and even fun, something you could never say before.

Strings

> String manipulation is arguably one of the most common activities in computer programming.

This is especially true in Web systems, where Java is a major player. In this chapter, we'll look more deeply at what may be the most heavily used class in the language, `String`, along with some of its associated classes and utilities.

Immutable `String`s

Objects of the `String` class are immutable. If you examine the JDK documentation for the `String` class, you'll see that every method in the class that appears to modify a `String` actually creates and returns a brand new `String` object containing the modification. The original `String` is left untouched.

Consider the following code:

```
// strings/Immutable.java

public class Immutable {
  public static String upcase(String s) {
    return s.toUpperCase();
  }
  public static void main(String[] args) {
    String q = "howdy";
    System.out.println(q); // howdy
    String qq = upcase(q);
    System.out.println(qq); // HOWDY
    System.out.println(q); // howdy
  }
}
/* Output:
howdy
```

```
HOWDY
howdy
*/
```

When `q` is passed in to `upcase()` it's actually a copy of the reference to `q`. The object this reference is connected to stays in a single physical location. The references are copied as they are passed around.

Looking at the definition for `upcase()`, notice that the reference that's passed in has the name `s`, and it exists for only as long as the body of `upcase()` is being executed. When `upcase()` completes, the local reference `s` vanishes. `upcase()` returns the result: a reference to the original `String`, with all the characters set to uppercase. But the reference it returns is for a new object, and the original `q` is left alone.

This behavior is usually what you want. Suppose you say:

```
String s = "asdf";
String x = Immutable.upcase(s);
```

Do you really want the `upcase()` method to *change* the argument? To the reader of the code, an argument usually looks like a piece of information provided to the method, not something to be modified. This is an important guarantee because it makes code easier to write and understand.

Overloading `+` vs. `StringBuilder`

`String` objects are immutable so you can alias to a particular `String` as many times as you want. Because a `String` is read-only, there's no possibility that one reference will change something that affect the other references.

Immutability can have efficiency issues. A case in point is the operator `+` that is overloaded for `String` objects. Overloading means an operation has extra meaning when used with a particular class. (The `+` and `+=` for `String` are the only operators overloaded in Java, and Java does not allow the programmer to overload any others.)[135]

[135] C++ allows the programmer to overload operators at will. Because this can often be a complicated process (see Chapter 10 of *Thinking in C++, 2nd Edition*, Prentice Hall, 2000), the Java designers deemed it a "bad" feature that shouldn't be included in Java. It wasn't so bad they didn't end up doing it themselves, and ironically enough, operator overloading would be much easier to use in Java than in C++. This can be seen in Python (see *www.Python.org*) and C#, which have garbage collection and straightforward operator overloading.

The + operator concatenates `String`s:

```
// strings/Concatenation.java

public class Concatenation {
  public static void main(String[] args) {
    String mango = "mango";
    String s = "abc" + mango + "def" + 47;
    System.out.println(s);
  }
}
/* Output:
abcmangodef47
*/
```

Imagine how this *might* work. The `String` "abc" could have a method `append()` that creates a new `String` object containing "abc" concatenated with the contents of `mango`. The new `String` object would then create another new `String` that added "def," and so on.

This would certainly work, but it requires the creation of many `String` objects just to put together this new `String`, then you have a bunch of intermediate `String` objects that must be garbage collected. I suspect that the Java designers tried this approach first (a lesson in software design—you don't really know anything about a system until you try it out in code and get something working). I also suspect they discovered it delivered unacceptable performance.

To see what really happens, you can decompile the above code using the `javap` tool that comes as part of the JDK. Here's the command line:

```
javap -c Concatenation
```

The `-c` flag will produce the JVM bytecodes. After we strip out the parts we're not interested in and do a bit of editing, here are the relevant bytecodes:

```
public static void main(java.lang.String[]);
 Code:
  Stack=2, Locals=3, Args_size=1
  0:  ldc #2; //String mango
  2:  astore_1
  3:  new #3; //class StringBuilder
  6:  dup
  7:  invokespecial #4; //StringBuilder."<init>":()
  10: ldc #5; //String abc
  12: invokevirtual #6; //StringBuilder.append:(String)
  15: aload_1
  16: invokevirtual #6; //StringBuilder.append:(String)
  19: ldc #7; //String def
  21: invokevirtual #6; //StringBuilder.append:(String)
  24: bipush 47
  26: invokevirtual #8; //StringBuilder.append:(I)
  29: invokevirtual #9; //StringBuilder.toString:()
  32: astore_2
  33: getstatic #10; //Field System.out:PrintStream;
  36: aload_2
  37: invokevirtual #11; //PrintStream.println:(String)
  40: return
```

If you've had experience with assembly language, this might look familiar to you—statements like `dup` and `invokevirtual` are the Java Virtual Machine (JVM) equivalent of assembly language. If you've never seen assembly language, don't worry about it—the important part to notice is the introduction by the compiler of the `java.lang.StringBuilder` class. There was no mention of `StringBuilder` in the source code, but the compiler decided to use it anyway, because it is much more efficient.

here, the compiler creates a `StringBuilder` object to build the `String s`, and calls `append()` four times, one for each of the pieces. Finally, it calls `toString()` to produce the result, which it stores (with `astore_2`) as `s`.

Before you assume you can just use `String`s everywhere and that the compiler will make everything efficient, let's look a little more closely at what the compiler is doing. Here's an example that produces a `String` result in two ways: using `String`s, and by hand-coding with `StringBuilder`:

```
// strings/WhitherStringBuilder.java

public class WhitherStringBuilder {
  public String implicit(String[] fields) {
    String result = "";
    for(String field : fields) {
      result += field;
    }
    return result;
  }
  public String explicit(String[] fields) {
    StringBuilder result = new StringBuilder();
    for(String field : fields) {
      result.append(field);
    }
    return result.toString();
  }
}
```

Now if you run `javap -c WitherStringBuilder`, you see the code for the two different methods (I've removed needless details). First, `implicit()`:

```
public java.lang.String implicit(java.lang.String[]);
0: ldc           #2  // String
2: astore_2
3: aload_1
4: astore_3
5: aload_3
6: arraylength
7: istore        4
9: iconst_0
10: istore        5
12: iload         5
14: iload         4
16: if_icmpge     51
19: aload_3
20: iload         5
22: aaload
23: astore        6
25: new           #3  // StringBuilder
28: dup
29: invokespecial #4  // StringBuilder."<init>"
```

```
32: aload_2
33: invokevirtual #5  // StringBuilder.append:(String)
36: aload         6
38: invokevirtual #5  // StringBuilder.append:(String;)
41: invokevirtual #6  // StringBuilder.toString:()
44: astore_2
45: iinc          5, 1
48: goto          12
51: aload_2
52: areturn
```

Notice `16:` and `35:`, which together form a loop. `16:` does an "integer compare greater than or equal to" of the operands on the stack and jumps to `51:` when the loop is done. `48:` is a goto back to the beginning of the loop, at `12:`. Notice that the `StringBuilder` construction happens *inside* this loop, which means you're going to get a new `StringBuilder` object every time you pass through the loop.

Here are the bytecodes for `explicit()`:

```
public java.lang.String explicit(java.lang.String[]);
0: new           #3  // StringBuilder
3: dup
4: invokespecial #4  // StringBuilder."<init>"
7: astore_2
8: aload_1
9: astore_3
10: aload_3
11: arraylength
12: istore        4
14: iconst_0
15: istore        5
17: iload         5
19: iload         4
21: if_icmpge     43
24: aload_3
25: iload         5
27: aaload
28: astore        6
30: aload_2
31: aload         6
33: invokevirtual #5  // StringBuilder.append:(String)
36: pop
```

```
37: iinc          5, 1
40: goto          17
43: aload_2
44: invokevirtual #6  // StringBuilder.toString:()
47: areturn
```

Not only is the loop code shorter and simpler, the method only creates a single `StringBuilder` object. With an explicit `StringBuilder`, you can preallocate its size if you know how big it might be, so it doesn't constantly reallocate the buffer.

Thus, when you create a `toString()` method, if the operations are simple ones the compiler can figure out on its own, you can generally rely on it to build the result in a reasonable fashion. But if looping is involved *and performance is an issue*, explicitly use a `StringBuilder` in your `toString()`, like this:

```
// strings/UsingStringBuilder.java
import java.util.*;
import java.util.stream.*;

public class UsingStringBuilder {
  public static String string1() {
    Random rand = new Random(47);
    StringBuilder result = new StringBuilder("[");
    for(int i = 0; i < 25; i++) {
      result.append(rand.nextInt(100));
      result.append(", ");
    }
    result.delete(result.length()-2, result.length());
    result.append("]");
    return result.toString();
  }
  public static String string2() {
    String result = new Random(47)
      .ints(25, 0, 100)
      .mapToObj(Integer::toString)
      .collect(Collectors.joining(", "));
    return "[" + result + "]";
  }
  public static void main(String[] args) {
    System.out.println(string1());
    System.out.println(string2());
  }
```

```
}
/* Output:
[58, 55, 93, 61, 61, 29, 68, 0, 22, 7, 88, 28, 51, 89,
9, 78, 98, 61, 20, 58, 16, 40, 11, 22, 4]
[58, 55, 93, 61, 61, 29, 68, 0, 22, 7, 88, 28, 51, 89,
9, 78, 98, 61, 20, 58, 16, 40, 11, 22, 4]
*/
```

In `string1()`, each piece of the result is added with an `append()` statement. If you try to take shortcuts and do something like `append(a + ": " + c)`, the compiler will jump in and start making more `StringBuilder` objects again. If you are in doubt about which approach to use, you can always run `javap` to double-check.

Although `StringBuilder` has a full complement of methods, including `insert()`, `replace()`, `substring()` and even `reverse()`, the ones you generally use are `append()` and `toString()`. Note how the call to `delete()` removes the last comma and space before adding the closing square bracket.

`string2()` uses `Streams` and the resulting code is far more aesthetically pleasing. As it turns out, `Collectors.joining()` also uses a `StringBuilder` internally, so you lose nothing!

`StringBuilder` was introduced in Java 5. Prior to this, Java used `StringBuffer`, which ensured thread safety (see the Concurrent Programming chapter) and so was significantly more expensive. With `StringBuilder`, `String` operations should be faster.

Unintended Recursion

Because (like every other class) the Java standard collections are ultimately inherited from `Object`, they contain a `toString()` method. This is overridden so they produce a `String` representation of themselves, including the objects they hold. `ArrayList.toString()`, for example, steps through the elements of the `ArrayList` and calls `toString()` for each one:

```
// strings/ArrayListDisplay.java
import java.util.*;
import java.util.stream.*;
import generics.coffee.*;

public class ArrayListDisplay {
  public static void main(String[] args) {
    List<Coffee> coffees =
      Stream.generate(new CoffeeSupplier())
        .limit(10)
        .collect(Collectors.toList());
    System.out.println(coffees);
  }
}
/* Output:
[Americano 0, Latte 1, Americano 2, Mocha 3, Mocha 4,
Breve 5, Americano 6, Latte 7, Cappuccino 8, Cappuccino
9]
*/
```

Suppose you'd like your `toString()` to print the address of your class. It seems to make sense to refer to `this`:

```
// strings/InfiniteRecursion.java
// Accidental recursion
// {ThrowsException}
// {VisuallyInspectOutput} Throws very long exception
import java.util.*;
import java.util.stream.*;

public class InfiniteRecursion {
  @Override public String toString() {
    return
      " InfiniteRecursion address: " + this + "\n";
  }
  public static void main(String[] args) {
    Stream.generate(InfiniteRecursion::new)
      .limit(10)
      .forEach(System.out::println);
  }
}
```

If you create an `InfiniteRecursion` object, then display it, you'll get a very long sequence of exceptions. This is also true if you place the `InfiniteRecursion` objects in an `ArrayList` and print that `ArrayList` as shown here. What's happening is *automatic type conversion* for `String`s. When you say:

```
"InfiniteRecursion address: " + this
```

The compiler sees a `String` followed by a `+` and something that's not a `String`, so it tries to convert `this` to a `String`. It does this conversion by calling `toString()`, which produces a recursive call.

If you really want to print the address of the object, the solution is to call the `Object` `toString()` method, which does just that. So instead of saying `this`, say `super.toString()`.

Operations on `String`s

Here are most of the methods available for `String` objects. Overloaded methods are summarized in a single row:

Method	Arguments, Overloading	Use
Constructor	Overloaded: default, `String`, `StringBuilder`, `StringBuffer`, `char` arrays, `byte` arrays.	Creating `String` objects.
`length()`		Number of characters in the `String`.
`charAt()`	`int Index`	The `char` at a location in the `String`.
`getChars()`, `getBytes()`	The beginning and end from which to copy, the array to copy into, an index into the destination array.	Copy `char`s or `byte`s into an external array.

Method	Arguments, Overloading	Use
`toCharArray()`		Produces a `char[]` containing the characters in the `String`.
`equals()`, `equalsIgnoreCase()`	A `String` to compare with.	An equality check on the contents of the two `String`s. Result is `true` if the contents are equal.
`compareTo()` `compareToIgnoreCase()`	A `String` to compare with.	Result is negative, zero, or positive depending on the lexicographical ordering of the `String` and the argument. Uppercase and lowercase are not equal!
`contains()`	The `CharSequence` you want to find.	Result is `true` if the argument is contained in the `String`.
`contentEquals()`	A `CharSequence` or `StringBuffer` to compare.	Result is `true` if there's an exact match with the argument.
`isEmpty()`		`boolean` result indicates whether the `String` is length `0`.
`regionMatches()`	Offset into this `String`, the other `String` and its offset and length to compare. Overload adds "ignore case."	`boolean` result indicates whether the region matches.
`startsWith()`	`String` that it might start with. Overload adds offset into argument.	`boolean` result indicates whether the `String` starts with the argument.

Method	Arguments, Overloading	Use
`endsWith()`	`String` that might be a suffix of this `String`.	`boolean` result indicates whether the argument is a suffix.
`indexOf()`, `lastIndexOf()`	Overloaded: `char`, `char` and starting index, `String`, `String` and starting index.	Returns -1 if the argument is not found within this `String`; otherwise, returns the index where the argument starts. `lastIndexOf()` searches backward from end.
`matches()`	A regular expression.	Returns `boolean` indicating whether this `String` matches the given regular expression.
`split()`	A regular expression. Optional second argument is maximum number of splits to make.	Splits the `String` around the regular expression. Returns an array of results.
`join()` (introduced in Java 8)	Delimiter, elements. Produces a new `String` by joining together elements with delimiter.	Pieces become a new `String` separated by delimiter.
`substring()` (also `subSequence()`)	Overloaded: starting index; starting index + ending index.	Returns a new `String` object containing the specified character set.
`concat()`	The `String` to concatenate.	Returns a new `String` object containing the original `String`'s characters followed by the characters in the argument.

Method	Arguments, Overloading	Use
`replace()`	The old character to search for, the new character to replace it with. Can also replace a `CharSequence` with a `CharSequence`.	Returns a new `String` object with the replacements made. Uses the old `String` if no match is found.
`replaceFirst()`	A regular expression to search for, the new `String` to replace it with.	Returns a new `String` object with the replacement made.
`replaceAll()`	A regular expression to search for, the new `String` to replace it with.	Returns a new `String` object with all replacements made.
`toLowerCase()`, `toUpperCase()`		Returns a new `String` object with the case of all letters changed. Uses the old `String` if no changes are made.
`trim()`		Returns a new `String` object with the whitespace removed from each end. Uses the old `String` if no changes are made.
`valueOf()` (`static`)	Overloaded: `Object`, `char[]`, `char[]` and offset and count, `boolean`, `char`, `int`, `long`, `float`, `double`.	Returns a `String` containing a character representation of the argument.
`intern()`		Produces one and only one `String` reference per unique character sequence.
`format()`	Format string and arguments for substitution into the format specifiers.	Produces formatted result `String`.

Every `String` method carefully returns a new `String` object when it's necessary to change the contents. If the contents don't need changing, the method just returns a reference to the original `String`. This saves storage and overhead.

The `String` methods involving *regular expressions* are explained later in this chapter.

Formatting Output

One of the long-awaited features that finally appeared in Java 5 is output formatting in the style of C's `printf()` statement. Not only does this allow for simplified output code, but it also gives Java developers powerful control over output formatting and alignment.

`printf()`

C's `printf()` doesn't assemble strings the way Java does, but takes a single *format string* and inserts values into it, formatting as it goes. Instead of using the overloaded + operator (which C doesn't overload) to concatenate quoted text and variables, `printf()` uses special placeholders to show where the data should go. The arguments inserted into the format `String` follow in a comma-separated list. For example:

```
System.out.printf("Row 1: [%d %f]%n", x, y);
```

At run time, the value of `x` is inserted into `%d` and the value of `y` is inserted into `%f`. These placeholders are called *format specifiers* and, in addition to telling where to insert the value, they also tell what kind of variable is inserted and how to format it. For instance, the `%d` above says that `x` is an integer and the `%f`says `y` is a floating point value (a `float` or `double`).

`System.out.format()`

Java 5 introduced the `format()` method, available to `PrintStream` or `PrintWriter` objects (which you'll learn more about in the Appendix: I/O Streams), which includes `System.out`. The `format()` method is modeled after C's `printf()`. There's even a convenience `printf()` method you can use if you're feeling nostalgic, which just calls `format()`. Here's a simple example:

```
// strings/SimpleFormat.java

public class SimpleFormat {
  public static void main(String[] args) {
    int x = 5;
    double y = 5.332542;
    // The old way:
    System.out.println("Row 1: [" + x + " " + y + "]");
    // The new way:
    System.out.format("Row 1: [%d %f]%n", x, y);
    // or
    System.out.printf("Row 1: [%d %f]%n", x, y);
  }
}
/* Output:
Row 1: [5 5.332542]
Row 1: [5 5.332542]
Row 1: [5 5.332542]
*/
```

`format()` and `printf()` are equivalent. In both cases, there's only a single format `String`, followed by one argument for each format specifier.

The `String` class also has a `static format()` method which produces a formatted `String`.

The `Formatter` Class

All of Java's formatting functionality is handled by the `Formatter` class in the `java.util` package. You can think of `Formatter` as a translator that converts your format `String` and data into the desired result. When you create a `Formatter` object, you tell it where you want this result to go by passing that information to the constructor:

```
// strings/Turtle.java
import java.io.*;
import java.util.*;

public class Turtle {
  private String name;
  private Formatter f;
  public Turtle(String name, Formatter f) {
    this.name = name;
    this.f = f;
  }
  public void move(int x, int y) {
    f.format("%s The Turtle is at (%d,%d)%n",
      name, x, y);
  }
  public static void main(String[] args) {
    PrintStream outAlias = System.out;
    Turtle tommy = new Turtle("Tommy",
      new Formatter(System.out));
    Turtle terry = new Turtle("Terry",
      new Formatter(outAlias));
    tommy.move(0,0);
    terry.move(4,8);
    tommy.move(3,4);
    terry.move(2,5);
    tommy.move(3,3);
    terry.move(3,3);
  }
}
/* Output:
Tommy The Turtle is at (0,0)
Terry The Turtle is at (4,8)
Tommy The Turtle is at (3,4)
Terry The Turtle is at (2,5)
Tommy The Turtle is at (3,3)
Terry The Turtle is at (3,3)
*/
```

The `%s` format specifier indicates a `String` argument.

All the `tommy` output goes to `System.out` and all the `terry` output goes to an alias of `System.out`. The constructor is overloaded to take a range of output locations, but

the most useful are `PrintStreams` (as above), `OutputStreams`, and `Files`. You'll learn more about these in the Appendix: I/O Streams.

Format Specifiers

To control spacing and alignment when inserting data, you need more elaborate format specifiers. Here's the general syntax:

```
%[argument_index$][flags][width][.precision]conversion
```

Often, you must control the minimum size of a field. This can be accomplished by specifying a *width*. The `Formatter` guarantees that a field is at least a certain number of characters wide by padding it with spaces if necessary. By default, the data is right justified, but this can be overridden by including a `-` in the flags section.

The opposite of *width* is *precision*, used to specify a maximum. Unlike the *width*, which is applicable to all data conversion types and behaves the same with each, *precision* has a different meaning for different types. For `Strings`, the *precision* specifies the maximum number of `String` characters to print. For floating point numbers, *precision* specifies the number of decimal places to display (the default is 6), rounding if there are too many or adding trailing zeroes if there are too few. Integers have no fractional part so *precision* isn't applicable to them and you'll get an exception if you use precision with an integer conversion type.

Here, we'll use format specifiers to print a shopping receipt. This is a very simple example of the *Builder* design pattern, where you create a starting object, add things to it, and finally complete it with the `build()` method:

```
// strings/ReceiptBuilder.java
import java.util.*;

public class ReceiptBuilder {
  private double total = 0;
  private Formatter f =
    new Formatter(new StringBuilder());
  public ReceiptBuilder() {
    f.format(
      "%-15s %5s %10s%n", "Item", "Qty", "Price");
    f.format(
```

```
      "%-15s %5s %10s%n", "----", "---", "-----");
  }
  public void add(String name, int qty, double price) {
    f.format("%-15.15s %5d %10.2f%n", name, qty, price);
    total += price * qty;
  }
  public String build() {
    f.format("%-15s %5s %10.2f%n", "Tax", "",
      total * 0.06);
    f.format("%-15s %5s %10s%n", "", "", "-----");
    f.format("%-15s %5s %10.2f%n", "Total", "",
      total * 1.06);
    return f.toString();
  }
  public static void main(String[] args) {
    ReceiptBuilder receiptBuilder =
      new ReceiptBuilder();
    receiptBuilder.add("Jack's Magic Beans", 4, 4.25);
    receiptBuilder.add("Princess Peas", 3, 5.1);
    receiptBuilder.add(
      "Three Bears Porridge", 1, 14.29);
    System.out.println(receiptBuilder.build());
  }
}
/* Output:
Item             Qty      Price
----             ---      -----
Jack's Magic Be    4       4.25
Princess Peas      3       5.10
Three Bears Por    1      14.29
Tax                        2.80
                           -----
Total                     49.39
*/
```

By passing a `StringBuilder` to the `Formatter` constructor, I give it a place to build the `String`; you can also send it to standard output or even a file using the constructor argument.

`Formatter` provides powerful control over spacing and alignment with fairly concise notation. Here, the format `Strings` are copied to produce the appropriate spacing.

`Formatter` Conversions

These are the conversion characters you'll come across most frequently:

Character	Effect
`d`	Integral (as decimal)
`c`	Unicode character
`b`	Boolean value
`s`	String
`f`	Floating point (as decimal)
`e`	Floating point (in scientific notation)
`x`	Integral (as hex)
`h`	Hash code (as hex)
`%`	Literal "%"

Here's an example that shows these conversions in action:

```
// strings/Conversion.java
import java.math.*;
import java.util.*;

public class Conversion {
  public static void main(String[] args) {
    Formatter f = new Formatter(System.out);

    char u = 'a';
    System.out.println("u = 'a'");
    f.format("s: %s%n", u);
    // f.format("d: %d%n", u);
    f.format("c: %c%n", u);
    f.format("b: %b%n", u);
    // f.format("f: %f%n", u);
    // f.format("e: %e%n", u);
```

```
    // f.format("x: %x%n", u);
    f.format("h: %h%n", u);

    int v = 121;
    System.out.println("v = 121");
    f.format("d: %d%n", v);
    f.format("c: %c%n", v);
    f.format("b: %b%n", v);
    f.format("s: %s%n", v);
    // f.format("f: %f%n", v);
    // f.format("e: %e%n", v);
    f.format("x: %x%n", v);
    f.format("h: %h%n", v);

    BigInteger w = new BigInteger("50000000000000");
    System.out.println(
      "w = new BigInteger(\"50000000000000\")");
    f.format("d: %d%n", w);
    // f.format("c: %c%n", w);
    f.format("b: %b%n", w);
    f.format("s: %s%n", w);
    // f.format("f: %f%n", w);
    // f.format("e: %e%n", w);
    f.format("x: %x%n", w);
    f.format("h: %h%n", w);

    double x = 179.543;
    System.out.println("x = 179.543");
    // f.format("d: %d%n", x);
    // f.format("c: %c%n", x);
    f.format("b: %b%n", x);
    f.format("s: %s%n", x);
    f.format("f: %f%n", x);
    f.format("e: %e%n", x);
    // f.format("x: %x%n", x);
    f.format("h: %h%n", x);

    Conversion y = new Conversion();
    System.out.println("y = new Conversion()");
    // f.format("d: %d%n", y);
    // f.format("c: %c%n", y);
    f.format("b: %b%n", y);
```

```
    f.format("s: %s%n", y);
    // f.format("f: %f%n", y);
    // f.format("e: %e%n", y);
    // f.format("x: %x%n", y);
    f.format("h: %h%n", y);

    boolean z = false;
    System.out.println("z = false");
    // f.format("d: %d%n", z);
    // f.format("c: %c%n", z);
    f.format("b: %b%n", z);
    f.format("s: %s%n", z);
    // f.format("f: %f%n", z);
    // f.format("e: %e%n", z);
    // f.format("x: %x%n", z);
    f.format("h: %h%n", z);
  }
}
/* Output:
u = 'a'
s: a
c: a
b: true
h: 61
v = 121
d: 121
c: y
b: true
s: 121
x: 79
h: 79
w = new BigInteger("50000000000000")
d: 50000000000000
b: true
s: 50000000000000
x: 2d79883d2000
h: 8842a1a7
x = 179.543
b: true
s: 179.543
f: 179.543000
e: 1.795430e+02
```

```
h: 1ef462c
y = new Conversion()
b: true
s: Conversion@19e0bfd
h: 19e0bfd
z = false
b: false
s: false
h: 4d5
*/
```

The commented lines are invalid conversions for that particular variable type; executing them will trigger an exception.

Notice that the `b` conversion works for each variable above. Although it's valid for any argument type, it might not behave as you'd expect. For `boolean` primitives or `Boolean` objects, the result is `true` or `false`, accordingly. However, for any other argument, as long as the argument type is not `null` the result is always `true`. Even the numeric value of zero, synonymous with `false` in many languages (including C), will produce `true`, so be careful when using this conversion with non-Boolean types.

There are more obscure conversion types and other format specifier options. You can read about these in the JDK documentation for the `Formatter` class.

`String.format()`

Java 5 also took a cue from C's `sprintf()`, which is used to create `Strings`. `String.format()` is a `static` method which takes all the same arguments as `Formatter`'s `format()` but returns a `String`. It can come in handy when you only call `format()` once:

```
// strings/DatabaseException.java

public class DatabaseException extends Exception {
  public DatabaseException(int transactionID,
    int queryID, String message) {
    super(String.format("(t%d, q%d) %s", transactionID,
        queryID, message));
  }
  public static void main(String[] args) {
    try {
      throw new DatabaseException(3, 7, "Write failed");
    } catch(Exception e) {
      System.out.println(e);
    }
  }
}
/* Output:
DatabaseException: (t3, q7) Write failed
*/
```

Under the hood, all `String.format()` does is instantiate a `Formatter` and pass your arguments to it, but using this convenience method can often be clearer and easier than doing it by hand.

A Hex Dump Tool

As a second example, let's format the bytes in a binary file as hexadecimal. Here's a small utility that displays a binary array of bytes in a readable hex format, using `String.format()`:

```
// strings/Hex.java
// {java onjava.Hex}
package onjava;
import java.io.*;
import java.nio.file.*;

public class Hex {
  public static String format(byte[] data) {
    StringBuilder result = new StringBuilder();
    int n = 0;
    for(byte b : data) {
```

```
      if(n % 16 == 0)
        result.append(String.format("%05X: ", n));
      result.append(String.format("%02X ", b));
      n++;
      if(n % 16 == 0) result.append("\n");
    }
    result.append("\n");
    return result.toString();
  }
  public static void
  main(String[] args) throws Exception {
    if(args.length == 0)
      // Test by displaying this class file:
      System.out.println(format(
        Files.readAllBytes(Paths.get(
          "build/classes/java/main/onjava/Hex.class"))));
    else
      System.out.println(format(
        Files.readAllBytes(Paths.get(args[0]))));
  }
}
/* Output: (First 6 Lines)
00000: CA FE BA BE 00 00 00 34 00 61 0A 00 05 00 31 07
00010: 00 32 0A 00 02 00 31 08 00 33 07 00 34 0A 00 35
00020: 00 36 0A 00 0F 00 37 0A 00 02 00 38 08 00 39 0A
00030: 00 3A 00 3B 08 00 3C 0A 00 02 00 3D 09 00 3E 00
00040: 3F 08 00 40 07 00 41 0A 00 42 00 43 0A 00 44 00
00050: 45 0A 00 14 00 46 0A 00 47 00 48 07 00 49 01 00
                  ...
*/
```

To open and read the binary file, this uses another utility that is introduced in the Files chapter: `Files.readAllBytes()`, which returns the entire file as a `byte` array.

Regular Expressions

Regular expressions have long been integral to standard Unix utilities like sed and awk, and languages like Python and Perl (some would argue they are the predominant reason for Perl's success). String manipulation tools were previously

delegated to the `String`, `StringBuffer`, and `StringTokenizer` classes in Java, which had relatively simple facilities compared to regular expressions.

Regular expressions are powerful and flexible text-processing tools. They allow you to specify, programmatically, complex patterns of text that can be discovered in an input `String`. Once you discover these patterns, you can then react to them any way you want. Although the syntax of regular expressions can be intimidating at first, they provide a compact and dynamic language that can be employed to solve all sorts of `String` processing, matching and selection, editing, and verification problems in a completely general way.

Basics

A regular expression is a way to describe strings in general terms, so you can say, "If a string has these things in it, it matches my search criteria." For example, to say that a number might or might not be preceded by a minus sign, you put in the minus sign followed by a question mark, like this:

```
-?
```

To describe an integer, you say it's one or more digits. In regular expressions, a digit is described by saying `\d`. If you have any experience with regular expressions in other languages, you'll immediately notice a difference in the way backslashes are handled. In other languages, `\\` means "insert a plain old (literal) backslash in the regular expression. Don't give it any special meaning." In Java, `\\` means "I'm inserting a regular expression backslash, so the following character has special meaning." For example, to indicate a digit, your regular expression string is `\\d`. To insert a literal backslash, you say `\\\`. However, things like newlines and tabs just use a single backslash: `\n\t`.[136]

To indicate "one or more of the preceding expression," you use a `+`. So to say, "possibly a minus sign, followed by one or more digits," you write:

```
-?\\d+
```

The simplest way to use regular expressions is to use the functionality built into the `String` class. For example, we can see whether a `String` matches the regular expression above:

[136]Java wasn't designed from the beginning for regular expressions, so this awkward syntax was all they could shoehorn in.

```
// strings/IntegerMatch.java

public class IntegerMatch {
  public static void main(String[] args) {
    System.out.println("-1234".matches("-?\\d+"));
    System.out.println("5678".matches("-?\\d+"));
    System.out.println("+911".matches("-?\\d+"));
    System.out.println("+911".matches("(-|\\+)?\\d+"));
  }
}
/* Output:
true
true
false
true
*/
```

The first two expressions match, but the third one starts with a +, a legitimate sign but then the number doesn't match the regular expression. So we need a way to say, "can start with a + or a -." In regular expressions, parentheses have the effect of grouping an expression, and the vertical bar | means OR. So

```
(-|\\+)?
```

means this part of the `String` can be either a - or a + or nothing (because of the ?). Because the + character has special meaning in regular expressions, it must be escaped with a \\ to appear as an ordinary character in the expression.

A useful regular expression tool that's built into `String` is `split()`, which means, "Split this `String` around matches of the given regular expression."

```
// strings/Splitting.java
import java.util.*;

public class Splitting {
  public static String knights =
    "Then, when you have found the shrubbery, " +
    "you must cut down the mightiest tree in the " +
    "forest...with... a herring!";
  public static void split(String regex) {
    System.out.println(
      Arrays.toString(knights.split(regex)));
  }
  public static void main(String[] args) {
    split(" "); // Doesn't have to contain regex chars
    split("\\W+"); // Non-word characters
    split("n\\W+"); // 'n' followed by non-words
  }
}
/* Output:
[Then,, when, you, have, found, the, shrubbery,, you,
must, cut, down, the, mightiest, tree, in, the,
forest...with..., a, herring!]
[Then, when, you, have, found, the, shrubbery, you,
must, cut, down, the, mightiest, tree, in, the, forest,
with, a, herring]
[The, whe, you have found the shrubbery, you must cut
dow, the mightiest tree i, the forest...with... a
herring!]
*/
```

First, note you can use ordinary characters as regular expressions—a regular expression doesn't have to contain special characters, as shown in the first call to `split()`, which just splits on whitespace.

The second and third calls to `split()` use `\\W`, which means a non-word character (the lowercase version, `\\w`, means a word character)—the punctuation is removed in the second case. The third call to `split()` says, "the letter `n` followed by one or more non-word characters." The split patterns do not appear in the result.

An overloaded version of `String.split()` limits the number of splits that occur.

With regular expression replacement, you can either replace the first occurrence, or

all of them:

```
// strings/Replacing.java

public class Replacing {
  static String s = Splitting.knights;
  public static void main(String[] args) {
    System.out.println(
      s.replaceFirst("f\\w+", "located"));
    System.out.println(
      s.replaceAll("shrubbery|tree|herring","banana"));
  }
}
/* Output:
Then, when you have located the shrubbery, you must cut
down the mightiest tree in the forest...with... a
herring!
Then, when you have found the banana, you must cut down
the mightiest banana in the forest...with... a banana!
*/
```

The first expression matches the letter `f` followed by one or more word characters (note that the `w` is lowercase this time). It only replaces the first match it finds, so the word "found" is replaced by the word "located."

The second expression matches any of the three words separated by the OR vertical bars, and it replaces all matches it finds.

You'll see that the non-`String` regular expressions have more powerful replacement tools—for example, you can call methods to perform replacements. Non-`String` regular expressions are also significantly more efficient if you use the regular expression more than once.

Creating Regular Expressions

You can begin learning with a subset of the possible constructs for building regular expressions. A complete list is found in the JDK documentation for the `Pattern` class for package `java.util.regex`.

Item	Produces
`B`	The specific character `B`
`\xhh`	Character with hex value `0xhh`
`\uhhhh`	The Unicode character with hex representation `0xhhhh`
`\t`	Tab
`\n`	Newline
`\r`	Carriage return
`\f`	Form feed
`\e`	Escape

The power of regular expressions begins to appear when you are defining character classes. Here are some typical ways to create character classes, and some predefined classes:

Notation	Result
.	Any character
`[abc]`	Any of the characters `a`, `b`, or `c` (same as `a|b|c`)
`[^abc]`	Any character except `a`, `b`, or `c` (negation)
`[a-zA-Z]`	Any character `a` through `z` or `A` through `Z` (range)
`[abc[hij]]`	Any of `a`, `b`, `c`, `h`, `i`, `j` (same as `a|b|c|h|i|j`) (union)
`[a-z&&[hij]]`	Either `h`, `i`, or `j` (intersection)
`\s`	A whitespace character (space, tab, newline, form feed, carriage return)
`\S`	A non-whitespace character (`[^\s]`)
`\d`	A numeric digit (`[0-9]`)
`\D`	A non-digit (`[^0-9]`)

Notation	Result
`\w`	A word character (`[a-zA-Z_0-9]`)
`\W`	A non-word character (`[^\w]`)

What's shown here is only a sample; bookmark the JDK documentation page for `java.util.regex.Pattern` to easily access all the possible regular expression patterns.

Logical Operator	Meaning
`XY`	`X` followed by `Y`
`X|Y`	`X` or `Y`
`(X)`	A *capturing group*. You can refer to the *i*th captured group later in the expression with `\i`.

Here are the different boundary matchers:

Boundary Matcher	Meaning
`^`	Beginning of a line
`$`	End of a line
`\b`	Word boundary
`\B`	Non-word boundary
`\G`	End of the previous match

As an example, each of the following successfully matches the character sequence "Rudolph":

```
// strings/Rudolph.java

public class Rudolph {
  public static void main(String[] args) {
    for(String pattern : new String[]{
      "Rudolph",
      "[rR]udolph",
      "[rR][aeiou][a-z]ol.*",
      "R.*" })
      System.out.println("Rudolph".matches(pattern));
  }
}
/* Output:
true
true
true
true
*/
```

Your goal should not be to create the most obfuscated regular expression, but rather the simplest one necessary to do the job. You'll find that you'll often use your old code as a reference when writing new regular expressions.

Quantifiers

A *quantifier* describes the way that a pattern absorbs input text:

Greedy
: Quantifiers are greedy unless otherwise altered. A greedy expression finds as many possible matches for the pattern as possible. A typical cause of problems is to assume that your pattern will only match the first possible group of characters, when it's actually greedy and will keep going until it matches the largest possible `String`.

Reluctant
: Specified with a question mark, this quantifier matches the minimum number of characters necessary to satisfy the pattern. Also called *lazy*, *minimal matching*, *non-greedy*, or *ungreedy*.

Possessive
: Currently this is only available in Java (not in other languages) and is more

advanced, so you probably won't use it right away. As a regular expression is applied to a `String`, it generates many states so it can backtrack if the match fails. Possessive quantifiers do not keep those intermediate states, and thus prevent backtracking. They can prevent a regular expression from running away and also to make it execute more efficiently.

Greedy	Reluctant	Possessive	Matches
`X?`	`X??`	`X?+`	`X`, one or none
`X*`	`X*?`	`X*+`	`X`, zero or more
`X+`	`X+?`	`X++`	`X`, one or more
`X{n}`	`X{n}?`	`X{n}+`	`X`, exactly `n` times
`X{n,}`	`X{n,}?`	`X{n,}+`	`X`, at least `n` times
`X{n,m}`	`X{n,m}?`	`X{n,m}+`	`X`, at least `n` but not more than `m` times

Keep in mind that the expression `X` must often be surrounded in parentheses for it to work the way you desire. For example:

```
abc+
```

might seem like it would match the sequence `abc` one or more times, and if you apply it to the input `String` `abcabcabc`, you will in fact get three matches. However, the expression *actually* says, "Match `ab` followed by one or more occurrences of `c`." To match the entire `String` `abc` one or more times, you must say:

```
(abc)+
```

You can easily be fooled when using regular expressions; it's an orthogonal language, on top of Java.

`CharSequence`

The interface called `CharSequence` establishes a generalized definition of a character sequence abstracted from the `CharBuffer`, `String`, `StringBuffer`, or `StringBuilder` classes:

```
interface CharSequence {
  char charAt(int i);
  int length();
  CharSequence subSequence(int start, int end);
  String toString();
}
```

The aforementioned classes implement this interface. Many regular expression operations take `CharSequence` arguments.

`Pattern` and `Matcher`

You'll usually compile regular expression objects rather than using the fairly limited `String` utilities. To do this, import `java.util.regex`, then compile a regular expression with the `static Pattern.compile()` method. This produces a `Pattern` object based on its `String` argument. You use this `Pattern` by calling the `matcher()` method, passing the `String` to search. The `matcher()` method produces a `Matcher` object, which has a set of operations to choose from (All of these appear in the JDK documentation for `java.util.regex.Matcher`). For example, `replaceAll()` replaces all matches with its argument.

As a first example, the following class tests regular expressions against an input `String`. The first command-line argument is the input `String` to match against, followed by one or more regular expressions applied to the input. Under Unix/Linux, the regular expressions must be quoted on the command line. This program can be useful in testing regular expressions as you construct them, to see that they produce your intended matching behavior.[137]

[137]There are far more useful and sophisticated regular expression helper tools on the Internet.

```
// strings/TestRegularExpression.java
// Simple regular expression demonstration
// {java TestRegularExpression
// abcabcabcdefabc "abc+" "(abc)+" }
import java.util.regex.*;

public class TestRegularExpression {
  public static void main(String[] args) {
    if(args.length < 2) {
      System.out.println(
        "Usage:\njava TestRegularExpression " +
        "characterSequence regularExpression+");
      System.exit(0);
    }
    System.out.println("Input: \"" + args[0] + "\"");
    for(String arg : args) {
      System.out.println(
        "Regular expression: \"" + arg + "\"");
      Pattern p = Pattern.compile(arg);
      Matcher m = p.matcher(args[0]);
      while(m.find()) {
        System.out.println(
          "Match \"" + m.group() + "\" at positions " +
          m.start() + "-" + (m.end() - 1));
      }
    }
  }
}
/* Output:
Input: "abcabcabcdefabc"
Regular expression: "abcabcabcdefabc"
Match "abcabcabcdefabc" at positions 0-14
Regular expression: "abc+"
Match "abc" at positions 0-2
Match "abc" at positions 3-5
Match "abc" at positions 6-8
Match "abc" at positions 12-14
Regular expression: "(abc)+"
Match "abcabcabc" at positions 0-8
Match "abc" at positions 12-14
*/
```

Also try adding `"(abc){2,}"` to the command line.

A `Pattern` object represents the compiled version of a regular expression. As seen in the preceding example, you can use the `matcher()` method and the input `String` to produce a `Matcher` object from the compiled `Pattern` object. `Pattern` also has a `static` method:

```
static boolean matches(String regex, CharSequence input)
```

This checks whether `regex` matches the entire `input` `CharSequence`. There's also a `split()` method that produces an array of `String` broken around matches of the `regex`.

A `Matcher` object is generated by calling `Pattern.matcher()` with the input `String` as an argument. The `Matcher` object is then used to access the results with methods to evaluate the success or failure of different types of matches:

```
boolean matches()
boolean lookingAt()
boolean find()
boolean find(int start)
```

The `matches()` method is successful if the pattern matches the entire input `String`, while `lookingAt()` is successful if the input `String`, starting at the beginning, is a match to the pattern.

`find()`

`Matcher.find()` discovers multiple pattern matches in the `CharSequence` to which it is applied. For example:

```
// strings/Finding.java
import java.util.regex.*;

public class Finding {
  public static void main(String[] args) {
    Matcher m = Pattern.compile("\\w+")
      .matcher(
        "Evening is full of the linnet's wings");
    while(m.find())
      System.out.print(m.group() + " ");
    System.out.println();
    int i = 0;
    while(m.find(i)) {
      System.out.print(m.group() + " ");
      i++;
    }
  }
}
/* Output:
Evening is full of the linnet s wings
Evening vening ening ning ing ng g is is s full full
ull ll l of of f the the he e linnet linnet innet nnet
net et t s s wings wings ings ngs gs s
*/
```

The pattern `\\w+` splits the input into words. `find()` is like an iterator, moving forward through the input `String`. However, the second version of `find()` can be given an integer argument that tells it the character position for the beginning of the search—this version resets the search position to the value of the argument, as shown in the output.

Groups

Groups are regular expressions set off by parentheses that can be called up later with their group number. Group `0` indicates the whole expression match, group `1` is the first parenthesized group, etc. Thus, in

```
A(B(C))D
```

there are three groups: Group `0` is `ABCD`, group `1` is `BC`, and group `2` is `C`.

The `Matcher` object has methods to give you information about groups:

`public int groupCount()` returns the number of groups in this matcher's pattern. Group 0 is not included in this count.

`public String group()` returns group 0 (the entire match) from the previous match operation (`find()`, for example).

`public String group(int i)` returns the given group number during the previous match operation. If the match was successful, but the group specified failed to match any part of the input `String`, `null` is returned.

`public int start(int group)` returns the start index of the group found in the previous match operation.

`public int end(int group)` returns the index of the last character, plus one, of the group found in the previous match operation.

Here's an example:

```
// strings/Groups.java
import java.util.regex.*;

public class Groups {
  public static final String POEM =
    "Twas brillig, and the slithy toves\n" +
    "Did gyre and gimble in the wabe.\n" +
    "All mimsy were the borogoves,\n" +
    "And the mome raths outgrabe.\n\n" +
    "Beware the Jabberwock, my son,\n" +
    "The jaws that bite, the claws that catch.\n" +
    "Beware the Jubjub bird, and shun\n" +
    "The frumious Bandersnatch.";
  public static void main(String[] args) {
    Matcher m = Pattern.compile(
      "(?m)(\\S+)\\s+((\\S+)\\s+(\\S+))$")
      .matcher(POEM);
    while(m.find()) {
      for(int j = 0; j <= m.groupCount(); j++)
        System.out.print("[" + m.group(j) + "]");
      System.out.println();
    }
  }
```

```
}
/* Output:
[the slithy toves][the][slithy toves][slithy][toves]
[in the wabe.][in][the wabe.][the][wabe.]
[were the borogoves,][were][the
borogoves,][the][borogoves,]
[mome raths outgrabe.][mome][raths
outgrabe.][raths][outgrabe.]
[Jabberwock, my son,][Jabberwock,][my son,][my][son,]
[claws that catch.][claws][that catch.][that][catch.]
[bird, and shun][bird,][and shun][and][shun]
[The frumious Bandersnatch.][The][frumious
Bandersnatch.][frumious][Bandersnatch.]
*/
```

The poem is the first part of Lewis Carroll's "Jabberwocky," from *Through the Looking Glass*. The regular expression pattern has several parenthesized groups, consisting of any number of non-whitespace characters (`\\S+`) followed by any number of whitespace characters (`\\s+`). The goal is to capture the last three words on each line; the end of a line is delimited by `\$`. However, the normal behavior is to match `\$` with the end of the entire input sequence, so you must explicitly tell the regular expression to pay attention to newlines within the input. This is accomplished with the `(?m)` pattern flag at the beginning of the sequence (pattern flags are shown shortly).

`start()` and `end()`

Following a successful matching operation, `start()` returns the start index of the previous match, and `end()` returns the index of the last character matched, plus one. Invoking either `start()` or `end()` following an unsuccessful matching operation (or before attempting a matching operation) produces an `IllegalStateException`. The following program also demonstrates `matches()` and `lookingAt()`:[138]

[138] `input` is from one of Commander Taggart's speeches in Galaxy Quest.

```
// strings/StartEnd.java
import java.util.regex.*;

public class StartEnd {
  public static String input =
    "As long as there is injustice, whenever a\n" +
    "Targathian baby cries out, wherever a distress\n" +
    "signal sounds among the stars " +
    "... We'll be there.\n"+
    "This fine ship, and this fine crew ...\n" +
    "Never give up! Never surrender!";
  private static class Display {
    private boolean regexPrinted = false;
    private String regex;
    Display(String regex) { this.regex = regex; }
    void display(String message) {
      if(!regexPrinted) {
        System.out.println(regex);
        regexPrinted = true;
      }
      System.out.println(message);
    }
  }
  static void examine(String s, String regex) {
    Display d = new Display(regex);
    Pattern p = Pattern.compile(regex);
    Matcher m = p.matcher(s);
    while(m.find())
      d.display("find() '" + m.group() +
        "' start = "+ m.start() + " end = " + m.end());
    if(m.lookingAt()) // No reset() necessary
      d.display("lookingAt() start = "
        + m.start() + " end = " + m.end());
    if(m.matches()) // No reset() necessary
      d.display("matches() start = "
        + m.start() + " end = " + m.end());
  }
  public static void main(String[] args) {
    for(String in : input.split("\n")) {
      System.out.println("input : " + in);
      for(String regex : new String[]{"\\w*ere\\w*",
        "\\w*ever", "T\\w+", "Never.*?!"})
```

```
        examine(in, regex);
    }
  }
}
/* Output:
input : As long as there is injustice, whenever a
\w*ere\w*
find() 'there' start = 11 end = 16
\w*ever
find() 'whenever' start = 31 end = 39
input : Targathian baby cries out, wherever a distress
\w*ere\w*
find() 'wherever' start = 27 end = 35
\w*ever
find() 'wherever' start = 27 end = 35
T\w+
find() 'Targathian' start = 0 end = 10
lookingAt() start = 0 end = 10
input : signal sounds among the stars ... We'll be
there.
\w*ere\w*
find() 'there' start = 43 end = 48
input : This fine ship, and this fine crew ...
T\w+
find() 'This' start = 0 end = 4
lookingAt() start = 0 end = 4
input : Never give up! Never surrender!
\w*ever
find() 'Never' start = 0 end = 5
find() 'Never' start = 15 end = 20
lookingAt() start = 0 end = 5
Never.*?!
find() 'Never give up!' start = 0 end = 14
find() 'Never surrender!' start = 15 end = 31
lookingAt() start = 0 end = 14
matches() start = 0 end = 31
*/
```

`find()` will locate the regular expression anywhere in the input, but `lookingAt()` and `matches()` only succeed if the regular expression starts matching at the very beginning of the input. While `matches()` only succeeds if the *entire* input matches the regular expression, `lookingAt()` succeeds if only the first part of the input

matches. (I have no idea how they came up with the method name `lookingAt()`, or to what it refers. This is just one reason that code reviews are important).

`Pattern` Flags

An alternative `compile()` method accepts flags that affect matching behavior:

```
Pattern Pattern.compile(String regex, int flag)
```

where `flag` is drawn from among the following `Pattern` class constants:

Compile Flag	Effect
`Pattern.CANON_EQ`	Two characters are considered to match if, and only if, their full canonical decompositions match. The expression `\u003F`, for example, will match the `String` `?` when this flag is specified. By default, matching does not take canonical equivalence into account.
`Pattern.CASE_INSENSITIVE (?i)`	By default, case-insensitive matching assumes that only characters in the US-ASCII character set are matched. This flag allows your pattern to match without regard to case (upper or lower). Unicode-aware case-insensitive matching can be enabled by specifying the `UNICODE_CASE` flag in conjunction with this flag.
`Pattern.COMMENTS (?x)`	In this mode, whitespace is ignored, and embedded comments starting with `#` are ignored until the end of a line. Unix lines mode can also be enabled via the embedded flag expression.
`Pattern.DOTALL (?s)`	In dotall mode, the expression `.` matches any character, including a line terminator. By default, the `.` expression does not match line terminators.

Compile Flag	Effect
`Pattern.MULTILINE (?m)`	In multiline mode, the expressions `^` and `$` match the beginning and ending of a line, respectively. `^` also matches the beginning of the input `String`, and `$` also matches the end of the input `String`. By default, these expressions only match at the beginning and the end of the entire input `String`.
`Pattern.UNICODE_CASE (?u)`	Case-insensitive matching, when enabled by the `CASE_INSENSITIVE` flag, is done in a manner consistent with the Unicode Standard. By default, case-insensitive matching assumes that only characters in the US-ASCII character set are matched.
`Pattern.UNIX_LINES (?d)`	In this mode, only the `\n` line terminator is recognized in the behavior of `.`, `^`, and `$`.

Particularly useful among these flags are:

- `Pattern.CASE_INSENSITIVE`
- `Pattern.MULTILINE`
- `Pattern.COMMENTS` (helpful for clarity and/or documentation).

Note that the behavior of most of the flags can also be obtained by inserting the parenthesized characters, shown after the flags in the table, into your regular expression preceding the place where you want the mode to take effect.

You can combine the effect of these and other flags through an "OR" (`|`) operation:

```
// strings/ReFlags.java
import java.util.regex.*;

public class ReFlags {
  public static void main(String[] args) {
    Pattern p =  Pattern.compile("^java",
      Pattern.CASE_INSENSITIVE | Pattern.MULTILINE);
    Matcher m = p.matcher(
      "java has regex\nJava has regex\n" +
      "JAVA has pretty good regular expressions\n" +
      "Regular expressions are in Java");
    while(m.find())
      System.out.println(m.group());
  }
}
/* Output:
java
Java
JAVA
*/
```

This creates a pattern that matches lines starting with "java," "Java," "JAVA," etc., and attempts a match for lines within a multiline set (matches starting at the beginning of the character sequence and following each line terminator within the character sequence). Note that the `group()` method only produces the matched portion.

`split()`

`split()` divides an input `String` into an array of `String` objects, delimited by the regular expression.

```
String[] split(CharSequence input)
String[] split(CharSequence input, int limit)
```

This is a handy way to break input text on a common boundary:

```
// strings/SplitDemo.java
import java.util.regex.*;
import java.util.*;

public class SplitDemo {
  public static void main(String[] args) {
    String input =
      "This!!unusual use!!of exclamation!!points";
    System.out.println(Arrays.toString(
      Pattern.compile("!!").split(input)));
    // Only do the first three:
    System.out.println(Arrays.toString(
      Pattern.compile("!!").split(input, 3)));
  }
}
/* Output:
[This, unusual use, of exclamation, points]
[This, unusual use, of exclamation!!points]
*/
```

The second form of `split()` limits the number of splits that occur.

Replace Operations

Regular expressions are especially useful to replace text. Here are the available methods:

`replaceFirst(String replacement)` replaces the first matching part of the input `String` with `replacement`.

`replaceAll(String replacement)` replaces every matching part of the input `String` with `replacement`.

`appendReplacement(StringBuffer sbuf, String replacement)` performs step-by-step replacements into `sbuf`, rather than replacing only the first one or all of them, as in `replaceFirst()` and `replaceAll()`, respectively. This is a *very* important method, because you can call methods and perform other processing to produce `replacement` (`replaceFirst()` and `replaceAll()` are only able to put in fixed `Strings`). With this method, you can programmatically pick apart the groups and create powerful replacements.

`appendTail(StringBuffer sbuf)` is invoked after one or more invocations of the `appendReplacement()` method to copy the remainder of the input `String`.

Here's an example that shows all the replace operations. The block of commented text at the beginning is extracted and processed with regular expressions for use as input in the rest of the example:

```
// strings/TheReplacements.java
import java.util.regex.*;
import java.nio.file.*;
import java.util.stream.*;

/*! Here's a block of text to use as input to
    the regular expression matcher. Note that we
    first extract the block of text by looking for
    the special delimiters, then process the
    extracted block. !*/

public class TheReplacements {
  public static void
  main(String[] args) throws Exception {
    String s = Files.lines(
      Paths.get("TheReplacements.java"))
      .collect(Collectors.joining("\n"));
    // Match specially commented block of text above:
    Matcher mInput = Pattern.compile(
      "/\\*!(.*)!\\*/", Pattern.DOTALL).matcher(s);
    if(mInput.find())
      s = mInput.group(1); // Captured by parentheses
    // Replace two or more spaces with a single space:
    s = s.replaceAll(" {2,}", " ");
    // Replace 1+ spaces at the beginning of each
    // line with no spaces. Must enable MULTILINE mode:
    s = s.replaceAll("(?m)^ +", "");
    System.out.println(s);
    s = s.replaceFirst("[aeiou]", "(VOWEL1)");
    StringBuffer sbuf = new StringBuffer();
    Pattern p = Pattern.compile("[aeiou]");
    Matcher m = p.matcher(s);
    // Process the find information as you
    // perform the replacements:
    while(m.find())
```

```
      m.appendReplacement(
        sbuf, m.group().toUpperCase());
    // Put in the remainder of the text:
    m.appendTail(sbuf);
    System.out.println(sbuf);
  }
}
/* Output:
Here's a block of text to use as input to
the regular expression matcher. Note that we
first extract the block of text by looking for
the special delimiters, then process the
extracted block.
H(VOWEL1)rE's A blOck Of tExt tO UsE As InpUt tO
thE rEgUlAr ExprEssIOn mAtchEr. NOtE thAt wE
fIrst ExtrAct thE blOck Of tExt by lOOkIng fOr
thE spEcIAl dElImItErs, thEn prOcEss thE
ExtrActEd blOck.
*/
```

The file is opened and read using the `Files` class introduced in the Files chapter. `Files.lines()` produces a `Stream` of lines, and `Collectors.joining()` combines them into a single `String`, attaching the argument to the end of each line.

`mInput` matches all the text (notice the grouping parentheses) between `/*!` and `!*/`. Then, more than two spaces are reduced to a single space, and any space at the beginning of each line is removed (to do this on all lines and not just the beginning of the input, multiline mode must be enabled). These two replacements are performed with the equivalent (but more convenient, in this case) `replaceAll()` that's part of `String`. Note that because each replacement is only used once in the program, there's no extra cost to doing it this way rather than precompiling it as a `Pattern`.

`replaceFirst()` only performs the first replacement it finds. In addition, the replacement `Strings` in `replaceFirst()` and `replaceAll()` are just literals, so when performing some processing on each replacement, they don't help. In that case, use `appendReplacement()` to write any amount of code in the process of performing the replacement. In the preceding example, a `group()` is selected and processed—in this situation, setting the vowel found by the regular expression to uppercase—as the resulting `sbuf` is built. Normally, you step through and perform all the replacements, then call `appendTail()`, but to simulate `replaceFirst()` (or "replace n"), you only

do the replacement one time, then call `appendTail()` to put the rest into `sbuf`.

`appendReplacement()` also refers to captured groups directly in the replacement `String` by saying `\$g`, where `g` is the group number. However, this is for simpler processing and wouldn't give you the desired results in the preceding program.

`reset()`

An existing `Matcher` object can be applied to a new character sequence using the `reset()` methods:

```
// strings/Resetting.java
import java.util.regex.*;

public class Resetting {
  public static void
  main(String[] args) throws Exception {
    Matcher m = Pattern.compile("[frb][aiu][gx]")
      .matcher("fix the rug with bags");
    while(m.find())
      System.out.print(m.group() + " ");
    System.out.println();
    m.reset("fix the rig with rags");
    while(m.find())
      System.out.print(m.group() + " ");
  }
}
/* Output:
fix rug bag
fix rig rag
*/
```

`reset()` without any arguments sets the `Matcher` to the beginning of the current sequence.

Regular Expressions and Java I/O

Most of the examples so far have shown regular expressions applied to static `Strings`. The following example shows one way to apply regular expressions to search for

matches in a file. Inspired by Unix's *grep*, `JGrep.java` takes two arguments: a file name and the regular expression to match. The output shows each line where a match occurs and the match position(s) within the line.

```
// strings/JGrep.java
// A very simple version of the "grep" program
// {java JGrep
// WhitherStringBuilder.java 'return|for|String'}
import java.util.regex.*;
import java.nio.file.*;
import java.util.stream.*;

public class JGrep {
  public static void
  main(String[] args) throws Exception {
    if(args.length < 2) {
      System.out.println(
        "Usage: java JGrep file regex");
      System.exit(0);
    }
    Pattern p = Pattern.compile(args[1]);
    // Iterate through the lines of the input file:
    int index = 0;
    Matcher m = p.matcher("");
    for(String line :
        Files.readAllLines(Paths.get(args[0]))) {
      m.reset(line);
      while(m.find())
        System.out.println(index++ + ": " +
          m.group() + ": " + m.start());
    }
  }
}
/* Output:
0: for: 4
1: for: 4
*/
```

`Files.readAllLines()` produces a `List<String>`, which means the *for-in* can iterate through it.

Although it's possible to create a new `Matcher` object within the `for` loop, it is slightly more optimal to create an empty `Matcher` object outside the loop and use

the `reset()` method to assign each line of the input to the `Matcher`. The result is scanned with `find()`.

The test arguments open the `JGrep.java` file to read as input, and search for words starting with `[Ssct]`.

You can learn much more about regular expressions in *Mastering Regular Expressions, 2nd Edition*, by Jeffrey E. F. Friedl (O'Reilly, 2002). There are also numerous introductions to regular expressions on the Internet, and you can often find helpful information in the documentation for languages like Perl and Python.

Scanning Input

Until now it was relatively painful to read data from a human-readable file or from standard input. The usual solution is to read in a line of text, tokenize it, then use the various parse methods of `Integer`, `Double`, etc., to parse the data:

```
// strings/SimpleRead.java
import java.io.*;

public class SimpleRead {
  public static BufferedReader input =
    new BufferedReader(new StringReader(
    "Sir Robin of Camelot\n22 1.61803"));
  public static void main(String[] args) {
    try {
      System.out.println("What is your name?");
      String name = input.readLine();
      System.out.println(name);
      System.out.println("How old are you? " +
        "What is your favorite double?");
      System.out.println("(input: <age> <double>)");
      String numbers = input.readLine();
      System.out.println(numbers);
      String[] numArray = numbers.split(" ");
      int age = Integer.parseInt(numArray[0]);
      double favorite = Double.parseDouble(numArray[1]);
      System.out.format("Hi %s.%n", name);
      System.out.format("In 5 years you will be %d.%n",
        age + 5);
```

```
      System.out.format("My favorite double is %f.",
        favorite / 2);
    } catch(IOException e) {
      System.err.println("I/O exception");
    }
  }
}
/* Output:
What is your name?
Sir Robin of Camelot
How old are you? What is your favorite double?
(input: <age> <double>)
22 1.61803
Hi Sir Robin of Camelot.
In 5 years you will be 27.
My favorite double is 0.809015.
*/
```

The `input` field uses classes from `java.io` which are described in the Appendix: I/O Streams. A `StringReader` turns a `String` into a readable stream, and this object is used to create a `BufferedReader` because `BufferedReader` has a `readLine()` method. The result is that the `input` object can be read a line at a time, just as if it were standard input from the console.

`readLine()` is used to get the `String` for each line of input. It's fairly straightforward when you want one input for each line of data, but if two input values are on a single line, things get messy—the line must be split so we can parse each input separately. Here, the splitting takes place when creating `numArray`.

The `Scanner` class, added in Java 5, relieves much of the burden of scanning input:

```
// strings/BetterRead.java
import java.util.*;

public class BetterRead {
  public static void main(String[] args) {
    Scanner stdin = new Scanner(SimpleRead.input);
    System.out.println("What is your name?");
    String name = stdin.nextLine();
    System.out.println(name);
    System.out.println(
      "How old are you? What is your favorite double?");
    System.out.println("(input: <age> <double>)");
    int age = stdin.nextInt();
    double favorite = stdin.nextDouble();
    System.out.println(age);
    System.out.println(favorite);
    System.out.format("Hi %s.%n", name);
    System.out.format("In 5 years you will be %d.%n",
      age + 5);
    System.out.format("My favorite double is %f.",
      favorite / 2);
  }
}
/* Output:
What is your name?
Sir Robin of Camelot
How old are you? What is your favorite double?
(input: <age> <double>)
22
1.61803
Hi Sir Robin of Camelot.
In 5 years you will be 27.
My favorite double is 0.809015.
*/
```

The `Scanner` constructor can take just about any kind of input object, including a `File` object, an `InputStream`, a `String`, or in this case a `Readable`, an interface introduced in Java 5 to describe "something that has a `read()` method." The `BufferedReader` from the previous example falls into this category.

With `Scanner`, the input, tokenizing, and parsing are all ensconced in various different kinds of "next" methods. A plain `next()` returns the next `String` token,

and there are "next" methods for all the primitive types (except `char`) as well as for `BigDecimal` and `BigInteger`. All "next" methods *block*, meaning they will return only after a complete data token is available for input. There are also corresponding "hasNext" methods that return `true` if the next input token is of the correct type.

In `BetterRead.java` there is no `try` block for `IOException`. One of the assumptions made by the `Scanner` is that an `IOException` signals the end of input, and so these are swallowed by the `Scanner`. However, the most recent exception is available through the `ioException()` method, so you are able to examine it if necessary.

`Scanner` Delimiters

By default, a `Scanner` splits input tokens along whitespace, but you can also specify your own delimiter pattern in the form of a regular expression:

```
// strings/ScannerDelimiter.java
import java.util.*;

public class ScannerDelimiter {
  public static void main(String[] args) {
    Scanner scanner = new Scanner("12, 42, 78, 99, 42");
    scanner.useDelimiter("\\s*,\\s*");
    while(scanner.hasNextInt())
      System.out.println(scanner.nextInt());
  }
}
/* Output:
12
42
78
99
42
*/
```

This example uses commas (surrounded by arbitrary amounts of whitespace) as the delimiter when reading from the given `String`. This same technique can read from comma-delimited files. In addition to `useDelimiter()` for setting the delimiter pattern, there is also `delimiter()`, which returns the current `Pattern` being used as a delimiter.

Scanning with Regular Expressions

In addition to scanning for predefined primitive types, you can also scan for your own user-defined patterns, helpful when scanning more complex data. This example scans threat data from a log that your firewall might produce:

```
// strings/ThreatAnalyzer.java
import java.util.regex.*;
import java.util.*;

public class ThreatAnalyzer {
  static String threatData =
    "58.27.82.161@08/10/2015\n" +
    "204.45.234.40@08/11/2015\n" +
    "58.27.82.161@08/11/2015\n" +
    "58.27.82.161@08/12/2015\n" +
    "58.27.82.161@08/12/2015\n" +
    "[Next log section with different data format]";
  public static void main(String[] args) {
    Scanner scanner = new Scanner(threatData);
    String pattern = "(\\d+[.]\\d+[.]\\d+[.]\\d+)@" +
      "(\\d{2}/\\d{2}/\\d{4})";
    while(scanner.hasNext(pattern)) {
      scanner.next(pattern);
      MatchResult match = scanner.match();
      String ip = match.group(1);
      String date = match.group(2);
      System.out.format(
        "Threat on %s from %s%n", date,ip);
    }
  }
}
/* Output:
Threat on 08/10/2015 from 58.27.82.161
Threat on 08/11/2015 from 204.45.234.40
Threat on 08/11/2015 from 58.27.82.161
Threat on 08/12/2015 from 58.27.82.161
Threat on 08/12/2015 from 58.27.82.161
*/
```

When you use `next()` with a specific pattern, that pattern is matched against the

next input token. The result is made available by the `match()` method, and as you see above, it works just like the regular expression matching you saw earlier.

There's one caveat when scanning with regular expressions. The pattern is matched against the next input token only, so if your pattern contains a delimiter it will never be matched.

`StringTokenizer`

Before regular expressions (in Java 1.4) or the `Scanner` class (in Java 5), the way to split a `String` into parts was to "tokenize" it with `StringTokenizer`. But now it's much easier and more succinct to do the same thing with regular expressions or the `Scanner` class. Here's a simple comparison of `StringTokenizer` to the other two techniques:

```
// strings/ReplacingStringTokenizer.java
import java.util.*;

public class ReplacingStringTokenizer {
  public static void main(String[] args) {
    String input =
      "But I'm not dead yet! I feel happy!";
    StringTokenizer stoke = new StringTokenizer(input);
    while(stoke.hasMoreElements())
      System.out.print(stoke.nextToken() + " ");
    System.out.println();
    System.out.println(
      Arrays.toString(input.split(" ")));
    Scanner scanner = new Scanner(input);
    while(scanner.hasNext())
      System.out.print(scanner.next() + " ");
  }
}
/* Output:
But I'm not dead yet! I feel happy!
[But, I'm, not, dead, yet!, I, feel, happy!]
But I'm not dead yet! I feel happy!
*/
```

With regular expressions or `Scanner` objects, you can also split a `String` into parts using more complex patterns—something that's difficult with `StringTokenizer`. It seems safe to say that `StringTokenizer` is obsolete.

Summary

In the beginning, Java support for `String` manipulation was rudimentary, but in later editions of the language we saw far more sophisticated support adopted from other languages. Now the support for `Strings` is reasonably complete, although you must sometimes pay attention to efficiency details such as the appropriate use of `StringBuilder`.

Reflection

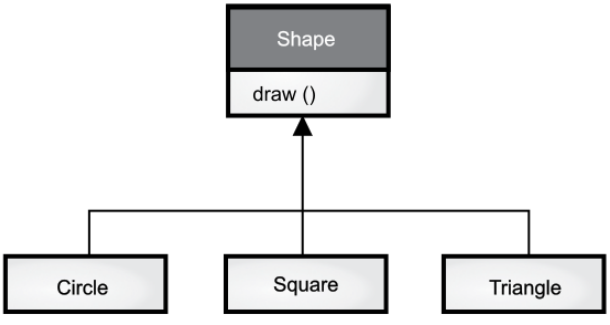
> Reflection uses information about an object's type, which it discovers while the program is running.

It frees you from the constraint of doing type-oriented things only at compile time, and can enable some very powerful programs. The need for reflection uncovers a plethora of interesting (and often perplexing) OO design issues, and raises fundamental questions about how to structure your programs.

This chapter looks at the ways that Java discovers information about objects and classes at run time. This takes two forms: simple reflection, which assumes you have all the types available at compile time, and more complex reflection which discovers and uses class information solely at run time.

The Need for Reflection

Consider the now-familiar example of a class hierarchy that uses polymorphism. The generic type is the base class `Shape`, and the specific derived types are `Circle`, `Square`, and `Triangle`:



This is a typical class hierarchy diagram, with the base class at the top and the derived classes growing downward. A common goal in object-oriented programming is to

write code that manipulates references to the base type (Shape, in this case), so if you decide to extend the program by adding a new class (such as Rhomboid, derived from Shape), the bulk of the code is not affected. In this example, the dynamically bound method in the Shape interface is draw(), so the intent is for the client programmer to call draw() through a generic Shape reference. In all derived classes, draw() is overridden, and because it is a dynamically bound method, the proper behavior will occur even though it is called through a generic Shape reference. That's polymorphism.

Thus, you generally create a specific object (Circle, Square, or Triangle), upcast it to a Shape (forgetting the specific type of the object), and use that anonymous Shape reference in the rest of the program.

You might code the Shape hierarchy like this:

```
// reflection/Shapes.java
import java.util.stream.*;

abstract class Shape {
  void draw() {
    System.out.println(this + ".draw()");
  }
  @Override public abstract String toString();
}

class Circle extends Shape {
  @Override public String toString() {
    return "Circle";
  }
}

class Square extends Shape {
  @Override public String toString() {
    return "Square";
  }
}

class Triangle extends Shape {
  @Override public String toString() {
    return "Triangle";
  }
```

```
}

public class Shapes {
  public static void main(String[] args) {
    Stream.of(
      new Circle(), new Square(), new Triangle())
      .forEach(Shape::draw);
  }
}
/* Output:
Circle.draw()
Square.draw()
Triangle.draw()
*/
```

The base class contains a `draw()` method that indirectly uses `toString()` to display an identifier for the class by passing `this` to `System.out.println()` (`toString()` is declared `abstract` to force inheritors to override it, and to prevent the instantiation of a plain `Shape`). If an object appears in a `String` concatenation expression (involving + and `String` objects), the `toString()` method is automatically called to produce a `String` representation for that object. Each of the derived classes overrides the `toString()` method (from `Object`) so `draw()` ends up (polymorphically) printing something different in each case.

In this example, the upcast occurs when the shape is placed into the (implicit) `Stream<Shape>`. During the upcast to `Shape`, the fact that the objects are *specific types* of `Shape` is lost. To the stream, they are just `Shapes`.

Technically, the `Stream<Shape>` is actually holding everything as an `Object`. When an element emerges, it is automatically cast back to a `Shape`. This is the most basic form of reflection, because all casts are checked at run time for correctness. That's what reflection means: At run time, the type of an object is identified.

Here, the reflection cast is only partial: The `Object` is cast to a `Shape`, and not all the way to a `Circle`, `Square`, or `Triangle`. That's because the only thing you *know* is that the `List<Shape>` is full of `Shapes`. At compile time, this is enforced by the stream and the Java generic system, but at run time the cast ensures it.

Now polymorphism takes over and the exact code that's executed for the `Shape` is determined by whether the reference is for a `Circle`, `Square`, or `Triangle`. And

in general, this is how it should be; you want the bulk of your code to know as little as possible about *specific* types of objects, and to just deal with the general representation of a family of objects (in this case, `Shape`). As a result, your code is easier to write, read, and maintain, and your designs are easier to implement, understand, and change. So polymorphism is a general goal in object-oriented programming.

But what about a special programming problem that's easiest to solve if you know the exact type of a generic reference? For example, suppose you allow your users to highlight all the shapes of any particular type by turning them a special color. This way, they can find all the triangles on the screen by highlighting them. Or perhaps your method must "rotate" a list of shapes, but it makes no sense to rotate a circle so you'd like to skip the circles. With reflection, you can ask a `Shape` reference the exact type it's referring to, and thus select and isolate special cases.

The `Class` Object

To understand how reflection works in Java, you must first know how type information is represented at run time. This is accomplished through a special kind of object called the *Class object*, which contains information about the class. In fact, the `Class` object is used to create all "regular" objects of your class. Java performs reflection using the `Class` object, even if you're doing something like a cast. The class `Class` also has a number of other ways you can use reflection.

There's one `Class` object for each class that is part of your program. That is, each time you write and compile a new class, a single `Class` object is also created (and stored, appropriately enough, in an identically named `.class` file). To make an object of that class, the Java Virtual Machine (JVM) uses a subsystem called a *class loader*.

The class loader subsystem can actually contain a chain of class loaders, but there's only one *primordial class loader*, part of the JVM implementation. The primordial class loader loads so-called *trusted classes*, including Java API classes, typically from the local disk. Additional class loaders in the chain are usually not necessary, but for special needs (such as loading classes in a certain way to support Web server applications, or downloading classes across a network), you can hook in additional class loaders.

All classes are loaded into the JVM dynamically, upon the first use of a class. This happens when the program makes the first reference to a `static` member of that class. The constructor is a `static` method of a class, even though the `static` keyword is not used for a constructor. Therefore, creating a new object of that class using the `new` operator also counts as a reference to a `static` member of the class, and so causes that class to be loaded upon the first use of the constructor.

Thus, a Java program isn't completely loaded before it begins, but instead pieces of it are loaded when necessary. This is different from many traditional languages. Dynamic loading enables behavior that is difficult or impossible to duplicate in a statically-loaded language like C++.

The class loader first checks to see if the `Class` object for that type is loaded. If not, the default class loader locates the `.class` file with that name (an add-on class loader might, for example, look for the bytecodes in a database instead). As the bytes for that class are loaded, they are *verified* to ensure they have not been corrupted and they do not comprise bad Java code (this is one of the lines of defense for security in Java).

Once the `Class` object for that type is in memory, it is used to create all objects of that type. Here's a program to prove it:

```
// reflection/SweetShop.java
// Examination of the way the class loader works

class Cookie {
  static { System.out.println("Loading Cookie"); }
}

class Gum {
  static { System.out.println("Loading Gum"); }
}

class Candy {
  static { System.out.println("Loading Candy"); }
}

public class SweetShop {
  public static void main(String[] args) {
    System.out.println("inside main");
    new Candy();
```

```
    System.out.println("After creating Candy");
    try {
      Class.forName("Gum");
    } catch(ClassNotFoundException e) {
      System.out.println("Couldn't find Gum");
    }
    System.out.println("After Class.forName(\"Gum\")");
    new Cookie();
    System.out.println("After creating Cookie");
  }
}
/* Output:
inside main
Loading Candy
After creating Candy
Loading Gum
After Class.forName("Gum")
Loading Cookie
After creating Cookie
*/
```

Each of the classes `Candy`, `Gum`, and `Cookie` has a `static` clause that is executed as the class is loaded for the first time. Information is displayed to tell you when loading occurs for that class. In `main()`, the object creations are spread out between print statements, to help detect the time of loading.

The output shows that each `Class` object is loaded only when it's needed, and the `static` initialization is performed upon class loading.

A particularly interesting line is:

```
Class.forName("Gum");
```

All `Class` objects belong to the class `Class`. A `Class` object is like any other object, so you can get and manipulate a reference to it (that's what the loader does). One of the ways to get a reference to the `Class` object is the `static forName()` method, which takes a `String` containing the textual name (watch the spelling and capitalization!) of the your desired class. It returns a `Class` reference, which is ignored here; the call to `forName()` is made here for its side effect: to load the class `Gum` if it isn't already loaded. In the process of loading, `Gum`'s `static` clause is executed.

In the preceding example, if `Class.forName()` fails because it can't find the class you're trying to load, it will throw a `ClassNotFoundException`. Here, we simply report the problem and move on, but in more sophisticated programs, you might try to fix the problem inside the exception handler (there's an example of this in the Patterns chapter).

Anytime you use type information at run time, you must first get a reference to the appropriate `Class` object. `Class.forName()` is one convenient way to do this, because you don't need an object of that type to get the `Class` reference. However, if you already have an object of the type you're interested in, you can fetch the `Class` reference by calling a method that's part of the `Object` root class: `getClass()`. This returns the `Class` reference representing the actual type of the object. `Class` has many methods; here are a few:

```
// reflection/toys/ToyTest.java
// Testing class Class
// {java reflection.toys.ToyTest}
package reflection.toys;
import java.lang.reflect.InvocationTargetException;

interface HasBatteries {}
interface Waterproof {}
interface Shoots {}

class Toy {
  // Comment out the following zero-argument
  // constructor to see NoSuchMethodError
  public Toy() {}
  public Toy(int i) {}
}

class FancyToy extends Toy
implements HasBatteries, Waterproof, Shoots {
  public FancyToy() { super(1); }
}

public class ToyTest {
  static void printInfo(Class cc) {
    System.out.println("Class name: " + cc.getName() +
      " is interface? [" + cc.isInterface() + "]");
    System.out.println(
```

```
      "Simple name: " + cc.getSimpleName());
    System.out.println(
      "Canonical name : " + cc.getCanonicalName());
  }
  @SuppressWarnings("deprecation")
  public static void main(String[] args) {
    Class c = null;
    try {
      c = Class.forName("reflection.toys.FancyToy");
    } catch(ClassNotFoundException e) {
      System.out.println("Can't find FancyToy");
      System.exit(1);
    }
    printInfo(c);
    for(Class face : c.getInterfaces())
      printInfo(face);
    Class up = c.getSuperclass();
    Object obj = null;
    try {
      // Requires public zero-argument constructor:
      obj = up.newInstance();
    } catch(Exception e) {
      throw new
        RuntimeException("Cannot instantiate");
    }
    printInfo(obj.getClass());
  }
}
/* Output:
Class name: reflection.toys.FancyToy is interface?
[false]
Simple name: FancyToy
Canonical name : reflection.toys.FancyToy
Class name: reflection.toys.HasBatteries is interface?
[true]
Simple name: HasBatteries
Canonical name : reflection.toys.HasBatteries
Class name: reflection.toys.Waterproof is interface?
[true]
Simple name: Waterproof
Canonical name : reflection.toys.Waterproof
Class name: reflection.toys.Shoots is interface? [true]
```

```
Simple name: Shoots
Canonical name : reflection.toys.Shoots
Class name: reflection.toys.Toy is interface? [false]
Simple name: Toy
Canonical name : reflection.toys.Toy
*/
```

`FancyToy` inherits from `Toy` and `implements` the interfaces `HasBatteries`, `Waterproof`, and `Shoots`. In `main()`, a `Class` reference is created and initialized to the `FancyToy` `Class` using `forName()` inside an appropriate `try` block. Notice you must use the fully qualified name (including the package name) in the `String` you pass to `forName()`.

`printInfo()` uses `getName()` to produce the fully qualified class name, and `getSimpleName()` and `getCanonicalName()` to produce the name without the package, and the fully qualified name, respectively. As its name implies, `isInterface()` tells you whether this `Class` object represents an interface. Thus, with the `Class` object you can find out just about everything you want to know about a type.

The `Class.getInterfaces()` method called in `main()` returns an array of `Class` objects representing the interfaces contained in the `Class` object of interest.

You can also ask a `Class` object for its direct base class using `getSuperclass()`. This returns a `Class` reference you can further query. Thus you can discover an object's entire class hierarchy at run time.

The `newInstance()` method of `Class` is a way to implement a "virtual constructor," which says, "I don't know exactly what type you are, but create yourself properly anyway." In the preceding example, `up` is just a `Class` reference with no further type information known at compile time. When you create a new instance, you get back an `Object` reference. But that reference is pointing to a `Toy` object. Before you can send any messages other than those accepted by `Object`, you must investigate it a bit and do some casting. In addition, the class that's created with `Class.newInstance()` must have a zero-argument constructor. Later in this chapter, you'll see how to dynamically create objects of classes using any constructor, with the Java reflection API.

Note that `newInstance()` as shown in this example is fine in Java 8, but is deprecated in later versions of Java, in favor of `Constructor.newInstance()`. The

deprecation warning is suppressed for those later versions using `@SuppressWarnings("deprecation")`.

Class Literals

Java provides a second way to produce a reference to a `Class` object: the *class literal*. In the preceding program this would look like:

```
FancyToy.class;
```

This is simpler and also safer because it's checked at compile time (and thus does not have to be placed in a `try` block). Because it eliminates the `forName()` method call, it's also more efficient.

Class literals work with regular classes as well as interfaces, arrays, and primitive types. In addition, there's a standard field called `TYPE` that exists for each of the primitive wrapper classes. The `TYPE` field produces a reference to the `Class` object for the associated primitive type, such that:

Class literal	is equivalent to
`boolean.class`	`Boolean.TYPE`
`char.class`	`Character.TYPE`
`byte.class`	`Byte.TYPE`
`short.class`	`Short.TYPE`
`int.class`	`Integer.TYPE`
`long.class`	`Long.TYPE`
`float.class`	`Float.TYPE`
`double.class`	`Double.TYPE`
`void.class`	`Void.TYPE`

My preference is to use the "`.class`" versions if you can because they're more

consistent with regular classes.

Notice that creating a reference to a `Class` object using "`.class`" doesn't automatically initialize the `Class` object. There are actually three steps in preparing a class for use:

1. *Loading*, performed by the class loader. This finds the bytecodes (usually, but not necessarily, on your disk in your classpath) and creates a `Class` object from those bytecodes.
2. *Linking*. The link phase verifies the bytecodes in the class, allocates storage for `static` fields, and if necessary, resolves all references to other classes made by this class.
3. *Initialization*. If there's a superclass, initialize that. Execute `static` initializers and `static` initialization blocks.

Initialization is delayed until the first reference to a `static` method (the constructor is implicitly `static`) or to a non-constant `static` field:

```
// reflection/ClassInitialization.java
import java.util.*;

class Initable {
  static final int STATIC_FINAL = 47;
  static final int STATIC_FINAL2 =
    ClassInitialization.rand.nextInt(1000);
  static {
    System.out.println("Initializing Initable");
  }
}

class Initable2 {
  static int staticNonFinal = 147;
  static {
    System.out.println("Initializing Initable2");
  }
}

class Initable3 {
  static int staticNonFinal = 74;
  static {
    System.out.println("Initializing Initable3");
```

```
  }
}

public class ClassInitialization {
  public static Random rand = new Random(47);
  public static void
  main(String[] args) throws Exception {
    Class initable = Initable.class;
    System.out.println("After creating Initable ref");
    // Does not trigger initialization:
    System.out.println(Initable.STATIC_FINAL);
    // Triggers initialization:
    System.out.println(Initable.STATIC_FINAL2);
    // Triggers initialization:
    System.out.println(Initable2.staticNonFinal);
    Class initable3 = Class.forName("Initable3");
    System.out.println("After creating Initable3 ref");
    System.out.println(Initable3.staticNonFinal);
  }
}
/* Output:
After creating Initable ref
47
Initializing Initable
258
Initializing Initable2
147
Initializing Initable3
After creating Initable3 ref
74
*/
```

Effectively, initialization is "as lazy as possible." From the creation of the `initable` reference, you see that just using the `.class` syntax to get a reference to the class doesn't cause initialization. However, `Class.forName()` initializes the class immediately to produce the `Class` reference, as seen in the creation of `initable3`.

If a `static final` value is a "compile-time constant" such as `Initable.staticFinal`, that value can be read without causing the `Initable` class to be initialized. Making a field `static` and `final`, however, does not guarantee this behavior: accessing `Initable.staticFinal2` forces class initialization because it cannot be a compile-

time constant.

If a `static` field is not `final`, accessing it always requires linking (to allocate storage for the field) and initialization (to initialize that storage) before it can be read, as you see in the access to `Initable2.staticNonFinal`.

Generic Class References

A `Class` reference points to a `Class` object, which manufactures instances of classes and contains all the method code for those instances. It also contains the `statics` for that class. So a `Class` reference really does indicate the exact type of what it's pointing to: an object of the class `Class`.

You can constrain the type of a `Class` reference using generic syntax. In the following example, both syntaxes are correct:

```
// reflection/GenericClassReferences.java

public class GenericClassReferences {
  public static void main(String[] args) {
    Class intClass = int.class;
    intClass = double.class;
    Class<Integer> genericIntClass = int.class;
    genericIntClass = Integer.class; // Same thing
    // genericIntClass = double.class; // Illegal
  }
}
```

`intClass` can be reassigned to any other `Class` object such as `double.class` without producing a warning. The generic class reference `genericIntClass` can only be assigned to its declared type. The generic syntax allows the compiler to enforce extra type checking.

What if you'd like to loosen the constraint a little? Initially, it seems like you could do something like:

```
Class<Number> genericNumberClass = int.class;
```

This would seem to make sense because `Integer` inherits `Number`. But this doesn't work, because the `Integer Class` object is not a subclass of the `Number Class`

object (this can seem like a subtle distinction; we'll look into it more deeply in the Generics chapter).

To loosen the constraints when using generic `Class` references, use a *wildcard*, which is part of Java generics. The wildcard symbol is `?`, and it indicates "anything." So we can add wildcards to the ordinary `Class` reference in the above example and produce the same results:

```
// reflection/WildcardClassReferences.java

public class WildcardClassReferences {
  public static void main(String[] args) {
    Class<?> intClass = int.class;
    intClass = double.class;
  }
}
```

`Class<?>` is preferred over plain `Class`, even though they are equivalent and the plain `Class`, as you saw, doesn't produce a compiler warning. The benefit of `Class<?>` is it indicates you aren't just using a non-specific class reference by accident, or out of ignorance. You *chose* the non-specific version.

To create a `Class` reference that is constrained to a type *or any subtype*, you combine the wildcard with the `extends` keyword to create a *bound*. So instead of just saying `Class<Number>`, you say:

```
// reflection/BoundedClassReferences.java

public class BoundedClassReferences {
  public static void main(String[] args) {
    Class<? extends Number> bounded = int.class;
    bounded = double.class;
    bounded = Number.class;
    // Or anything else derived from Number.
  }
}
```

One reason for adding the generic syntax to `Class` references is to provide compile-time type checking, so if you do something wrong you find out about it a little sooner. You can't actually go astray with ordinary `Class` references, but if you make a mistake you won't find out until run time, which can be inconvenient or problematic.

Here's an example that uses the generic class syntax. It stores a class reference, and later generates objects using `newInstance()`:

```
// reflection/DynamicSupplier.java
import java.util.function.*;
import java.util.stream.*;

class ID {
  private static long counter;
  private final long id = counter++;
  @Override public String toString() {
    return Long.toString(id);
  }
  // A public default constructor is required
  // to call getConstructor().newInstance():
  public ID() {}
}

public class DynamicSupplier<T> implements Supplier<T> {
  private Class<T> type;
  public DynamicSupplier(Class<T> type) {
    this.type = type;
  }
  @Override public T get() {
    try {
      return type.getConstructor().newInstance();
    } catch(Exception e) {
      throw new RuntimeException(e);
    }
  }
  public static void main(String[] args) {
    Stream.generate(
      new DynamicSupplier<>(ID.class))
      .skip(10)
      .limit(5)
      .forEach(System.out::println);
  }
}
/* Output:
10
11
12
```

```
13
14
*/
```

`DynamicSupplier` must assume that any type it works with has a `public` zero-argument constructor (one without arguments), and you'll get an exception if that isn't the case. In the above example, the automatically-generated zero-argument constructor for `ID` is not `public` because `class ID` is not `public`, so we must explicitly define it. In the following, `class ID2` is `public` so the automatically-generated zero-argument constructor is thus also `public`, so we don't need to explicitly define it:

```
// reflection/ID2.java
import java.util.stream.*;

public class ID2 {
  private static long counter;
  private final long id = counter++;
  @Override public String toString() {
    return Long.toString(id);
  }
  public static void main(String[] args) {
    Stream.generate(
      new DynamicSupplier<>(ID2.class))
      .skip(10)
      .limit(5)
      .forEach(System.out::println);
  }
}
/* Output:
10
11
12
13
14
*/
```

When you use the generic syntax for `Class` objects, `newInstance()` returns the exact type of the object, rather than just a basic `Object` as you saw in `ToyTest.java`. This is somewhat limited:

```
// reflection/toys/GenericToyTest.java
// Testing class Class
// {java reflection.toys.GenericToyTest}
package reflection.toys;

public class GenericToyTest {
  public static void
  main(String[] args) throws Exception {
    Class<FancyToy> ftc = FancyToy.class;
    // Produces exact type:
    FancyToy fancyToy =
      ftc.getConstructor().newInstance();
    Class<? super FancyToy> up = ftc.getSuperclass();
    // This won't compile:
    // Class<Toy> up2 = ftc.getSuperclass();
    // Only produces Object:
    Object obj = up.getConstructor().newInstance();
  }
}
```

If you get the superclass, the compiler will only allow you to say that the superclass reference is "some class that is a superclass of `FancyToy`" as seen in the expression `Class<? super FancyToy>`. It will not accept a declaration of `Class<Toy>`. This seems a bit strange because `getSuperclass()` returns the base *class* (not interface) and the compiler knows what that class is at compile time—here, `Toy.class`, not just "some superclass of `FancyToy`." In any event, because of the vagueness, the return value of `up.getConstructor().newInstance()` is not a precise type, but just an `Object`.

The `cast()` Method

There's also a casting syntax for use with `Class` references: the `cast()` method:

```
// reflection/ClassCasts.java

class Building {}
class House extends Building {}

public class ClassCasts {
  public static void main(String[] args) {
    Building b = new House();
    Class<House> houseType = House.class;
    House h = houseType.cast(b);
    h = (House)b; // ... or just do this.
  }
}
```

The `cast()` method takes the argument object and casts it to the type of the `Class` reference. However, if you look at the above code it seems like a lot of extra work compared to the last line in `main()`, which does the same thing.

`cast()` is useful for situations where you *can't* use an ordinary cast. This usually happens when you're writing generic code (which you'll learn about in the Generics chapter), and you've stored a `Class` reference to use for casting. This turns out to be a rare thing—I found only one instance where `cast()` was used in the entire Java library (it was in `com.sun.mirror.util.DeclarationFilter`).

Another feature had *no* usage in the Java library: `Class.asSubclass()`. This casts the class object to a more specific type.

Checking Before a Cast

So far, you've seen:

1. The classic cast; e.g., "`(Shape)`," which uses reflection to ensure the cast is correct. This will throw a `ClassCastException` if you've performed a bad cast.
2. The `Class` object representing the type of your object. The `Class` object can be queried for useful runtime information.

In C++, the classic cast "`(Shape)`" does *not* perform reflection (in C++ it's called *run-time type identification* (RTTI)). It simply tells the compiler to treat the object as the

new type. In Java, which *does* perform the type check, this cast is often called a "type-safe downcast." The reason for the term "downcast" is the historical arrangement of the class hierarchy diagram. If casting a `Circle` to a `Shape` is an upcast, then casting a `Shape` to a `Circle` is a downcast. However, because it knows that a `Circle` is also a `Shape`, the compiler freely allows an upcast assignment, without requiring any extra syntax. The compiler *cannot* know, given a `Shape`, what that `Shape` actually is—it could be exactly a `Shape`, or it could be a subtype of `Shape`, such as a `Circle`, `Square`, `Triangle`, or some other type. At compile time, the compiler only sees a `Shape`. Thus, it won't allow you to perform a downcast assignment without using an explicit cast, to tell the compiler you know this is a particular type (the compiler *will* check to see if that downcast is reasonable, so it won't let you downcast to a type that's not actually a subclass).

There's a third form of reflection in Java. This is the keyword `instanceof`, which tells you whether an object is an instance of a particular type. It returns a `boolean` so you use it in the form of a question, like this:

```
if(x instanceof Dog)
  ((Dog)x).bark();
```

The `if` statement checks to see if the object `x` belongs to the class `Dog` *before* you cast `x` to a `Dog`. It's important to use `instanceof` before a downcast when you don't have other information that tells you the type of the object. Otherwise, you'll get a `ClassCastException`.

Ordinarily, you might be hunting for one type (triangles to turn purple, for example), but you can easily tally *all* objects using `instanceof`. For example, suppose you have a family of classes that describe `Pets` (and their people, a feature that comes in handy in a later example). Each `Individual` in the hierarchy has an `id` and an optional name. Although the following classes inherit from `Individual`, there are some complexities in the `Individual` class, so that code is shown and explained in the Appendix: Understanding `equals()` and `hashCode()`. It's not really necessary to see the code for `Individual`—you know you can create it with or without a name, and that each `Individual` has a method `id()` that returns a unique identifier (created by counting each object). There's also a `toString()` method—if you don't provide a name for an `Individual`, `toString()` only produces the simple type name.

Here is the class hierarchy that inherits from `Individual`:

```
// reflection/pets/Person.java
package reflection.pets;

public class Person extends Individual {
  public Person(String name) { super(name); }
}


// reflection/pets/Pet.java
package reflection.pets;

public class Pet extends Individual {
  public Pet(String name) { super(name); }
  public Pet() { super(); }
}


// reflection/pets/Dog.java
package reflection.pets;

public class Dog extends Pet {
  public Dog(String name) { super(name); }
  public Dog() { super(); }
}


// reflection/pets/Mutt.java
package reflection.pets;

public class Mutt extends Dog {
  public Mutt(String name) { super(name); }
  public Mutt() { super(); }
}


// reflection/pets/Pug.java
package reflection.pets;

public class Pug extends Dog {
  public Pug(String name) { super(name); }
  public Pug() { super(); }
}
```

```
// reflection/pets/Cat.java
package reflection.pets;

public class Cat extends Pet {
  public Cat(String name) { super(name); }
  public Cat() { super(); }
}
```

```
// reflection/pets/EgyptianMau.java
package reflection.pets;

public class EgyptianMau extends Cat {
  public EgyptianMau(String name) { super(name); }
  public EgyptianMau() { super(); }
}
```

```
// reflection/pets/Manx.java
package reflection.pets;

public class Manx extends Cat {
  public Manx(String name) { super(name); }
  public Manx() { super(); }
}
```

```
// reflection/pets/Cymric.java
package reflection.pets;

public class Cymric extends Manx {
  public Cymric(String name) { super(name); }
  public Cymric() { super(); }
}
```

```
// reflection/pets/Rodent.java
package reflection.pets;

public class Rodent extends Pet {
  public Rodent(String name) { super(name); }
  public Rodent() { super(); }
}


// reflection/pets/Rat.java
package reflection.pets;

public class Rat extends Rodent {
  public Rat(String name) { super(name); }
  public Rat() { super(); }
}


// reflection/pets/Mouse.java
package reflection.pets;

public class Mouse extends Rodent {
  public Mouse(String name) { super(name); }
  public Mouse() { super(); }
}


// reflection/pets/Hamster.java
package reflection.pets;

public class Hamster extends Rodent {
  public Hamster(String name) { super(name); }
  public Hamster() { super(); }
}
```

We must explicitly write the zero-argument constructor in each case because every class also has a constructor with an argument, which precludes the compiler automatically generating the zero-argument constructor.

Next, we need a way to create random `Pet` objects. To allow this tool to evolve through different implementations, we'll define it as an abstract class:

```
// reflection/pets/Creator.java
// Creates random Pets
package reflection.pets;
import java.util.*;
import java.util.function.*;
import java.util.stream.*;
import java.lang.reflect.InvocationTargetException;

public abstract class Creator implements Supplier<Pet> {
  private Random rand = new Random(47);
  // The different types of Pet to create:
  public abstract List<Class<? extends Pet>> types();
  @Override public Pet get() { // Create one random Pet
    int n = rand.nextInt(types().size());
    try {
      return types().get(n)
        .getConstructor().newInstance();
    } catch(InstantiationException |
            NoSuchMethodException |
            InvocationTargetException |
            IllegalAccessException e) {
      throw new RuntimeException(e);
    }
  }
  public Stream<Pet> stream() {
    return Stream.generate(this);
  }
  public Pet[] array(int size) {
    return stream().limit(size).toArray(Pet[]::new);
  }
  public List<Pet> list(int size) {
    return stream().limit(size)
      .collect(Collectors.toCollection(ArrayList::new));
  }
}
```

The `abstract types()` method expects any subclass of `Creator` to produce the `List` of `Class` objects (this is an example of the *Template Method* design pattern). Notice that the `List` generic parameter is specified to be "anything derived from `Pet`," so `newInstance()` produces a `Pet` without requiring a cast. `get()` indexes into the `List` to yield a `Class` object. `getConstructor()` produces a `Constructor`

object, and `newInstance()` creates an object using that `Constructor`.

You can get four kinds of exceptions when calling `newInstance()`. You see these handled in the `catch` clause following the `try` block. The names of the exceptions are relatively useful explanations of what went wrong (`IllegalAccessException` indicates a security violation, in this case if the zero-argument constructor is `private`).

`Creator` provides utility methods to create groups of randomly-generated `Pets`: `stream()` produces a `Stream<Pet>`, `array()` produces a `Pet[]`, and `list()` produces a `List<Pet>`.

When deriving a subclass of `Creator`, you must supply the `List` of `Pet` types to return from `get()`. The `types()` method will normally just return a reference to a `static List`. Here's an implementation using `forName()`:

```
// reflection/pets/ForNamePetCreator.java
package reflection.pets;
import java.util.*;

public class ForNamePetCreator extends Creator {
  private static List<Class<? extends Pet>> types =
    new ArrayList<>();
  // Types you want randomly created:
  private static String[] typeNames = {
    "reflection.pets.Mutt",
    "reflection.pets.Pug",
    "reflection.pets.EgyptianMau",
    "reflection.pets.Manx",
    "reflection.pets.Cymric",
    "reflection.pets.Rat",
    "reflection.pets.Mouse",
    "reflection.pets.Hamster"
  };
  @SuppressWarnings("unchecked")
  private static void loader() {
    try {
      for(String name : typeNames)
        types.add(
          (Class<? extends Pet>)Class.forName(name));
    } catch(ClassNotFoundException e) {
      throw new RuntimeException(e);
```

```
    }
  }
  static { loader(); }
  @Override public List<Class<? extends Pet>> types() {
    return types;
  }
}
```

The `loader()` method creates the `List` of `Class` objects using `Class.forName()`. This can generate a `ClassNotFoundException`, which makes sense because you're passing it a `String` which cannot be validated at compile time. The `Pet` objects are in package `reflection` so the package name must be used when referring to the classes.

To produce a typed `List` of `Class` objects, a cast is required, which produces a compile-time warning. The `loader()` method is defined separately, then called in a static initialization clause because the `@SuppressWarnings("unchecked")` annotation cannot be placed directly onto the static initialization clause.

To count `Pets`, we need a tool that keeps track of the quantities of various different types of `Pet`. A `Map` is perfect for this; the keys are the `Pet` type names and the values are `Integers` to hold the `Pet` quantities. This way, you can say, "How many `Hamster` objects are there?" We use `instanceof` to count `Pets`:

```
// reflection/PetCounter.java
// Using instanceof
import reflection.pets.*;
import java.util.*;

public class PetCounter {
  static class Counter extends HashMap<String,Integer> {
    public void count(String type) {
      Integer quantity = get(type);
      if(quantity == null)
        put(type, 1);
      else
        put(type, quantity + 1);
    }
  }
  private Counter counter = new Counter();
  private void countPet(Pet pet) {
```

```
    System.out.print(
      pet.getClass().getSimpleName() + " ");
    if(pet instanceof Pet)
      counter.count("Pet");
    if(pet instanceof Dog)
      counter.count("Dog");
    if(pet instanceof Mutt)
      counter.count("Mutt");
    if(pet instanceof Pug)
      counter.count("Pug");
    if(pet instanceof Cat)
      counter.count("Cat");
    if(pet instanceof EgyptianMau)
      counter.count("EgyptianMau");
    if(pet instanceof Manx)
      counter.count("Manx");
    if(pet instanceof Cymric)
      counter.count("Cymric");
    if(pet instanceof Rodent)
      counter.count("Rodent");
    if(pet instanceof Rat)
      counter.count("Rat");
    if(pet instanceof Mouse)
      counter.count("Mouse");
    if(pet instanceof Hamster)
      counter.count("Hamster");
  }
  public void count(Creator creator) {
    creator.stream().limit(20)
      .forEach(pet -> countPet(pet));
    System.out.println();
    System.out.println(counter);
  }
  public static void main(String[] args) {
    new PetCounter().count(new ForNamePetCreator());
  }
}
/* Output:
Rat Manx Cymric Mutt Pug Cymric Pug Manx Cymric Rat
EgyptianMau Hamster EgyptianMau Mutt Mutt Cymric Mouse
Pug Mouse Cymric
{EgyptianMau=2, Pug=3, Rat=2, Cymric=5, Mouse=2, Cat=9,
```

```
Manx=7, Rodent=5, Mutt=3, Dog=6, Pet=20, Hamster=1}
*/
```

In `countPet()`, each `Pet` in the array is tested and counted using `instanceof`.

There's a rather narrow restriction on `instanceof`: You can compare it to a named type only, and not to a `Class` object. In the preceding example you might think it's tedious to write out all of those `instanceof` expressions, and you're right. But there is no way to cleverly automate `instanceof` by creating an array of `Class` objects and comparing it to those instead (stay tuned—you'll see an alternative). This isn't as great a restriction as you might think, because you'll eventually understand that your design is probably flawed if you end up writing many `instanceof` expressions.

Using Class Literals

If we reimplement `Creator` using class literals, the result is cleaner in several ways:

```
// reflection/pets/PetCreator.java
// Using class literals
// {java reflection.pets.PetCreator}
package reflection.pets;
import java.util.*;

public class PetCreator extends Creator {
  // No try block needed.
  public static final
  List<Class<? extends Pet>> ALL_TYPES =
    Collections.unmodifiableList(Arrays.asList(
      Pet.class, Dog.class, Cat.class, Rodent.class,
      Mutt.class, Pug.class, EgyptianMau.class,
      Manx.class, Cymric.class, Rat.class,
      Mouse.class, Hamster.class));
  // Types for random creation:
  private static final
  List<Class<? extends Pet>> TYPES =
    ALL_TYPES.subList(
      ALL_TYPES.indexOf(Mutt.class),
      ALL_TYPES.size());
  @Override
  public List<Class<? extends Pet>> types() {
```

```
    return TYPES;
  }
  public static void main(String[] args) {
    System.out.println(TYPES);
    List<Pet> pets = new PetCreator().list(7);
    System.out.println(pets);
  }
}
/* Output:
[class reflection.pets.Mutt, class reflection.pets.Pug,
class reflection.pets.EgyptianMau, class
reflection.pets.Manx, class reflection.pets.Cymric, class
reflection.pets.Rat, class reflection.pets.Mouse, class
reflection.pets.Hamster]
[Rat, Manx, Cymric, Mutt, Pug, Cymric, Pug]
*/
```

This time, the creation of `types` is not surrounded by a `try` block because it's evaluated at compile time and thus won't throw any exceptions, unlike `Class.forName()`.

In the upcoming `PetCounter3.java` example, we pre-load a `Map` with all the `Pet` types (not just the ones that are randomly generated), so the `ALL_TYPES List` is necessary. The `types` list is the portion of `ALL_TYPES` (created using `List.subList()`) that includes the exact pet types, so it is used for random `Pet` generation.

Because `PetCounter.count()` takes a `Creator` argument, we can easily test `PetCreator`:

```
// reflection/PetCounter2.java
import reflection.pets.*;

public class PetCounter2 {
  public static void main(String[] args) {
    new PetCounter().count(new PetCreator());
  }
}
/* Output:
Rat Manx Cymric Mutt Pug Cymric Pug Manx Cymric Rat
EgyptianMau Hamster EgyptianMau Mutt Mutt Cymric Mouse
Pug Mouse Cymric
{EgyptianMau=2, Pug=3, Rat=2, Cymric=5, Mouse=2, Cat=9,
Manx=7, Rodent=5, Mutt=3, Dog=6, Pet=20, Hamster=1}
*/
```

The output is the same as that of `PetCounter.java`.

A Dynamic `instanceof`

The `Class.isInstance()` method provides a way to dynamically test the type of an object. Thus, all those tedious `instanceof` statements can be removed from `PetCounter.java`:

```
// reflection/PetCounter3.java
// Using isInstance()
import java.util.*;
import java.util.stream.*;
import onjava.*;
import reflection.pets.*;

public class PetCounter3 {
  static class Counter extends
  HashMap<Class<? extends Pet>, Integer> {
    Counter() {
      super(PetCreator.ALL_TYPES.stream()
        .map(type -> Pair.make(type, 0))
        .collect(
          Collectors.toMap(Pair::key, Pair::value)));
    }
    public void count(Pet pet) {
      // Class.isInstance() eliminates instanceofs:
      entrySet().stream()
        .filter(pair -> pair.getKey().isInstance(pet))
        .forEach(pair ->
          put(pair.getKey(), pair.getValue() + 1));
    }
    @Override public String toString() {
      String result = entrySet().stream()
        .map(pair -> String.format("%s=%s",
          pair.getKey().getSimpleName(),
          pair.getValue()))
        .collect(Collectors.joining(", "));
      return "{" + result + "}";
    }
  }
  public static void main(String[] args) {
```

```
    Counter petCount = new Counter();
    new PetCreator().stream()
      .limit(20)
      .peek(petCount::count)
      .forEach(p -> System.out.print(
        p.getClass().getSimpleName() + " "));
    System.out.println("\n" + petCount);
  }
}
/* Output:
Rat Manx Cymric Mutt Pug Cymric Pug Manx Cymric Rat
EgyptianMau Hamster EgyptianMau Mutt Mutt Cymric Mouse
Pug Mouse Cymric
{EgyptianMau=2, Mouse=2, Pet=20, Cymric=5, Rat=2,
Dog=6, Mutt=3, Hamster=1, Cat=9, Manx=7, Rodent=5,
Pug=3}
*/
```

To count all the different types of `Pet`, `Counter` inherits `HashMap` and is pre-loaded with the types from `PetCreator.ALL_TYPES`. Without pre-loading the `Map`, you would only end up counting the randomly-generated types, not the base types like `Pet` and `Cat`.

The `isInstance()` method eliminates the need for the `instanceof` expressions. In addition, this means you can add new types of `Pet` by changing the `PetCreator.types` array; the rest of the program does not need modification (as it did when using the `instanceof` expressions).

The `toString()` method is overloaded for easier-to-read output that still matches the typical output you see when displaying a `Map`.

Counting Recursively

The `Map` in `PetCounter3.Counter` was pre-loaded with all the different `Pet` classes. Instead of pre-loading the map, we can use `Class.isAssignableFrom()` and create a general-purpose tool that is not limited to counting `Pets`:

```
// onjava/TypeCounter.java
// Counts instances of a type family
package onjava;
import java.util.*;
import java.util.stream.*;

public class
TypeCounter extends HashMap<Class<?>, Integer> {
  private Class<?> baseType;
  public TypeCounter(Class<?> baseType) {
    this.baseType = baseType;
  }
  public void count(Object obj) {
    Class<?> type = obj.getClass();
    if(!baseType.isAssignableFrom(type))
      throw new RuntimeException(
        obj + " incorrect type: " + type +
        ", should be type or subtype of " + baseType);
    countClass(type);
  }
  private void countClass(Class<?> type) {
    Integer quantity = get(type);
    put(type, quantity == null ? 1 : quantity + 1);
    Class<?> superClass = type.getSuperclass();
    if(superClass != null &&
       baseType.isAssignableFrom(superClass)) {
      countClass(superClass);
    }
  }
  @Override public String toString() {
    String result = entrySet().stream()
      .map(pair -> String.format("%s=%s",
        pair.getKey().getSimpleName(),
        pair.getValue()))
      .collect(Collectors.joining(", "));
    return "{" + result + "}";
  }
}
```

The `count()` method gets the `Class` of its argument, and uses `isAssignableFrom()` to verify at runtime that the object you passed actually belongs to the hierarchy of interest. `countClass()` first counts the exact type of the class. Then, if `baseType` is

assignable from the superclass, `countClass()` is called recursively on the superclass.

```
// reflection/PetCounter4.java
import reflection.pets.*;
import onjava.*;

public class PetCounter4 {
  public static void main(String[] args) {
    TypeCounter counter = new TypeCounter(Pet.class);
    new PetCreator().stream()
      .limit(20)
      .peek(counter::count)
      .forEach(p -> System.out.print(
        p.getClass().getSimpleName() + " "));
    System.out.println("\n" + counter);
  }
}
/* Output:
Rat Manx Cymric Mutt Pug Cymric Pug Manx Cymric Rat
EgyptianMau Hamster EgyptianMau Mutt Mutt Cymric Mouse
Pug Mouse Cymric
{Rodent=5, Mouse=2, Hamster=1, Cymric=5, Dog=6,
EgyptianMau=2, Pet=20, Rat=2, Pug=3, Manx=7, Cat=9,
Mutt=3}
*/
```

The output shows that both base types as well as exact types are counted.

Registered Factories

A problem with generating objects from the `Pet` hierarchy is the fact that every time you add a new type of `Pet` to the hierarchy you must remember to add it to the entries in `PetCreator.java`. In a system where you add more classes on a regular basis this can become problematic.

You might think of adding a static initializer to each subclass, so the initializer would add its class to a list somewhere. Unfortunately, static initializers are only called when the class is first loaded, so you have a chicken-and-egg problem: The generator doesn't have the class in its list, so it can never create an object of that class, so the class won't get loaded and placed in the list.

Basically, you're forced to create the list yourself, by hand (unless you write a tool that searches through and analyzes your source code, then creates and compiles the list). So the best you can probably do is to put the list in one central, obvious place. The base class for the hierarchy of interest is probably the best place.

The other change we'll make here is to defer the creation of the object to the class itself, using the *Factory Method* design pattern. A factory method can be called polymorphically, and creates an object of the appropriate type for you. It turns out that `java.util.function.Supplier` describes the prototypical factory method with its `T get()`. Covariant return types allow `get()` to return a different type for each subclass implementation of `Supplier`.

In this example, the base class `Part` contains a `static List` of factory objects (`Supplier<Part>`). Factories for types that should be produced by the `get()` method are "registered" with the base class by adding them to the `prototypes List`. These factories are, oddly enough, instances of the objects themselves. Each object in this list is a *Prototype* for creating other objects:

```
// reflection/RegisteredFactories.java
// Registering Factories in the base class
import java.util.*;
import java.util.function.*;
import java.util.stream.*;

class Part implements Supplier<Part> {
  @Override public String toString() {
    return getClass().getSimpleName();
  }
  static List<Supplier<? extends Part>> prototypes =
    Arrays.asList(
      new FuelFilter(),
      new AirFilter(),
      new CabinAirFilter(),
      new OilFilter(),
      new FanBelt(),
      new PowerSteeringBelt(),
      new GeneratorBelt()
    );
  private static Random rand = new Random(47);
  @Override public Part get() {
    int n = rand.nextInt(prototypes.size());
```

```
    return prototypes.get(n).get();
  }
}

class Filter extends Part {}

class FuelFilter extends Filter {
  @Override
  public FuelFilter get() { return new FuelFilter(); }
}

class AirFilter extends Filter {
  @Override
  public AirFilter get() { return new AirFilter(); }
}

class CabinAirFilter extends Filter {
  @Override public CabinAirFilter get() {
    return new CabinAirFilter();
  }
}

class OilFilter extends Filter {
  @Override
  public OilFilter get() { return new OilFilter(); }
}

class Belt extends Part {}

class FanBelt extends Belt {
  @Override
  public FanBelt get() { return new FanBelt(); }
}

class GeneratorBelt extends Belt {
  @Override public GeneratorBelt get() {
    return new GeneratorBelt();
  }
}

class PowerSteeringBelt extends Belt {
  @Override public PowerSteeringBelt get() {
```

```
    return new PowerSteeringBelt();
  }
}

public class RegisteredFactories {
  public static void main(String[] args) {
    Stream.generate(new Part())
      .limit(10)
      .forEach(System.out::println);
  }
}
/* Output:
GeneratorBelt
CabinAirFilter
GeneratorBelt
AirFilter
PowerSteeringBelt
CabinAirFilter
FuelFilter
PowerSteeringBelt
PowerSteeringBelt
FuelFilter
*/
```

Not all classes in the hierarchy should be instantiated; here `Filter` and `Belt` are just classifiers so you do not create an instance of either one, but only of their subclasses (if you try, you get the behavior of the `Part` base class).

Because `Part implements Supplier<Part>`, a `Part` supplies other `Parts` via its `get()`. If you call `get()` (or if `generate()` calls `get()`) for a base-class `Part`, it creates random specific `Part` subtypes, each of which are ultimately inherited from `Part` and override the appropriate `get()` to produce one of themselves.

`Instanceof` vs. `Class` Equivalence

When you are querying for type information, there's an important difference between `instanceof` and `isInstance()` (which produce equivalent results) and the direct comparison of the `Class` objects. This example demonstrates the difference:

```
// reflection/FamilyVsExactType.java
// The difference between instanceof and class
// {java reflection.FamilyVsExactType}
package reflection;

class Base {}
class Derived extends Base {}

public class FamilyVsExactType {
  static void test(Object x) {
    System.out.println(
      "Testing x of type " + x.getClass());
    System.out.println(
      "x instanceof Base " + (x instanceof Base));
    System.out.println(
      "x instanceof Derived " + (x instanceof Derived));
    System.out.println(
      "Base.isInstance(x) " + Base.class.isInstance(x));
    System.out.println(
      "Derived.isInstance(x) " +
      Derived.class.isInstance(x));
    System.out.println(
      "x.getClass() == Base.class " +
      (x.getClass() == Base.class));
    System.out.println(
      "x.getClass() == Derived.class " +
      (x.getClass() == Derived.class));
    System.out.println(
      "x.getClass().equals(Base.class)) "+
      (x.getClass().equals(Base.class)));
    System.out.println(
      "x.getClass().equals(Derived.class)) " +
      (x.getClass().equals(Derived.class)));
  }
  public static void main(String[] args) {
    test(new Base());
    test(new Derived());
  }
}
/* Output:
Testing x of type class reflection.Base
x instanceof Base true
```

```
x instanceof Derived false
Base.isInstance(x) true
Derived.isInstance(x) false
x.getClass() == Base.class true
x.getClass() == Derived.class false
x.getClass().equals(Base.class)) true
x.getClass().equals(Derived.class)) false
Testing x of type class reflection.Derived
x instanceof Base true
x instanceof Derived true
Base.isInstance(x) true
Derived.isInstance(x) true
x.getClass() == Base.class false
x.getClass() == Derived.class true
x.getClass().equals(Base.class)) false
x.getClass().equals(Derived.class)) true
*/
```

The `test()` method performs type checking with its argument using both forms of `instanceof`. It then gets the `Class` reference and uses `==` and `equals()` to test for equality of the `Class` objects. Reassuringly, `instanceof` and `isInstance()` produce exactly the same results, as do `equals()` and `==`. But the tests themselves draw different conclusions. In keeping with the concept of type, `instanceof` says, "Are you this class, or a class derived from this class?" On the other hand, if you compare the actual `Class` objects using `==`, there is no concern with inheritance—it's either the exact type or it isn't.

Runtime Class Information

If you don't know the precise type of an object, `instanceof` will tell you. However, there's a limitation: The type must be known at compile time to detect it using `instanceof` and do something useful with the information. Put another way, the compiler must know about all the classes you use.

This doesn't seem like that much of a limitation at first, but suppose you're given a reference to an object that's not in your program space. In fact, the class of that object isn't even available to your program at compile time. Perhaps you get a bunch of bytes from a disk file or from a network connection, and you're told that those

bytes represent a class. This class shows up long after the compiler generates the code for your program so how can you possibly use such a class?

In a traditional programming environment, this is a far-fetched scenario. But as we move into a larger programming world, there are important cases when this happens. The first is component-based programming, where you build projects using *Rapid Application Development* (RAD) in an application builder *Integrated Development Environment* (IDE). This is a visual approach to creating a program by moving icons that represent components onto a form. These components are then configured by setting some of their values at program time. This design-time configuration requires that any component be instantiable, that it exposes parts of itself, and it allows its properties to be read and modified. In addition, components that handle *Graphical User Interface* (GUI) events must expose information about appropriate methods so the IDE can assist the programmer in overriding these event-handling methods. Reflection provides the mechanism to detect the available methods and produce the method names.

Another compelling motivation for discovering class information at run time is to provide the ability to create and execute objects on remote platforms, across a network. This is called *Remote Method Invocation* (RMI), and it enables a program's objects to be distributed across many machines. This distribution can happen for a number of reasons. If you want to speed up a computation-intensive task, you can break it into pieces to put onto idle machines. Or you might place code that handles particular types of tasks (e.g., "Business Rules" in a multitier client/server architecture) on a particular machine, so the machine becomes a common repository describing those actions, and it can be easily changed to affect everyone in the system. Distributed computing also supports specialized hardware that might be good at a particular task—matrix inversions, for example—but inappropriate or too expensive for general-purpose programming.

The class `Class` supports reflection, along with the `java.lang.reflect` library which contains the classes `Field`, `Method`, and `Constructor` (each of which implements the `Member` interface). Objects of these types are created by the JVM at run time to represent the corresponding member in the unknown class. You can then use the `Constructors` to create new objects, the `get()` and `set()` methods to read and modify the fields associated with `Field` objects, and the `invoke()` method to call a method associated with a `Method` object. In addition, you can call the convenience methods `getFields()`, `getMethods()`, `getConstructors()`, etc., to return arrays

of objects representing the fields, methods, and constructors. (You can find out more by looking up the class `Class` in the JDK documentation.) Thus, the class information for anonymous objects can be completely determined at run time, and nothing need be known at compile time.

It's important to realize there's nothing magic about reflection. When you're using reflection to interact with an object of an unknown type, the JVM will look at the object and see it belongs to a particular class. Before anything can be done with it, the `Class` object must be loaded. Thus, the `.class` file for that particular type must still be available to the JVM, either on the local machine or across the network. With reflection, the `.class` file—which is unavailable at compile time—is opened and examined by the runtime environment.

A Class Method Extractor

Normally you won't use the reflection tools directly, but they can be helpful to create more dynamic code. Reflection is in the language to support other Java features, such as object serialization (see Appendix: Object Serialization). However, there are times when it's useful to dynamically extract information about a class.

Consider a class method extractor. Looking at a class definition's source code or JDK documentation shows only the methods defined or overridden *within that class definition*. But there might be dozens more available to you that have come from base classes. To locate these is both tedious and time consuming.[139] Fortunately, reflection provides a way to write a simple tool to automatically show you the entire interface:

```
// reflection/ShowMethods.java
// Using reflection to show all the methods of a class,
// even if the methods are defined in the base class
// {java ShowMethods ShowMethods}
import java.lang.reflect.*;
import java.util.regex.*;

public class ShowMethods {
  private static String usage =
    "usage:\n" +
    "ShowMethods qualified.class.name\n" +
```

[139] Especially in the past. However, great improvements in the HTML Java documentation makes it easier to see base-class methods.

```
    "To show all methods in class or:\n" +
    "ShowMethods qualified.class.name word\n" +
    "To search for methods involving 'word'";
  private static Pattern p = Pattern.compile("\\w+\\.");
  public static void main(String[] args) {
    if(args.length < 1) {
      System.out.println(usage);
      System.exit(0);
    }
    int lines = 0;
    try {
      Class<?> c = Class.forName(args[0]);
      Method[] methods = c.getMethods();
      Constructor[] ctors = c.getConstructors();
      if(args.length == 1) {
        for(Method method : methods)
          System.out.println(
            p.matcher(
              method.toString()).replaceAll(""));
        for(Constructor ctor : ctors)
          System.out.println(
            p.matcher(ctor.toString()).replaceAll(""));
        lines = methods.length + ctors.length;
      } else {
        for(Method method : methods)
          if(method.toString().contains(args[1])) {
            System.out.println(p.matcher(
              method.toString()).replaceAll(""));
            lines++;
          }
        for(Constructor ctor : ctors)
          if(ctor.toString().contains(args[1])) {
            System.out.println(p.matcher(
              ctor.toString()).replaceAll(""));
            lines++;
          }
      }
    } catch(ClassNotFoundException e) {
      System.out.println("No such class: " + e);
    }
  }
}
```

```
/* Output:
public static void main(String[])
public final void wait(long,int) throws
InterruptedException
public final native void wait(long) throws
InterruptedException
public final void wait() throws InterruptedException
public boolean equals(Object)
public String toString()
public native int hashCode()
public final native Class getClass()
public final native void notify()
public final native void notifyAll()
public ShowMethods()
*/
```

The `Class` methods `getMethods()` and `getConstructors()` return an array of `Method` and array of `Constructor`, respectively. Each of these classes has further methods to dissect the names, arguments, and return values of the methods they represent. But you can also just use `toString()`, as is done here, to produce a `String` with the entire method signature. The rest of the code extracts the command-line information, determines if a particular signature matches your target `String` (using `indexOf()`), and strips off the name qualifiers using regular expressions (introduced in the Strings chapter).

The result produced by `Class.forName()` cannot be known at compile time, and therefore all the method signature information is extracted at run time. If you investigate the JDK reflection documentation, you'll see there is enough support to actually set up and make a method call on an object that's totally unknown at compile time (there are examples of this later in this book). Although initially this is something you might not think you'll ever need, the value of full reflection can be surprising.

The output above is produced from the command line:

```
java ShowMethods ShowMethods
```

The output includes a `public` zero-argument constructor, even though no constructor was defined. The constructor you see is the one that's automatically synthesized by the compiler. If you then make `ShowMethods` a non-`public` class (that is, package

access), the synthesized zero-argument constructor no longer shows up in the output. The synthesized zero-argument constructor is automatically given the same access as the class.

Try running `java ShowMethods java.lang.String` with an extra argument of `char`, `int`, `String`, etc.

This tool can be a real time-saver while you're programming, when you can't remember if a class has a particular method and you don't want to hunt through the index or class hierarchy in the JDK documentation, or if you don't know whether that class can do anything with, for example, `Color` objects.

Dynamic Proxies

Proxy is one of the basic design patterns. It is an object you insert in place of the "real" object to provide additional or different operations—these usually involve communication with a "real" object, so a proxy typically acts as a go-between. Here's a trivial example to show the structure of a proxy:

```
// reflection/SimpleProxyDemo.java

interface Interface {
  void doSomething();
  void somethingElse(String arg);
}

class RealObject implements Interface {
  @Override public void doSomething() {
    System.out.println("doSomething");
  }
  @Override public void somethingElse(String arg) {
    System.out.println("somethingElse " + arg);
  }
}

class SimpleProxy implements Interface {
  private Interface proxied;
  SimpleProxy(Interface proxied) {
    this.proxied = proxied;
```

```
  }
  @Override public void doSomething() {
    System.out.println("SimpleProxy doSomething");
    proxied.doSomething();
  }
  @Override public void somethingElse(String arg) {
    System.out.println(
      "SimpleProxy somethingElse " + arg);
    proxied.somethingElse(arg);
  }
}

class SimpleProxyDemo {
  public static void consumer(Interface iface) {
    iface.doSomething();
    iface.somethingElse("bonobo");
  }
  public static void main(String[] args) {
    consumer(new RealObject());
    consumer(new SimpleProxy(new RealObject()));
  }
}
/* Output:
doSomething
somethingElse bonobo
SimpleProxy doSomething
doSomething
SimpleProxy somethingElse bonobo
somethingElse bonobo
*/
```

Because `consumer()` accepts an `Interface`, it can't know if it's getting a `RealObject` or a `SimpleProxy`, because both implement `Interface`. The `SimpleProxy` inserted between the client and the `RealObject` performs operations, then calls the identical method on a `RealObject`.

A proxy can be helpful anytime you'd like to separate extra operations into a different place than the "real object," and especially when you want to easily change from not using the extra operations to using them, and vice versa (the point of design patterns is to encapsulate change—so you must be changing things to justify the pattern). For example, what if you wanted to track calls to the methods in `RealObject`, or to

measure the overhead of such calls? This is not code you want incorporated in your application, and a proxy enables adding and removing it easily.

Java's *dynamic proxy* takes the idea of a proxy one step further, by both creating the proxy object dynamically and handling calls to the proxied methods dynamically. All calls made on a dynamic proxy are redirected to a single *invocation handler*, which has the job of discovering what the call is and deciding what to do about it. Here's `SimpleProxyDemo.java` rewritten to use a dynamic proxy:

```
// reflection/SimpleDynamicProxy.java
import java.lang.reflect.*;

class DynamicProxyHandler implements InvocationHandler {
  private Object proxied;
  DynamicProxyHandler(Object proxied) {
    this.proxied = proxied;
  }
  @Override public Object
  invoke(Object proxy, Method method, Object[] args)
  throws Throwable {
    System.out.println(
      "**** proxy: " + proxy.getClass() +
      ", method: " + method + ", args: " + args);
    if(args != null)
      for(Object arg : args)
        System.out.println("  " + arg);
    return method.invoke(proxied, args);
  }
}

class SimpleDynamicProxy {
  public static void consumer(Interface iface) {
    iface.doSomething();
    iface.somethingElse("bonobo");
  }
  public static void main(String[] args) {
    RealObject real = new RealObject();
    consumer(real);
    // Insert a proxy and call again:
    Interface proxy = (Interface)Proxy.newProxyInstance(
      Interface.class.getClassLoader(),
      new Class[]{ Interface.class },
```

```
      new DynamicProxyHandler(real));
    consumer(proxy);
  }
}
/* Output:
doSomething
somethingElse bonobo
**** proxy: class $Proxy0, method: public abstract void
Interface.doSomething(), args: null
doSomething
**** proxy: class $Proxy0, method: public abstract void
Interface.somethingElse(java.lang.String), args:
[Ljava.lang.Object;@1b84c92
  bonobo
somethingElse bonobo
*/
```

To create a dynamic proxy, call the `static` method `Proxy.newProxyInstance()`. This requires a class loader (you can generally just hand it a class loader from an object that has already been loaded), a list of interfaces (not classes or abstract classes) you wish the proxy to implement, and an implementation of the interface `InvocationHandler`. The dynamic proxy will redirect all calls to the invocation handler, so the constructor for the invocation handler is usually given the reference to the "real" object so it can forward requests once it performs its intermediary task.

The `invoke()` method is handed the proxy object, in case you must distinguish where the request came from (in many cases you won't care). However, be careful when calling methods on the proxy inside `invoke()`, because calls through the interface are redirected through the proxy.

In general you perform the proxied operation, then use `Method.invoke()` to forward the request to the proxied object, passing the necessary arguments. This can initially seem limiting, as if you can only perform generic operations. However, you can filter for certain method calls, while passing others through:

```
// reflection/SelectingMethods.java
// Looking for particular methods in a dynamic proxy
import java.lang.reflect.*;

class MethodSelector implements InvocationHandler {
  private Object proxied;
  MethodSelector(Object proxied) {
    this.proxied = proxied;
  }
  @Override public Object
  invoke(Object proxy, Method method, Object[] args)
  throws Throwable {
    if(method.getName().equals("interesting"))
      System.out.println(
        "Proxy detected the interesting method");
    return method.invoke(proxied, args);
  }
}

interface SomeMethods {
  void boring1();
  void boring2();
  void interesting(String arg);
  void boring3();
}

class Implementation implements SomeMethods {
  @Override public void boring1() {
    System.out.println("boring1");
  }
  @Override public void boring2() {
    System.out.println("boring2");
  }
  @Override public void interesting(String arg) {
    System.out.println("interesting " + arg);
  }
  @Override public void boring3() {
    System.out.println("boring3");
  }
}

class SelectingMethods {
```

```
  public static void main(String[] args) {
    SomeMethods proxy =
      (SomeMethods)Proxy.newProxyInstance(
        SomeMethods.class.getClassLoader(),
        new Class[]{ SomeMethods.class },
        new MethodSelector(new Implementation()));
    proxy.boring1();
    proxy.boring2();
    proxy.interesting("bonobo");
    proxy.boring3();
  }
}
/* Output:
boring1
boring2
Proxy detected the interesting method
interesting bonobo
boring3
*/
```

Here, we are just looking for method names, but you can also look for other aspects of the method signature, and you can even search for particular argument values.

The dynamic proxy is not a tool that you'll use every day, but it can solve certain types of problems very nicely. You can learn more about *Proxy* and other design patterns in *Design Patterns*, by Erich Gamma et al. (Addison-Wesley, 1995), and in the Patterns chapter.

Using Optional

When you use the built-in `null` to indicate the absence of an object, to be completely safe you must test a reference for `null`-ness every time you use it. This can get very tedious and produce ponderous code. The problem is that `null` has no behavior of its own except for producing a `NullPointerException` if you try to do anything with it. `java.util.Optional`, which you first saw in the Functional Programming chapter, creates a thin proxy to shield potentially `null` values. `Optional` objects prevent your code from directly causing a `NullPointerException`.

Although `Optionals` were introduced in Java 8 to support `Streams`, they are a general-purpose tool, which will demonstrate by applying them to ordinary classes. The topic is included in this chapter because it involves run-time detection.

In practice it doesn't make sense to use `Optionals` everywhere—sometimes checking for `null` is fine, and sometimes you can reasonably assume you won't encounter `null`, and sometimes even detecting aberrations via `NullPointerException` is acceptable. The place where `Optionals` seem most useful is "closer to the data," with objects that represent entities in the problem space. As a simple example, many systems have a `Person` class, and there are situations in the code where you don't have an actual person (or perhaps you do, but you don't have all the information about that person yet), so traditionally you'd use a `null` reference and test for it. Instead, we can use `Optional`:

```
// reflection/Person.java
// Using Optional with regular classes
import onjava.*;
import java.util.*;

class Person {
  public final Optional<String> first;
  public final Optional<String> last;
  public final Optional<String> address;
  // etc.
  public final boolean empty;
  Person(String first, String last, String address) {
    this.first = Optional.ofNullable(first);
    this.last = Optional.ofNullable(last);
    this.address = Optional.ofNullable(address);
    empty = !this.first.isPresent()
         && !this.last.isPresent()
         && !this.address.isPresent();
  }
  Person(String first, String last) {
    this(first, last, null);
  }
  Person(String last) { this(null, last, null); }
  Person() { this(null, null, null); }
  @Override public String toString() {
    if(empty)
      return "<Empty>";
```

```
    return (first.orElse("") +
      " " + last.orElse("") +
      " " + address.orElse("")).trim();
  }
  public static void main(String[] args) {
    System.out.println(new Person());
    System.out.println(new Person("Smith"));
    System.out.println(new Person("Bob", "Smith"));
    System.out.println(new Person("Bob", "Smith",
      "11 Degree Lane, Frostbite Falls, MN"));
  }
}
/* Output:
<Empty>
Smith
Bob Smith
Bob Smith 11 Degree Lane, Frostbite Falls, MN
*/
```

The design of `Person` is sometimes called a "data-transfer object." Notice that all the fields are `public` and `final`, so there are no getter and setter methods. That is, `Person` is *immutable*—you can only set the values with the constructor, then read those values, but you can't modify them (`Strings` themselves are inherently immutable, so you can't modify the contents of the `Strings` nor can you re-assign to the fields). To change a `Person`, you can only replace it with a new `Person` object. The `empty` field is set during construction to easily check to see whether this `Person` represents an empty object.

Anyone using a `Person` is forced to use the `Optional` interface when accessing the `String` fields, and so cannot accidentally trigger a `NullPointerException`.

Now suppose you've been given a big pile of venture funding for your Amazing Idea. You're ready to staff up, but while you're waiting for positions to be filled, you can use `Person Optionals` as placeholders for each `Position`:

```
// reflection/Position.java
import java.util.*;

class EmptyTitleException extends RuntimeException {}

class Position {
  private String title;
  private Person person;
  Position(String jobTitle, Person employee) {
    setTitle(jobTitle);
    setPerson(employee);
  }
  Position(String jobTitle) {
    this(jobTitle, null);
  }
  public String getTitle() { return title; }
  public void setTitle(String newTitle) {
    // Throws EmptyTitleException if newTitle is null:
    title = Optional.ofNullable(newTitle)
      .orElseThrow(EmptyTitleException::new);
  }
  public Person getPerson() { return person; }
  public void setPerson(Person newPerson) {
    // Uses empty Person if newPerson is null:
    person = Optional.ofNullable(newPerson)
      .orElse(new Person());
  }
  @Override public String toString() {
    return "Position: " + title +
      ", Employee: " + person;
  }
  public static void main(String[] args) {
    System.out.println(new Position("CEO"));
    System.out.println(new Position("Programmer",
      new Person("Arthur", "Fonzarelli")));
    try {
      new Position(null);
    } catch(Exception e) {
      System.out.println("caught " + e);
    }
  }
}
```

```
/* Output:
Position: CEO, Employee: <Empty>
Position: Programmer, Employee: Arthur Fonzarelli
caught EmptyTitleException
*/
```

This uses `Optional` in a different way. Notice that `title` and `person` are ordinary fields, unprotected by `Optional`. However, the only way to modify those fields is via `setTitle()` and `setPerson()`, both of which use the functionality of `Optional` to impose restrictions on the fields.

We want to guarantee that `title` is never set to `null`. In `setTitle()`, we could check the `newTitle` argument ourselves. But a big part of functional programming is the ability to reuse tried-and-tested functionality, often even if it's small, to reduce the kinds of little programming errors you make when you code everything by hand. So we take `newTitle` and turn it into an `Optional` using `ofNullable()`, which means that if it's `null` it will produce an `Optional.empty()`. We then immediately take that `Optional` result and apply `orElseThrow()`, so if `newTitle` was `null`, you'll get an exception. We never store the field as an `Optional`, but we apply the `Optional` functionality to enforce our desired constraint on the `title` field.

`EmptyTitleException` is a `RuntimeException` because it represents a programmer error. You still get an exception with this scheme, but you get it at the point the error is made—when the `null` is passed to `setTitle()`—and not at some other point in the program which will force you to debug until you find the problem. Also, the use of `EmptyTitleException` further helps localize the bug.

The `person` field has a different constraint: if you try to set it to `null`, it should automatically get set to an empty `Person` object instead. We use the same approach as before of turning it into an `Option`, but in this case when we extract the result we use `orElse(new Person())` to insert the empty `Person` for the `null`.

With `Position`, we don't make an "empty" flag or method because the existence of an empty `Person` in the `person` field implies an available `Position`. Later, you might discover you must add something explicit here, but YAGNI (*You Aren't Going to Need It*)[140] says to "try the simplest thing that could possibly work" for your first draft, and to wait until some aspect of the program requires you to add in the extra feature, rather than assuming it's necessary.

[140] A tenet of *Extreme Programming* (XP), as is "Try the simplest thing that could possibly work."

Notice that the `Staff` class blithely ignores the existence of the `Optional`s, although you know they are there, protecting you from `NullPointerException`s:

```
// reflection/Staff.java
import java.util.*;

public class Staff extends ArrayList<Position> {
  public void add(String title, Person person) {
    add(new Position(title, person));
  }
  public void add(String... titles) {
    for(String title : titles)
      add(new Position(title));
  }
  public Staff(String... titles) { add(titles); }
  public boolean positionAvailable(String title) {
    for(Position position : this)
      if(position.getTitle().equals(title) &&
         position.getPerson().empty)
        return true;
    return false;
  }
  public void fillPosition(String title, Person hire) {
    for(Position position : this)
      if(position.getTitle().equals(title) &&
         position.getPerson().empty) {
        position.setPerson(hire);
        return;
      }
    throw new RuntimeException(
      "Position " + title + " not available");
  }
  public static void main(String[] args) {
    Staff staff = new Staff("President", "CTO",
      "Marketing Manager", "Product Manager",
      "Project Lead", "Software Engineer",
      "Software Engineer", "Software Engineer",
      "Software Engineer", "Test Engineer",
      "Technical Writer");
    staff.fillPosition("President",
      new Person("Me", "Last", "The Top, Lonely At"));
    staff.fillPosition("Project Lead",
```

```
      new Person("Janet", "Planner", "The Burbs"));
    if(staff.positionAvailable("Software Engineer"))
      staff.fillPosition("Software Engineer",
        new Person(
          "Bob", "Coder", "Bright Light City"));
    System.out.println(staff);
  }
}
/* Output:
[Position: President, Employee: Me Last The Top, Lonely
At, Position: CTO, Employee: <Empty>, Position:
Marketing Manager, Employee: <Empty>, Position: Product
Manager, Employee: <Empty>, Position: Project Lead,
Employee: Janet Planner The Burbs, Position: Software
Engineer, Employee: Bob Coder Bright Light City,
Position: Software Engineer, Employee: <Empty>,
Position: Software Engineer, Employee: <Empty>,
Position: Software Engineer, Employee: <Empty>,
Position: Test Engineer, Employee: <Empty>, Position:
Technical Writer, Employee: <Empty>]
*/
```

You might still need to test for `Optionals` in some places, which is not that different from checking for `null`, but in other places (such as `toString()` conversions, in this case), you don't perform extra tests; you can just assume that all object references are valid.

Tagging Interfaces

Sometimes it's more convenient to use a *tagging interface* to indicate `null`-ness. A tagging interface has no elements; you just use its name as a tag:

```
// onjava/Null.java
package onjava;
public interface Null {}
```

If you work with interfaces instead of concrete classes, you can use a `DynamicProxy` to automatically create the `Nulls`. Suppose we have a `Robot` interface that defines a name, model, and a `List<Operation>` that describes what the `Robot` does:

```
// reflection/Robot.java
import onjava.*;
import java.util.*;

public interface Robot {
  String name();
  String model();
  List<Operation> operations();
  static void test(Robot r) {
    if(r instanceof Null)
      System.out.println("[Null Robot]");
    System.out.println("Robot name: " + r.name());
    System.out.println("Robot model: " + r.model());
    for(Operation operation : r.operations()) {
      System.out.println(operation.description.get());
      operation.command.run();
    }
  }
}
```

You access a `Robot`'s services by calling `operations()`. `Robot` also incorporates a `static` method to perform tests.

`Operation` contains a description and a command (it's a type of *Command* pattern). These are defined as references to functional interfaces so you can pass lambda expressions or method references to the `Operation` constructor:

```
// reflection/Operation.java
import java.util.function.*;

public class Operation {
  public final Supplier<String> description;
  public final Runnable command;
  public
  Operation(Supplier<String> descr, Runnable cmd) {
    description = descr;
    command = cmd;
  }
}
```

We can now create a `Robot` that removes snow:

```
// reflection/SnowRobot.java
import java.util.*;

public class SnowRobot implements Robot {
  private String name;
  public SnowRobot(String name) {
    this.name = name;
  }
  @Override public String name() { return name; }
  @Override public String model() {
    return "SnowBot Series 11";
  }
  private List<Operation> ops = Arrays.asList(
    new Operation(
      () -> name + " can shovel snow",
      () -> System.out.println(
        name + " shoveling snow")),
    new Operation(
      () -> name + " can chip ice",
      () -> System.out.println(name + " chipping ice")),
    new Operation(
      () -> name + " can clear the roof",
      () -> System.out.println(
        name + " clearing roof")));
  @Override
  public List<Operation> operations() { return ops; }
  public static void main(String[] args) {
    Robot.test(new SnowRobot("Slusher"));
  }
}
/* Output:
Robot name: Slusher
Robot model: SnowBot Series 11
Slusher can shovel snow
Slusher shoveling snow
Slusher can chip ice
Slusher chipping ice
Slusher can clear the roof
Slusher clearing roof
*/
```

There will presumably be many different types of `Robot`, and we'd like each `Null`

to do something special for each Robot type—here, incorporate information about the exact type of Robot the Null represents. This information is captured by the dynamic proxy:

```
// reflection/NullRobot.java
// Using a dynamic proxy to create an Optional
import java.lang.reflect.*;
import java.util.*;
import java.util.stream.*;
import onjava.*;

class NullRobotProxyHandler
implements InvocationHandler {
  private String nullName;
  private Robot proxied = new NRobot();
  NullRobotProxyHandler(Class<? extends Robot> type) {
    nullName = type.getSimpleName() + " NullRobot";
  }
  private class NRobot implements Null, Robot {
    @Override
    public String name() { return nullName; }
    @Override
    public String model() { return nullName; }
    @Override public List<Operation> operations() {
      return Collections.emptyList();
    }
  }
  @Override public Object
  invoke(Object proxy, Method method, Object[] args)
  throws Throwable {
    return method.invoke(proxied, args);
  }
}

public class NullRobot {
  public static Robot
  newNullRobot(Class<? extends Robot> type) {
    return (Robot)Proxy.newProxyInstance(
      NullRobot.class.getClassLoader(),
      new Class[]{ Null.class, Robot.class },
      new NullRobotProxyHandler(type));
  }
```

```
  public static void main(String[] args) {
    Stream.of(
      new SnowRobot("SnowBee"),
      newNullRobot(SnowRobot.class)
    ).forEach(Robot::test);
  }
}
/* Output:
Robot name: SnowBee
Robot model: SnowBot Series 11
SnowBee can shovel snow
SnowBee shoveling snow
SnowBee can chip ice
SnowBee chipping ice
SnowBee can clear the roof
SnowBee clearing roof
[Null Robot]
Robot name: SnowRobot NullRobot
Robot model: SnowRobot NullRobot
*/
```

Whenever you need a `null Robot` object, you just call `newNullRobot()`, passing your desired type of `Robot`, and it returns a proxy. The proxy fulfills the requirements of the `Robot` and `Null` interfaces, and provides the specific name of the type it proxies.

Mock Objects & Stubs

Logical variations of `Optional` are *Mock Objects* and *Stubs*. Both of these are proxies for the "real" object used in the finished program. Both Mock Objects and Stubs pretend to be real objects that deliver real information, rather than hiding objects including potential `null`s, as `Optional` does.

The distinction between Mock Objects and Stubs is one of degree. Mock Objects tend to be lightweight and self-testing, and usually many of them are created to handle various testing situations. Stubs just return stubbed data, are typically heavyweight and are often reused between tests. Stubs can be configured to change depending on how they are called. So a Stub is a sophisticated object that does one thing, whereas you usually create lots of small, simple Mock Objects if you must do many things.

Interfaces and Type Information

An important goal of the `interface` keyword is to allow the programmer to isolate components, and thus reduce coupling. If you write to interfaces, you accomplish this, but with type information it's possible to get around that—interfaces are not airtight guarantees of decoupling. If we start with an interface:

```
// reflection/interfacea/A.java
package reflection.interfacea;

public interface A {
  void f();
}
```

Here's how you sneak around and access the actual implementation type:

```
// reflection/InterfaceViolation.java
// Sneaking around an interface
import reflection.interfacea.*;

class B implements A {
  @Override public void f() {}
  public void g() {}
}

public class InterfaceViolation {
  public static void main(String[] args) {
    A a = new B();
    a.f();
    // a.g(); // Compile error
    System.out.println(a.getClass().getName());
    if(a instanceof B) {
      B b = (B)a;
      b.g();
    }
  }
}
/* Output:
B
*/
```

Using reflection, we discover that `a` is implemented as a `B`. By casting to `B`, we can call a method that's not in `A`.

This is perfectly legal and acceptable, but you might not want client programmers to do this, because it gives them an opportunity to couple more closely to your code than you'd like. That is, you might think the `interface` keyword is protecting you, but it isn't, and the fact that you're using `B` to implement `A` here is effectively a matter of public record.[141]

One solution is to simply say that programmers are on their own if they decide to use the actual class rather than the interface. This is probably reasonable in many cases, but if "probably" isn't enough, you might apply more stringent controls.

The easiest approach is to use package access for the implementation, so clients outside the package cannot see it:

```
// reflection/packageaccess/HiddenC.java
package reflection.packageaccess;
import reflection.interfacea.*;

class C implements A {
  @Override public void f() {
    System.out.println("public C.f()");
  }
  public void g() {
    System.out.println("public C.g()");
  }
  void u() {
    System.out.println("package C.u()");
  }
  protected void v() {
    System.out.println("protected C.v()");
  }
  private void w() {
    System.out.println("private C.w()");
  }
}
```

[141] The most famous case of this is the Windows operating system, which had a published API you were supposed to write to, and an unpublished but visible set of functions you could discover and call. To solve problems, programmers used the hidden API functions, which forced Microsoft to maintain them as if they were part of the public API. This became a source of great cost and effort for the company.

```
public class HiddenC {
  public static A makeA() { return new C(); }
}
```

The only `public` part of this package, `class HiddenC`, produces an `A` interface when you call it. Even if you were to return a `C` from `makeA()`, you still couldn't use anything but an `A` from outside the package because you cannot name `C` outside the package.

Now if you try to downcast to `C`, you can't do it because there is no `C` type available outside the package:

```
// reflection/HiddenImplementation.java
// Sneaking around package hiding
import reflection.interfacea.*;
import reflection.packageaccess.*;
import java.lang.reflect.*;

public class HiddenImplementation {
  public static void
  main(String[] args) throws Exception {
    A a = HiddenC.makeA();
    a.f();
    System.out.println(a.getClass().getName());
    // Compile error: cannot find symbol 'C':
    /* if(a instanceof C) {
      C c = (C)a;
      c.g();
    } */
    // Oops! Reflection still allows us to call g():
    callHiddenMethod(a, "g");
    // And even less accessible methods!
    callHiddenMethod(a, "u");
    callHiddenMethod(a, "v");
    callHiddenMethod(a, "w");
  }
  static void
  callHiddenMethod(Object a, String methodName)
  throws Exception {
    Method g =
      a.getClass().getDeclaredMethod(methodName);
    g.setAccessible(true);
```

```
    g.invoke(a);
  }
}
/* Output:
public C.f()
reflection.packageaccess.C
public C.g()
package C.u()
protected C.v()
private C.w()
*/
```

It's still possible to reach in and call *all* methods using reflection, even `private` methods! If you know the name of the method, you can call `setAccessible(true)` on the `Method` object to make it callable, as seen in `callHiddenMethod()`.

You might think you can prevent this by only distributing compiled code, but that's no solution. All you must do is run `javap`, which is the decompiler that comes with the JDK. Here's the command line:

```
javap -private C
```

The `-private` flag indicates that all members should be displayed, even private ones. Here's the output:

```
class reflection.packageaccess.C extends
java.lang.Object implements reflection.interfacea.A {
  reflection.packageaccess.C();
  public void f();
  public void g();
  void u();
  protected void v();
  private void w();
}
```

So anyone can get the names and signatures of your most private methods, and call them.

What if you implement the interface as a `private` inner class? Here's what it looks like:

```
// reflection/InnerImplementation.java
// Private inner classes can't hide from reflection.
import reflection.interfacea.*;

class InnerA {
  private static class C implements A {
    @Override public void f() {
      System.out.println("public C.f()");
    }
    public void g() {
      System.out.println("public C.g()");
    }
    void u() {
      System.out.println("package C.u()");
    }
    protected void v() {
      System.out.println("protected C.v()");
    }
    private void w() {
      System.out.println("private C.w()");
    }
  }
  public static A makeA() { return new C(); }
}

public class InnerImplementation {
  public static void
  main(String[] args) throws Exception {
    A a = InnerA.makeA();
    a.f();
    System.out.println(a.getClass().getName());
    // Reflection still gets into the private class:
    HiddenImplementation.callHiddenMethod(a, "g");
    HiddenImplementation.callHiddenMethod(a, "u");
    HiddenImplementation.callHiddenMethod(a, "v");
    HiddenImplementation.callHiddenMethod(a, "w");
  }
}
/* Output:
public C.f()
InnerA$C
public C.g()
```

```
package C.u()
protected C.v()
private C.w()
*/
```

That didn't hide anything from reflection. What about an anonymous class?

```
// reflection/AnonymousImplementation.java
// Anonymous inner classes can't hide from reflection
import reflection.interfacea.*;

class AnonymousA {
  public static A makeA() {
    return new A() {
      @Override public void f() {
        System.out.println("public C.f()");
      }
      public void g() {
        System.out.println("public C.g()");
      }
      void u() {
        System.out.println("package C.u()");
      }
      protected void v() {
        System.out.println("protected C.v()");
      }
      private void w() {
        System.out.println("private C.w()");
      }
    };
  }
}

public class AnonymousImplementation {
  public static void
  main(String[] args) throws Exception {
    A a = AnonymousA.makeA();
    a.f();
    System.out.println(a.getClass().getName());
    // Reflection still gets into the anonymous class:
    HiddenImplementation.callHiddenMethod(a, "g");
    HiddenImplementation.callHiddenMethod(a, "u");
```

```
    HiddenImplementation.callHiddenMethod(a, "v");
    HiddenImplementation.callHiddenMethod(a, "w");
  }
}
/* Output:
public C.f()
AnonymousA$1
public C.g()
package C.u()
protected C.v()
private C.w()
*/
```

There doesn't seem to be any way to prevent reflection from reaching in and calling methods that have non-public access. This is also true for fields, even `private` fields:

```
// reflection/ModifyingPrivateFields.java
import java.lang.reflect.*;

class WithPrivateFinalField {
  private int i = 1;
  private final String s = "I'm totally safe";
  private String s2 = "Am I safe?";
  @Override public String toString() {
    return "i = " + i + ", " + s + ", " + s2;
  }
}

public class ModifyingPrivateFields {
  public static void
  main(String[] args) throws Exception {
    WithPrivateFinalField pf =
      new WithPrivateFinalField();
    System.out.println(pf);
    Field f = pf.getClass().getDeclaredField("i");
    f.setAccessible(true);
    System.out.println(
      "f.getInt(pf): " + f.getInt(pf));
    f.setInt(pf, 47);
    System.out.println(pf);
    f = pf.getClass().getDeclaredField("s");
    f.setAccessible(true);
```

```
    System.out.println("f.get(pf): " + f.get(pf));
    f.set(pf, "No, you're not!");
    System.out.println(pf);
    f = pf.getClass().getDeclaredField("s2");
    f.setAccessible(true);
    System.out.println("f.get(pf): " + f.get(pf));
    f.set(pf, "No, you're not!");
    System.out.println(pf);
  }
}
/* Output:
i = 1, I'm totally safe, Am I safe?
f.getInt(pf): 1
i = 47, I'm totally safe, Am I safe?
f.get(pf): I'm totally safe
i = 47, I'm totally safe, Am I safe?
f.get(pf): Am I safe?
i = 47, I'm totally safe, No, you're not!
*/
```

However, `final` fields are actually safe from change. The runtime system accepts any attempts at change without complaint, but nothing actually happens.

In general, all these access violations are not the worst thing in the world. If someone uses such a technique to call methods you marked with `private` or package access (meaning, they should not be called), then it's difficult for them to complain if you change some aspect of those methods. On the other hand, the fact you always have a back door into a class can allow you to solve certain types of problems that could otherwise be difficult or impossible, and the benefits of reflection in general are undeniable.

Programmers often become overconfident about the access control afforded by the language, going so far as to believe that Java is somehow superior in its safety than other languages that provide (apparently) less stringent access control[142]. As you can see, it's not.

[142] In Python, for example, you put a double underscore __ in front of any element you want hidden, and the runtime complains if you try to access it outside the class or package.

Summary

Reflection discovers type information from an anonymous base-class reference. Thus, it's ripe for misuse by the novice because it might make sense before polymorphic method calls do. For people coming from a procedural background, it's difficult not to organize programs into sets of `switch` statements. You can accomplish this with reflection and thus lose the important value of polymorphism in code development and maintenance. The intent of OO programming is to use polymorphic method calls everywhere you can, and reflection only when you must.

However, using polymorphic method calls as they are intended requires you have control of the base-class definition, because at some point in the extension of your program you might discover that the base class doesn't include the method you need. If the base class comes from someone else's library, one solution is reflection: You can inherit a new type and add your extra method. Elsewhere in the code you can detect your particular type and call that special method. This doesn't destroy the polymorphism and extensibility of the program, because adding a new type will not require you to hunt for `switch` statements in your program. However, when you add code that requires your new feature, you must use reflection to detect your particular type.

Putting a feature in a base class might mean that, for the benefit of one particular class, the interface becomes less sensible. For example, consider a class hierarchy representing musical instruments. Suppose you want to clear the spit valves of all the appropriate instruments in your orchestra. One option is to put a `clearSpitValve()` method in the base class `Instrument`, but this is confusing because it implies that `Percussion`, `Stringed` and `Electronic` instruments also have spit valves. Reflection provides a reasonable solution because you can place the method in the specific class where it's appropriate (`Wind`, in this case). At the same time, you might discover there's a more sensible solution such as a `prepareInstrument()` method in the base class. However, you might not see such a solution when you're first solving the problem and could mistakenly assume you must use reflection.

Finally, reflection will sometimes solve efficiency problems. Suppose your code uses polymorphism, but one of your objects reacts to this general-purpose code in a horribly inefficient way. You can pick out that type using reflection and write case-specific code to improve the efficiency. Be wary, however, of programming for

efficiency too soon. It's a seductive trap. It's best to get the program working *first*, then decide if it's running fast enough, and only then should you attack efficiency issues using a profiler.

We've also seen that reflection opens up a new world of programming possibilities by allowing a much more dynamic style of programming. There are some for whom this dynamic nature of reflection is disturbing. The fact you can do things that can only be checked at run time and reported with exceptions seems, to a mind grown comfortable with the security of static type checking, to be the wrong direction. Some people go so far as to say that introducing the possibility of a runtime exception is a clear indicator that such code should be avoided. I find this sense of security is an illusion—there are always things that can happen at run time and throw exceptions, even in a program that contains no `try` blocks or exception specifications. Instead, I think the existence of a consistent error-reporting model *empowers* us to write dynamic code using reflection. It's worth trying to write code that can be statically checked … when you can. But I believe that dynamic code is one of the important facilities that separate Java from languages like C++.

Generics

> Ordinary classes and methods work with specific types: either primitives or class types. If you write code to use across more types, this rigidity can be overconstraining.

Polymorphism is an object-oriented generalization tool. You write a method that takes a base class object as an argument, then use that method with any class derived from the base class—including classes that haven't been created yet. Now your method is more general, and useful in more places. The same is true within classes—anyplace you use a specific type, a base type provides more flexibility. Anything but a `final` class (or a class with all `private` constructors) can be extended, so this flexibility is automatic much of the time.

A single hierarchy can be too limiting because you must inherit *from that hierarchy* to produce an object that fits your method argument. If a method argument is an interface instead of a class, the limitations are loosened to include anything that implements the interface. This gives the client programmer the option of implementing an interface in combination with an existing class—that is, to adapt an existing class to fit your method. Interfaces cut across class hierarchies, as long as you have the option to implement those interfaces to fit.

Sometimes even an interface is too restrictive. An interface still requires that your code work with that particular interface. You can write even more general code if you can say that your code works with "some unspecified type," rather than a specific interface or class.

This is the concept of generics, one of the more significant changes in Java 5. Generics produce *parameterized types*, so you can create components (most notably collections) that are easy to use with multiple types. The term "generic" means "pertaining or appropriate to large groups of classes." The original intent of generics in programming languages was to allow the programmer the greatest amount of expressiveness possible when writing classes or methods, by loosening type constraints on those classes or methods. As you will see in this chapter, the Java

implementation of generics is not that broad—indeed, you might question whether the term "generic" is even appropriate for this feature.

If you've never seen any kind of parameterized type mechanism before, Java generics probably seem like a convenient addition to the language. When you create an instance of a parameterized type, casts are automatic and type correctness is ensured at compile time. This is an improvement.

However, if you've had experience with parameterized types, for example in C++, you'll find you can't do everything you might expect when using Java generics. While using someone else's generic type is usually easy, creating your own produces numerous surprises.

This is not to say that Java generics are useless. In many cases they make code more straightforward and even elegant. But if you're coming from a language that has implemented a more pure version of generics, you might be frustrated. In this chapter, we will examine strengths and limitations of Java generics. I try to explain how the feature came to be as it is, so you can use generics more effectively.[143]

Comparison with C++

The Java designers stated that much of the inspiration for the language came as a reaction to C++. Despite this, it is possible to teach Java largely without reference to C++.

Generics require comparison with C++ for two reasons. First, understanding certain aspects of C++ *templates* (the main inspiration for generics, including the basic syntax) illuminates the foundations of the concept, as well as—and this is very important—the limitations of Java generics and why those limitations exist. The ultimate goal is a clear understanding of the boundaries, making you a more powerful programmer. Knowing what you can't do, you make better use of what you *can* do (partly because you don't waste time bumping against walls).

The second reason is there is significant misunderstanding in the Java community about C++ templates, and this misunderstanding can further confuse you about the intent of generics.

[143] Angelika Langer's Java Generics FAQ as well as her other writings (together with Klaus Kreft) were invaluable during the preparation of this chapter.

Thus, I introduce a few C++ template examples in this chapter, but only when they improve your depth of understanding.

Simple Generics

One of the most compelling initial motivations for generics is to create *collection classes*, which you saw in the Collections chapter. A collection is an object that holds other objects, while you're working with them. This is also true of arrays, but collections tend to be more flexible and have different characteristics than simple arrays. Virtually all programs require you hold a group of objects while you use them, so collections are one of the most reusable of class libraries.

Let's look at a class that holds a single object. The class can specify the exact type of the object, like this:

```
// generics/Holder1.java

class Automobile {}

public class Holder1 {
  private Automobile a;
  public Holder1(Automobile a) { this.a = a; }
  Automobile get() { return a; }
}
```

This is not a very reusable tool because it can't be used to hold anything else. We would prefer not to write a new one of these for every type we encounter.

Before Java 5, we would simply make it hold an `Object`:

```
// generics/ObjectHolder.java

public class ObjectHolder {
  private Object a;
  public ObjectHolder(Object a) { this.a = a; }
  public void set(Object a) { this.a = a; }
  public Object get() { return a; }
  public static void main(String[] args) {
    ObjectHolder h2 =
      new ObjectHolder(new Automobile());
    Automobile a = (Automobile)h2.get();
    h2.set("Not an Automobile");
    String s = (String)h2.get();
    h2.set(1); // Autoboxes to Integer
    Integer x = (Integer)h2.get();
  }
}
```

Now an `ObjectHolder` can hold anything—and in this example, a single `ObjectHolder` holds three different types of objects.

In rare cases, you'll want a collection to hold multiple types of objects, but typically you only put one type of object into a particular collection object. One of the primary motivations for generics is to specify the type of object a collection holds, and to enforce that specification via the compiler.

So instead of `Object`, we'd like to give a type placeholder, to be decided at a later time. To do this, you put a *type parameter* inside angle brackets after the class name, then substitute an actual type when you use the class. For the "holder" class, it looks like this, where `T` is the type parameter:

```
// generics/GenericHolder.java

public class GenericHolder<T> {
  private T a;
  public GenericHolder() {}
  public void set(T a) { this.a = a; }
  public T get() { return a; }
  public static void main(String[] args) {
    GenericHolder<Automobile> h3 =
      new GenericHolder<>();
    h3.set(new Automobile()); // type checked
    Automobile a = h3.get(); // No cast needed
    //- h3.set("Not an Automobile"); // Error
    //- h3.set(1); // Error
  }
}
```

When creating a `GenericHolder`, you specify the type it holds using the angle-bracket syntax, as you see in `main()`. You are only allowed to put objects of that type (or a subtype, because the substitution principle still works with generics) into the holder. When you call `get()` to produce a value, it is automatically the right type.

That's the core idea of Java generics: You tell it what type to use, and it takes care of the details.

You'll note that the definition of `h3` is rather wordy and redundant. On the left side of the equals sign, you say `GenericHolder<Automobile>`, then you say the same thing again on the right side of the equals sign. When Java 5 came out, this noise was explained away as "necessary," but by Java 7 the designers had fixed the problem (and this new simplicity then became a touted feature). So now you can use the simpler form:

```
// generics/Diamond.java

class Bob {}

public class Diamond<T> {
  public static void main(String[] args) {
    GenericHolder<Bob> h3 = new GenericHolder<>();
    h3.set(new Bob());
  }
}
```

Notice that the right-hand side of the definition of `h3` now uses the empty "diamond" syntax rather than duplicating the type information from the left. You'll see this used throughout the rest of the book.

In general, you can treat generics as if they are any other type—they just happen to have type parameters. To use a generic definition, you just name it along with its type argument list.

A Tuple Library

You'll often return multiple objects from a method call. The `return` statement only returns a single object, so the solution is to create an object that holds multiple objects, and return that object. You can write a special class every time you encounter the situation, but with generics it's possible to solve the problem once and save yourself the effort in the future. At the same time, you are ensuring compile-time type safety.

This concept is called a *tuple*, and it is a group of objects wrapped together into a single object. The recipient of the object is allowed to read the elements but not put new ones in. (This concept is also called a *Data Transfer Object* or *Messenger*.)

Tuples can typically be any length, and each object in the tuple can be of a different type. However, we specify the type of each object and ensure that, when the recipient reads the value, they get the right type. To deal with the problem of multiple lengths, we create multiple different tuples. Here's one that holds two objects:

```
// onjava/Tuple2.java
package onjava;

public class Tuple2<A, B> {
  public final A a1;
  public final B a2;
  public Tuple2(A a, B b) { a1 = a; a2 = b; }
  public String rep() { return  a1 + ", " + a2; }
  @Override public String toString() {
    return "(" + rep() + ")";
  }
}
```

The constructor captures the object to be stored. The tuple implicitly keeps its elements in order.

Upon first reading, you might think this violates common safety principles of Java programming. Shouldn't `a1` and `a2` be `private`, and only accessed with methods named `getFirst()` and `getSecond()`? Consider the "safety" that produces: Clients could still read the objects and do whatever they want with them, but they could not assign `a1` or `a2` to anything else. The `final` declaration buys you the same safety, but the above form is shorter and simpler.

Another design observation is that you might *want* to allow a client programmer to point `a1` or `a2` to another object. However, it's safer to leave it in the above form, and just force the user to create a new `Tuple2` if they want one that has different elements.

The longer-length tuples can be created with inheritance. Adding more type parameters is a simple matter:

```
// onjava/Tuple3.java
package onjava;

public class Tuple3<A, B, C> extends Tuple2<A, B> {
  public final C a3;
  public Tuple3(A a, B b, C c) {
    super(a, b);
    a3 = c;
  }
  @Override public String rep() {
    return super.rep() + ", " + a3;
  }
}
```

```
// onjava/Tuple4.java
package onjava;

public class Tuple4<A, B, C, D>
  extends Tuple3<A, B, C> {
  public final D a4;
  public Tuple4(A a, B b, C c, D d) {
    super(a, b, c);
    a4 = d;
  }
  @Override public String rep() {
    return super.rep() + ", " + a4;
  }
}
```

```
// onjava/Tuple5.java
package onjava;

public class Tuple5<A, B, C, D, E>
extends Tuple4<A, B, C, D> {
  public final E a5;
  public Tuple5(A a, B b, C c, D d, E e) {
    super(a, b, c, d);
    a5 = e;
  }
  @Override public String rep() {
    return super.rep() + ", " + a5;
  }
}
```

For tuple experiments, we'll define a couple of classes:

```
// generics/Amphibian.java
public class Amphibian {}

// generics/Vehicle.java
public class Vehicle {}
```

To use a tuple, you define the appropriate-length tuple as the return value for your function, then create and return it. Notice the return type declarations for the method definitions:

```
// generics/TupleTest.java
import onjava.*;

public class TupleTest {
  static Tuple2<String, Integer> f() {
    // Autoboxing converts the int to Integer:
    return new Tuple2<>("hi", 47);
  }
  static Tuple3<Amphibian, String, Integer> g() {
    return new Tuple3<>(new Amphibian(), "hi", 47);
  }
  static
  Tuple4<Vehicle, Amphibian, String, Integer> h() {
    return
      new Tuple4<>(
        new Vehicle(), new Amphibian(), "hi", 47);
  }
  static
  Tuple5<Vehicle, Amphibian,
         String, Integer, Double> k() {
    return new
      Tuple5<>(
        new Vehicle(), new Amphibian(), "hi", 47, 11.1);
  }
  public static void main(String[] args) {
    Tuple2<String, Integer> ttsi = f();
    System.out.println(ttsi);
    // ttsi.a1 = "there"; // Compile error: final
    System.out.println(g());
    System.out.println(h());
```

```
    System.out.println(k());
  }
}
/* Output:
(hi, 47)
(Amphibian@1c7c054, hi, 47)
(Vehicle@14991ad, Amphibian@d93b30, hi, 47)
(Vehicle@a14482, Amphibian@140e19d, hi, 47, 11.1)
*/
```

With generics, you can easily create any tuple to return any group of types, just by writing the expression.

The `final` specification on the public fields prevents them from reassignment after construction, as shown in the failure of the statement `ttsi.a1 = "there"`.

The `new` expressions are a little verbose. Later in this chapter you'll see how to simplify them using *generic methods*.

A Stack Class

Let's look at something slightly more complicated: the traditional pushdown stack. In the Collections chapter, you saw this implemented using a `LinkedList` as the `onjava.Stack` class. That example showed that a `LinkedList` already has the necessary methods to create a stack. The `Stack` was constructed by composing one generic class (`Stack<T>`) with another generic class (`LinkedList<T>`). In that example, notice that (with a few differences we look at later) a generic type is just another type.

Instead of using `LinkedList`, we can implement our own internal linked storage mechanism.

```
// generics/LinkedStack.java
// A stack implemented with an internal linked structure

public class LinkedStack<T> {
  private static class Node<U> {
    U item;
    Node<U> next;
    Node() { item = null; next = null; }
    Node(U item, Node<U> next) {
      this.item = item;
      this.next = next;
    }
    boolean end() {
      return item == null && next == null;
    }
  }
  private Node<T> top = new Node<>(); // End sentinel
  public void push(T item) {
    top = new Node<>(item, top);
  }
  public T pop() {
    T result = top.item;
    if(!top.end())
      top = top.next;
    return result;
  }
  public static void main(String[] args) {
    LinkedStack<String> lss = new LinkedStack<>();
    for(String s : "Phasers on stun!".split(" "))
      lss.push(s);
    String s;
    while((s = lss.pop()) != null)
      System.out.println(s);
  }
}
/* Output:
stun!
on
Phasers
*/
```

The inner class Node is also generic, and has its own type parameter.

This example makes use of an *end sentinel* to determine when the stack is empty. The end sentinel is created when the `LinkedStack` is constructed, and each time you call `push()` a new `Node<T>` is created and linked to the previous `Node<T>`. When you call `pop()`, you always return the `top.item`, then you discard the current `Node<T>` and move to the next one—except when you hit the end sentinel, when you don't move. That way, if the client keeps calling `pop()`, they keep getting `null` back to indicate the stack is empty.

`RandomList`

For another example of a holder, suppose you'd like a special type of list that randomly selects one of its elements each time you call `select()`. To build a tool that works with all objects, use generics:

```
// generics/RandomList.java
import java.util.*;
import java.util.stream.*;

public class RandomList<T> extends ArrayList<T> {
  private Random rand = new Random(47);
  public T select() {
    return get(rand.nextInt(size()));
  }
  public static void main(String[] args) {
    RandomList<String> rs = new RandomList<>();
    Arrays.stream(
      ("The quick brown fox jumped over " +
      "the lazy brown dog").split(" "))
      .forEach(rs::add);
    IntStream.range(0, 11).forEach(i ->
      System.out.print(rs.select() + " "));
  }
}
/* Output:
brown over fox quick quick dog brown The brown lazy
brown
*/
```

Because it inherits `ArrayList`, `RandomList` has all the baked-in behaviors from `ArrayList`. We've only added the `select()` method.

Generic Interfaces

Generics also work with interfaces. For example, a *generator* is a class that creates objects. The generator actually a specialization of the *Factory Method* design pattern, but when you ask a generator for new object, you don't pass it any arguments, whereas you typically do pass arguments to a Factory Method. The generator knows how to create new objects without any extra information.

Typically, a generator just defines one method, the method that produces new objects. The `java.util.function` library defines a generator as `Supplier`, and the producer method is called `get()`. The return type of `get()` is parameterized to `T`.

To create a `Supplier`, we'll need some classes. Here's a coffee hierarchy:

```
// generics/coffee/Coffee.java
package generics.coffee;

public class Coffee {
  private static long counter = 0;
  private final long id = counter++;
  @Override public String toString() {
    return getClass().getSimpleName() + " " + id;
  }
}

// generics/coffee/Latte.java
package generics.coffee;
public class Latte extends Coffee {}

// generics/coffee/Mocha.java
package generics.coffee;
public class Mocha extends Coffee {}

// generics/coffee/Cappuccino.java
package generics.coffee;
public class Cappuccino extends Coffee {}
```

```
// generics/coffee/Americano.java
package generics.coffee;
public class Americano extends Coffee {}

// generics/coffee/Breve.java
package generics.coffee;
public class Breve extends Coffee {}
```

Now we can implement a `Supplier<Coffee>` that produces random different types of `Coffee` objects:

```
// generics/coffee/CoffeeSupplier.java
// {java generics.coffee.CoffeeSupplier}
package generics.coffee;
import java.util.*;
import java.util.function.*;
import java.util.stream.*;
import java.lang.reflect.InvocationTargetException;

public class CoffeeSupplier
implements Supplier<Coffee>, Iterable<Coffee> {
  private Class<?>[] types = { Latte.class, Mocha.class,
    Cappuccino.class, Americano.class, Breve.class, };
  private static Random rand = new Random(47);
  public CoffeeSupplier() {}
  // For iteration:
  private int size = 0;
  public CoffeeSupplier(int sz) { size = sz; }
  @Override public Coffee get() {
    try {
      return (Coffee)
        types[rand.nextInt(types.length)]
        .getConstructor().newInstance();
      // Report programmer errors at run time:
      } catch(InstantiationException |
              NoSuchMethodException |
              InvocationTargetException |
              IllegalAccessException e) {
      throw new RuntimeException(e);
    }
  }
  class CoffeeIterator implements Iterator<Coffee> {
```

```
    int count = size;
    @Override
    public boolean hasNext() { return count > 0; }
    @Override public Coffee next() {
      count--;
      return CoffeeSupplier.this.get();
    }
    @Override
    public void remove() { // Not implemented
      throw new UnsupportedOperationException();
    }
  }
  @Override public Iterator<Coffee> iterator() {
    return new CoffeeIterator();
  }
  public static void main(String[] args) {
    Stream.generate(new CoffeeSupplier())
      .limit(5)
      .forEach(System.out::println);
    for(Coffee c : new CoffeeSupplier(5))
      System.out.println(c);
  }
}
/* Output:
Americano 0
Latte 1
Americano 2
Mocha 3
Mocha 4
Breve 5
Americano 6
Latte 7
Cappuccino 8
Cappuccino 9
*/
```

The parameterized `Supplier` interface ensures that `get()` returns the parameter type. `CoffeeSupplier` also implements the `Iterable` interface, so it can be used in a *for-in* statement. However, it must know when to stop, and this is provided by the second constructor.

Here's a second implementation of `Supplier<T>`, this time to produce Fibonacci

numbers:

```
// generics/Fibonacci.java
// Generate a Fibonacci sequence
import java.util.function.*;
import java.util.stream.*;

public class Fibonacci implements Supplier<Integer> {
  private int count = 0;
  @Override
  public Integer get() { return fib(count++); }
  private int fib(int n) {
    if(n < 2) return 1;
    return fib(n-2) + fib(n-1);
  }
  public static void main(String[] args) {
    Stream.generate(new Fibonacci())
      .limit(18)
      .map(n -> n + " ")
      .forEach(System.out::print);
  }
}
/* Output:
1 1 2 3 5 8 13 21 34 55 89 144 233 377 610 987 1597
2584
*/
```

Although we are working with `int`s both inside and outside the class, the type parameter is `Integer`. This brings up one of the limitations of Java generics: You cannot use primitives as type parameters. However, Java 5 autoboxing and autounboxing converts from primitive types to wrapper types and back. You see the effect here because `int`s are seamlessly used and produced by the class.

We can go one step further and make an `Iterable` Fibonacci generator. One option is to reimplement the class and add the `Iterable` interface, but you don't always have control of the original code, and you don't rewrite unless you must. Instead, we can create an *Adapter* to produce the desired interface—this design pattern was introduced earlier in the book.

Adapters can be implemented in multiple ways. For example, inheritance can generate the adapted class:

```
// generics/IterableFibonacci.java
// Adapt the Fibonacci class to make it Iterable
import java.util.*;

public class IterableFibonacci
extends Fibonacci implements Iterable<Integer> {
  private int n;
  public IterableFibonacci(int count) { n = count; }
  @Override public Iterator<Integer> iterator() {
    return new Iterator<Integer>() {
      @Override
      public boolean hasNext() { return n > 0; }
      @Override public Integer next() {
        n--;
        return IterableFibonacci.this.get();
      }
      @Override
      public void remove() { // Not implemented
        throw new UnsupportedOperationException();
      }
    };
  }
  public static void main(String[] args) {
    for(int i : new IterableFibonacci(18))
      System.out.print(i + " ");
  }
}
/* Output:
1 1 2 3 5 8 13 21 34 55 89 144 233 377 610 987 1597
2584
*/
```

To use `IterableFibonacci` in a *for-in* statement, you give the constructor a boundary so `hasNext()` can know when to return `false`.

Generic Methods

So far we've looked at parameterizing entire classes. You can also parameterize methods within a class. The class itself might or might not be generic—this is independent of whether you have a generic method.

A generic method varies the method independently of the class. As a guideline, use generic methods "whenever you can." Making a single method generic rather than the entire class is generally clearer.

If a method is `static`, it has no access to the generic type parameters of the class, so if it uses genericity it must be a generic method.

To define a generic method, place the generic parameter list before the return value, like this:

```
// generics/GenericMethods.java

public class GenericMethods {
  public <T> void f(T x) {
    System.out.println(x.getClass().getName());
  }
  public static void main(String[] args) {
    GenericMethods gm = new GenericMethods();
    gm.f("");
    gm.f(1);
    gm.f(1.0);
    gm.f(1.0F);
    gm.f('c');
    gm.f(gm);
  }
}
/* Output:
java.lang.String
java.lang.Integer
java.lang.Double
java.lang.Float
java.lang.Character
GenericMethods
*/
```

The class `GenericMethods` is not parameterized, although both a class and its methods can be parameterized at the same time. But here, only the method `f()` has a type parameter, indicated by the parameter list before the method's return type.

With a generic class, you must specify the type parameters when you instantiate the class. With a generic method, you don't usually specify the parameter types, because the compiler figures that out for you. This is called *type argument inference*. So calls

to `f()` look like normal method calls, and it appears that `f()` is infinitely overloaded. It will even take an argument of the type `GenericMethods`.

For the calls to `f()` that use primitive types, autoboxing comes into play, automatically wrapping the primitive types in their associated objects.

Varargs and Generic Methods

Generic methods and variable argument lists coexist nicely:

```
// generics/GenericVarargs.java
import java.util.*;

public class GenericVarargs {
  @SafeVarargs
  public static <T> List<T> makeList(T... args) {
    List<T> result = new ArrayList<>();
    for(T item : args)
      result.add(item);
    return result;
  }
  public static void main(String[] args) {
    List<String> ls = makeList("A");
    System.out.println(ls);
    ls = makeList("A", "B", "C");
    System.out.println(ls);
    ls = makeList(
      "ABCDEFFHIJKLMNOPQRSTUVWXYZ".split(""));
    System.out.println(ls);
  }
}
/* Output:
[A]
[A, B, C]
[A, B, C, D, E, F, F, H, I, J, K, L, M, N, O, P, Q, R,
S, T, U, V, W, X, Y, Z]
*/
```

The `makeList()` method shown here produces the same functionality as the standard library's `java.util.Arrays.asList()` method.

The `@SafeVarargs` annotation promises we are not making any modifications to the variable argument list, which is true because we only read from it. Without the annotation the compiler can't know and issues a warning.

A General-Purpose `Supplier`

Here's a class that produces a `Supplier` for any class that has a zero-argument constructor. To reduce typing, it also includes a generic method to produce a `BasicSupplier`:

```
// onjava/BasicSupplier.java
// Supplier from a class with a zero-argument constructor
package onjava;
import java.util.function.*;
import java.lang.reflect.InvocationTargetException;

public class BasicSupplier<T> implements Supplier<T> {
  private Class<T> type;
  public BasicSupplier(Class<T> type) {
    this.type = type;
  }
  @Override public T get() {
    try {
      // Assumes type is a public class:
      return type.getConstructor().newInstance();
    } catch(InstantiationException |
            NoSuchMethodException |
            InvocationTargetException |
            IllegalAccessException e) {
      throw new RuntimeException(e);
    }
  }
  // Produce a default Supplier from a type token:
  public static <T> Supplier<T> create(Class<T> type) {
    return new BasicSupplier<>(type);
  }
}
```

This class provides a basic implementation to produce objects of a class that:

1. Is `public`. Because `BasicSupplier` is in a separate package, the class in question must have `public` and not just package access.
2. Has a zero-argument constructor. To create one of these `BasicSupplier` objects, you call the `create()` method and pass it the type token for the type you want generated. The generic `create()` method provides the syntax `BasicSupplier.create(MyType.class)` instead of the more awkward `new BasicSupplier<MyType>(MyType.class)`.

For example, here's a simple class that has a zero-argument constructor:

```
// generics/CountedObject.java

public class CountedObject {
  private static long counter = 0;
  private final long id = counter++;
  public long id() { return id; }
  @Override public String toString() {
    return "CountedObject " + id;
  }
}
```

The `CountedObject` class keeps track of how many instances of itself are created, and reports these via `toString()`. `BasicSupplier` easily creates a `Supplier` for `CountedObject`:

```
// generics/BasicSupplierDemo.java
import onjava.*;
import java.util.stream.*;

public class BasicSupplierDemo {
  public static void main(String[] args) {
    Stream.generate(
      BasicSupplier.create(CountedObject.class))
      .limit(5)
      .forEach(System.out::println);
  }
}
/* Output:
CountedObject 0
CountedObject 1
CountedObject 2
```

```
CountedObject 3
CountedObject 4
*/
```

The generic method reduces the amount of typing necessary to produce the `Supplier` object. Java generics force you to pass in the `Class` object, so you might as well use it for type inference in the `create()` method.

Simplifying Tuple Use

With type argument inference and `static` imports, we'll rewrite the earlier tuples into a more general-purpose library. Here, we create tuples using an overloaded `static` method:

```
// onjava/Tuple.java
// Tuple library using type argument inference
package onjava;

public class Tuple {
  public static <A, B> Tuple2<A, B> tuple(A a, B b) {
    return new Tuple2<>(a, b);
  }
  public static <A, B, C> Tuple3<A, B, C>
  tuple(A a, B b, C c) {
    return new Tuple3<>(a, b, c);
  }
  public static <A, B, C, D> Tuple4<A, B, C, D>
  tuple(A a, B b, C c, D d) {
    return new Tuple4<>(a, b, c, d);
  }
  public static <A, B, C, D, E>
  Tuple5<A, B, C, D, E> tuple(A a, B b, C c, D d, E e) {
    return new Tuple5<>(a, b, c, d, e);
  }
}
```

We modify `TupleTest.java` to test `Tuple.java`:

```
// generics/TupleTest2.java
import onjava.*;
import static onjava.Tuple.*;

public class TupleTest2 {
  static Tuple2<String, Integer> f() {
    return tuple("hi", 47);
  }
  static Tuple2 f2() { return tuple("hi", 47); }
  static Tuple3<Amphibian, String, Integer> g() {
    return tuple(new Amphibian(), "hi", 47);
  }
  static
  Tuple4<Vehicle, Amphibian, String, Integer> h() {
    return tuple(
      new Vehicle(), new Amphibian(), "hi", 47);
  }
  static
  Tuple5<Vehicle, Amphibian,
         String, Integer, Double> k() {
    return tuple(new Vehicle(), new Amphibian(),
      "hi", 47, 11.1);
  }
  public static void main(String[] args) {
    Tuple2<String, Integer> ttsi = f();
    System.out.println(ttsi);
    System.out.println(f2());
    System.out.println(g());
    System.out.println(h());
    System.out.println(k());
  }
}
/* Output:
(hi, 47)
(hi, 47)
(Amphibian@a298b7, hi, 47)
(Vehicle@16d3586, Amphibian@154617c, hi, 47)
(Vehicle@17327b6, Amphibian@14ae5a5, hi, 47, 11.1)
*/
```

Notice that `f()` returns a parameterized `Tuple2` object, while `f2()` returns an unparameterized `Tuple2` object. The compiler doesn't warn about `f2()` here because

the return value is not used in a parameterized fashion; in a sense, it is "upcast" to an unparameterized `Tuple2`. However, if you were to try to capture the result of `f2()` into a parameterized `Tuple2`, the compiler would issue a warning.

A `Set` Utility

For another example of generic methods, consider the mathematical relationships expressed by `Sets`. These are conveniently defined as generic methods for use with all different types:

```
// onjava/Sets.java
package onjava;
import java.util.*;

public class Sets {
  public static <T> Set<T> union(Set<T> a, Set<T> b) {
    Set<T> result = new HashSet<>(a);
    result.addAll(b);
    return result;
  }
  public static <T>
  Set<T> intersection(Set<T> a, Set<T> b) {
    Set<T> result = new HashSet<>(a);
    result.retainAll(b);
    return result;
  }
  // Subtract subset from superset:
  public static <T> Set<T>
  difference(Set<T> superset, Set<T> subset) {
    Set<T> result = new HashSet<>(superset);
    result.removeAll(subset);
    return result;
  }
  // Reflexive--everything not in the intersection:
  public static
  <T> Set<T> complement(Set<T> a, Set<T> b) {
    return difference(union(a, b), intersection(a, b));
  }
}
```

The first three methods duplicate the first argument by copying its references into a new `HashSet` object, so the argument `Set`s are not directly modified. The return value is thus a new `Set` object.

The four methods represent mathematical set operations: `union()` returns a `Set` containing the combination of the two arguments, `intersection()` returns a `Set` containing the common elements between the two arguments, `difference()` performs a subtraction of the `subset` elements from the `superset`, and `complement()` returns a `Set` of all the elements not in the intersection. As part of a simple example showing the effects of these methods, here's an `enum` containing different names of watercolors:

```
// generics/watercolors/Watercolors.java
package generics.watercolors;

public enum Watercolors {
  ZINC, LEMON_YELLOW, MEDIUM_YELLOW, DEEP_YELLOW,
  ORANGE, BRILLIANT_RED, CRIMSON, MAGENTA,
  ROSE_MADDER, VIOLET, CERULEAN_BLUE_HUE,
  PHTHALO_BLUE, ULTRAMARINE, COBALT_BLUE_HUE,
  PERMANENT_GREEN, VIRIDIAN_HUE, SAP_GREEN,
  YELLOW_OCHRE, BURNT_SIENNA, RAW_UMBER,
  BURNT_UMBER, PAYNES_GRAY, IVORY_BLACK
}
```

For convenience (so all the names don't have to be qualified), this is imported statically into the following example. This example uses the `EnumSet` for easy creation of `Set`s from `enum`s. (You'll learn more about `EnumSet` in the Enumerations chapter.) Here, the `static` method `EnumSet.range()` is given the first and last elements of the range to create in the resulting `Set`:

```
// generics/WatercolorSets.java
import generics.watercolors.*;
import java.util.*;
import static onjava.Sets.*;
import static generics.watercolors.Watercolors.*;

public class WatercolorSets {
  public static void main(String[] args) {
    Set<Watercolors> set1 =
      EnumSet.range(BRILLIANT_RED, VIRIDIAN_HUE);
    Set<Watercolors> set2 =
      EnumSet.range(CERULEAN_BLUE_HUE, BURNT_UMBER);
    System.out.println("set1: " + set1);
    System.out.println("set2: " + set2);
    System.out.println(
      "union(set1, set2): " + union(set1, set2));
    Set<Watercolors> subset = intersection(set1, set2);
    System.out.println(
      "intersection(set1, set2): " + subset);
    System.out.println("difference(set1, subset): " +
      difference(set1, subset));
    System.out.println("difference(set2, subset): " +
      difference(set2, subset));
    System.out.println("complement(set1, set2): " +
      complement(set1, set2));
  }
}
/* Output:
set1: [BRILLIANT_RED, CRIMSON, MAGENTA, ROSE_MADDER,
VIOLET, CERULEAN_BLUE_HUE, PHTHALO_BLUE, ULTRAMARINE,
COBALT_BLUE_HUE, PERMANENT_GREEN, VIRIDIAN_HUE]
set2: [CERULEAN_BLUE_HUE, PHTHALO_BLUE, ULTRAMARINE,
COBALT_BLUE_HUE, PERMANENT_GREEN, VIRIDIAN_HUE,
SAP_GREEN, YELLOW_OCHRE, BURNT_SIENNA, RAW_UMBER,
BURNT_UMBER]
union(set1, set2): [MAGENTA, COBALT_BLUE_HUE, VIOLET,
VIRIDIAN_HUE, BURNT_SIENNA, ULTRAMARINE,
CERULEAN_BLUE_HUE, BURNT_UMBER, BRILLIANT_RED,
PHTHALO_BLUE, YELLOW_OCHRE, SAP_GREEN, CRIMSON,
ROSE_MADDER, RAW_UMBER, PERMANENT_GREEN]
intersection(set1, set2): [COBALT_BLUE_HUE,
VIRIDIAN_HUE, ULTRAMARINE, CERULEAN_BLUE_HUE,
```

```
PHTHALO_BLUE, PERMANENT_GREEN]
difference(set1, subset): [CRIMSON, MAGENTA, VIOLET,
ROSE_MADDER, BRILLIANT_RED]
difference(set2, subset): [BURNT_SIENNA, BURNT_UMBER,
YELLOW_OCHRE, RAW_UMBER, SAP_GREEN]
complement(set1, set2): [MAGENTA, VIOLET, BURNT_SIENNA,
BURNT_UMBER, BRILLIANT_RED, YELLOW_OCHRE, SAP_GREEN,
CRIMSON, ROSE_MADDER, RAW_UMBER]
*/
```

The following example uses `Sets.difference()` to show the method differences between various `Collection` and `Map` classes in `java.util`:

```
// onjava/CollectionMethodDifferences.java
// {java onjava.CollectionMethodDifferences}
package onjava;
import java.lang.reflect.*;
import java.util.*;
import java.util.stream.*;

public class CollectionMethodDifferences {
  static Set<String> methodSet(Class<?> type) {
    return Arrays.stream(type.getMethods())
      .map(Method::getName)
      .collect(Collectors.toCollection(TreeSet::new));
  }
  static void interfaces(Class<?> type) {
    System.out.print("Interfaces in " +
      type.getSimpleName() + ": ");
    System.out.println(
      Arrays.stream(type.getInterfaces())
        .map(Class::getSimpleName)
        .collect(Collectors.toList()));
  }
  static Set<String> object = methodSet(Object.class);
  static { object.add("clone"); }
  static void
  difference(Class<?> superset, Class<?> subset) {
    System.out.print(superset.getSimpleName() +
      " extends " + subset.getSimpleName() +
      ", adds: ");
    Set<String> comp = Sets.difference(
```

```
      methodSet(superset), methodSet(subset));
    comp.removeAll(object); // Ignore 'Object' methods
    System.out.println(comp);
    interfaces(superset);
  }
  public static void main(String[] args) {
    System.out.println("Collection: " +
      methodSet(Collection.class));
    interfaces(Collection.class);
    difference(Set.class, Collection.class);
    difference(HashSet.class, Set.class);
    difference(LinkedHashSet.class, HashSet.class);
    difference(TreeSet.class, Set.class);
    difference(List.class, Collection.class);
    difference(ArrayList.class, List.class);
    difference(LinkedList.class, List.class);
    difference(Queue.class, Collection.class);
    difference(PriorityQueue.class, Queue.class);
    System.out.println("Map: " + methodSet(Map.class));
    difference(HashMap.class, Map.class);
    difference(LinkedHashMap.class, HashMap.class);
    difference(SortedMap.class, Map.class);
    difference(TreeMap.class, Map.class);
  }
}
/* Output:
Collection: [add, addAll, clear, contains, containsAll,
equals, forEach, hashCode, isEmpty, iterator,
parallelStream, remove, removeAll, removeIf, retainAll,
size, spliterator, stream, toArray]
Interfaces in Collection: [Iterable]
Set extends Collection, adds: []
Interfaces in Set: [Collection]
HashSet extends Set, adds: []
Interfaces in HashSet: [Set, Cloneable, Serializable]
LinkedHashSet extends HashSet, adds: []
Interfaces in LinkedHashSet: [Set, Cloneable,
Serializable]
TreeSet extends Set, adds: [headSet,
descendingIterator, descendingSet, pollLast, subSet,
floor, tailSet, ceiling, last, lower, comparator,
pollFirst, first, higher]
```

```
Interfaces in TreeSet: [NavigableSet, Cloneable,
Serializable]
List extends Collection, adds: [replaceAll, get,
indexOf, subList, set, sort, lastIndexOf, listIterator]
Interfaces in List: [Collection]
ArrayList extends List, adds: [trimToSize,
ensureCapacity]
Interfaces in ArrayList: [List, RandomAccess,
Cloneable, Serializable]
LinkedList extends List, adds: [offerFirst, poll,
getLast, offer, getFirst, removeFirst, element,
removeLastOccurrence, peekFirst, peekLast, push,
pollFirst, removeFirstOccurrence, descendingIterator,
pollLast, removeLast, pop, addLast, peek, offerLast,
addFirst]
Interfaces in LinkedList: [List, Deque, Cloneable,
Serializable]
Queue extends Collection, adds: [poll, peek, offer,
element]
Interfaces in Queue: [Collection]
PriorityQueue extends Queue, adds: [comparator]
Interfaces in PriorityQueue: [Serializable]
Map: [clear, compute, computeIfAbsent,
computeIfPresent, containsKey, containsValue, entrySet,
equals, forEach, get, getOrDefault, hashCode, isEmpty,
keySet, merge, put, putAll, putIfAbsent, remove,
replace, replaceAll, size, values]
HashMap extends Map, adds: []
Interfaces in HashMap: [Map, Cloneable, Serializable]
LinkedHashMap extends HashMap, adds: []
Interfaces in LinkedHashMap: [Map]
SortedMap extends Map, adds: [lastKey, subMap,
comparator, firstKey, headMap, tailMap]
Interfaces in SortedMap: [Map]
TreeMap extends Map, adds: [descendingKeySet,
navigableKeySet, higherEntry, higherKey, floorKey,
subMap, ceilingKey, pollLastEntry, firstKey, lowerKey,
headMap, tailMap, lowerEntry, ceilingEntry,
descendingMap, pollFirstEntry, lastKey, firstEntry,
floorEntry, comparator, lastEntry]
Interfaces in TreeMap: [NavigableMap, Cloneable,
Serializable]
```

```
*/
```

The output of this program was used in the "Summary" section of the Collections chapter.

Building Complex Models

An important benefit of generics is the ability to simply and safely create complex models. For example, we can easily create a `List` of tuples:

```
// generics/TupleList.java
// Combining generic types to make complex generic types
import java.util.*;
import onjava.*;
import java.util.stream.*;

public class TupleList<A, B, C, D>
extends ArrayList<Tuple4<A, B, C, D>> {
  public static void main(String[] args) {
    TupleList<Vehicle, Amphibian, String, Integer> tl =
      new TupleList<>();
    tl.add(TupleTest2.h());
    tl.add(TupleTest2.h());
    tl.forEach(System.out::println);
  }
}
/* Output:
(Vehicle@ec7777, Amphibian@107d329, hi, 47)
(Vehicle@1629346, Amphibian@4b9385, hi, 47)
*/
```

This produces a fairly powerful data structure without too much code.

Here's a second example. Even though each class is a building block, the total has many parts. Here, the model is a retail store with aisles, shelves and products:

```
// generics/Store.java
// Building a complex model using generic collections
import java.util.*;
import java.util.function.*;
import onjava.*;

class Product {
  private final int id;
  private String description;
  private double price;
  Product(int idNumber, String descr, double price) {
    id = idNumber;
    description = descr;
    this.price = price;
    System.out.println(toString());
  }
  @Override public String toString() {
    return id + ": " + description +
      ", price: $" + price;
  }
  public void priceChange(double change) {
    price += change;
  }
  public static Supplier<Product> generator =
    new Supplier<Product>() {
      private Random rand = new Random(47);
      @Override public Product get() {
        return new Product(rand.nextInt(1000), "Test",
          Math.round(
            rand.nextDouble() * 1000.0) + 0.99);
      }
    };
}

class Shelf extends ArrayList<Product> {
  Shelf(int nProducts) {
    Suppliers.fill(this, Product.generator, nProducts);
  }
}

class Aisle extends ArrayList<Shelf> {
  Aisle(int nShelves, int nProducts) {
```

```
    for(int i = 0; i < nShelves; i++)
      add(new Shelf(nProducts));
  }
}

class CheckoutStand {}
class Office {}

public class Store extends ArrayList<Aisle> {
  private ArrayList<CheckoutStand> checkouts =
    new ArrayList<>();
  private Office office = new Office();
  public Store(
    int nAisles, int nShelves, int nProducts) {
    for(int i = 0; i < nAisles; i++)
      add(new Aisle(nShelves, nProducts));
  }
  @Override public String toString() {
    StringBuilder result = new StringBuilder();
    for(Aisle a : this)
      for(Shelf s : a)
        for(Product p : s) {
          result.append(p);
          result.append("\n");
        }
    return result.toString();
  }
  public static void main(String[] args) {
    System.out.println(new Store(5, 4, 3));
  }
}
/* Output: (First 8 Lines)
258: Test, price: $400.99
861: Test, price: $160.99
868: Test, price: $417.99
207: Test, price: $268.99
551: Test, price: $114.99
278: Test, price: $804.99
520: Test, price: $554.99
140: Test, price: $530.99
                  ...
*/
```

`Store.toString()` shows the result: many layers of collections that are nonetheless type-safe and manageable. What's impressive is it is not intellectually prohibitive to assemble such a model.

`Shelf` uses `Suppliers.fill()`, a utility that takes a `Collection` (the first argument) and fills it using a `Supplier` (the second argument) with a number `n` (the third argument) of elements. The methods in the `Suppliers` class all perform some variation of filling, and are used in other examples in this chapter. The class is defined at the end of the chapter.

The Mystery of Erasure

As you begin to delve more deeply into generics, there are issues that won't initially make sense. For example, although you can say `ArrayList.class`, you cannot say `ArrayList<Integer>.class`. And consider the following:

```
// generics/ErasedTypeEquivalence.java
import java.util.*;

public class ErasedTypeEquivalence {
  public static void main(String[] args) {
    Class c1 = new ArrayList<String>().getClass();
    Class c2 = new ArrayList<Integer>().getClass();
    System.out.println(c1 == c2);
  }
}
/* Output:
true
*/
```

`ArrayList<String>` and `ArrayList<Integer>` should be distinct types. Different types behave differently. If you try, for example, to put an `Integer` into an `ArrayList<String>`, you get different behavior (it fails) than if you put an `Integer` into an `ArrayList<Integer>` (it succeeds). And yet the above program suggests they are the same type.

Here's an example that adds to this puzzle:

```
// generics/LostInformation.java
import java.util.*;

class Frob {}
class Fnorkle {}
class Quark<Q> {}
class Particle<POSITION, MOMENTUM> {}

public class LostInformation {
  public static void main(String[] args) {
    List<Frob> list = new ArrayList<>();
    Map<Frob, Fnorkle> map = new HashMap<>();
    Quark<Fnorkle> quark = new Quark<>();
    Particle<Long, Double> p = new Particle<>();
    System.out.println(Arrays.toString(
      list.getClass().getTypeParameters()));
    System.out.println(Arrays.toString(
      map.getClass().getTypeParameters()));
    System.out.println(Arrays.toString(
      quark.getClass().getTypeParameters()));
    System.out.println(Arrays.toString(
      p.getClass().getTypeParameters()));
  }
}
/* Output:
[E]
[K, V]
[Q]
[POSITION, MOMENTUM]
*/
```

According to the JDK documentation, `Class.getTypeParameters()` "returns an array of `TypeVariable` objects that represent the type variables declared by the generic declaration..." This suggests you can discover the parameter types. However, as the output shows, you only find out about identifiers used as the parameter placeholders—not so interesting.

The cold truth is:

> There's no information about generic parameter types available inside generic code.

Thus, you can know things like the identifier of the type parameter and the bounds of the generic type—you just can't know the actual type parameter(s) used to create a particular instance. This fact, especially frustrating if you're coming from C++, is the most fundamental issue you must deal with when working with Java generics.

Java generics are implemented using *erasure*. This means any specific type information is erased when you use a generic. Inside the generic, the only thing you know is that you're using an object. So `List<String>` and `List<Integer>` *are*, in fact, the same type at run time. Both forms are "erased" to their *raw type*, `List`. Understanding erasure and how you must deal with it is one of the biggest hurdles you face when learning Java generics. We explore erasure in this section.

The C++ Approach

Here's a C++ example which uses *templates*. The syntax for parameterized types is similar because Java took inspiration from C++:

```
// generics/Templates.cpp
#include <iostream>
using namespace std;

template<class T> class Manipulator {
  T obj;
public:
  Manipulator(T x) { obj = x; }
  void manipulate() { obj.f(); }
};

class HasF {
public:
  void f() { cout << "HasF::f()" << endl; }
};

int main() {
  HasF hf;
  Manipulator<HasF> manipulator(hf);
  manipulator.manipulate();
}
/* Output:
HasF::f()
*/
```

The `Manipulator` class stores an object of type `T`. The `manipulate()` method calls a method `f()` on `obj`. How can it know that the `f()` method exists for the type parameter `T`? The C++ compiler checks when you instantiate the template, so at the point of instantiation of `Manipulator<HasF>`, it sees that `HasF` has a method `f()`. If it were not the case, you'd get a compile-time error, preserving type safety.

Writing this kind of code in C++ is straightforward because when a template is instantiated, the template code knows the type of its template parameters. Java generics are different. Here's the translation of `HasF`:

```
// generics/HasF.java

public class HasF {
  public void f() {
    System.out.println("HasF.f()");
  }
}
```

If we take the rest of the example and translate it to Java, it won't compile:

```
// generics/Manipulation.java
// {WillNotCompile}

class Manipulator<T> {
  private T obj;
  Manipulator(T x) { obj = x; }
  // Error: cannot find symbol: method f():
  public void manipulate() { obj.f(); }
}

public class Manipulation {
  public static void main(String[] args) {
    HasF hf = new HasF();
    Manipulator<HasF> manipulator =
      new Manipulator<>(hf);
    manipulator.manipulate();
  }
}
```

Because of erasure, the Java compiler can't map the requirement that `manipulate()` must call `f()` on `obj` to the fact that `HasF` has a method `f()`. To call `f()`, we must

assist the generic class by giving it a *bound* that tells the compiler to only accept types that conform to that bound. This reuses the `extends` keyword. Because of the bound, the following compiles:

```
// generics/Manipulator2.java

class Manipulator2<T extends HasF> {
  private T obj;
  Manipulator2(T x) { obj = x; }
  public void manipulate() { obj.f(); }
}
```

The bound `<T extends HasF>` says that `T` must be of type `HasF` or something derived from `HasF`. If this is true, it is safe to call `f()` on `obj`.

We say that a generic type parameter *erases to its first bound* (multiple bounds are possible, as you shall see later). We also talk about *erasure of the type parameter*. The compiler actually replaces the type parameter with its erasure, so in the above case, `T` erases to `HasF`, which is the same as replacing `T` with `HasF` in the class body.

You might correctly observe that in `Manipulator2.java`, generics do not contribute anything. You can just as easily perform the erasure yourself and produce a class without generics:

```
// generics/Manipulator3.java

class Manipulator3 {
  private HasF obj;
  Manipulator3(HasF x) { obj = x; }
  public void manipulate() { obj.f(); }
}
```

This brings up an important point: Generics are only useful for type parameters more "generic" than a specific type (and all its subtypes)—that is, when you want code to work across multiple classes. As a result, the type parameters and their application in useful generic code will usually be more complex than simple class replacement. However, you can't just say that anything of the form `<T extends HasF>` is therefore flawed. For example, if a class has a method that returns `T`, generics are helpful, because they will then return the exact type:

```
// generics/ReturnGenericType.java

class ReturnGenericType<T extends HasF> {
  private T obj;
  ReturnGenericType(T x) { obj = x; }
  public T get() { return obj; }
}
```

You must look at all your code and determine whether it is "complex enough" to warrant generics.

We'll look at bounds in more detail later in the chapter.

Migration Compatibility

To allay any potential confusion about erasure, you must clearly understand that it is *not* a language feature. It is a compromise in the implementation of Java generics, necessary because generics were not made part of the language from the beginning. This compromise will cause you pain, so get used to it early and understand why it's there.

If generics had been part of Java 1.0, the feature would not have been implemented using erasure—it would have used *reification* to retain the type parameters as first-class entities, so you would perform type-based language and reflective operations on type parameters. You'll see later in this chapter that erasure reduces the "genericity" of generics. Generics are still useful in Java, just not as useful as they could be, and the reason is erasure.

In an erasure-based implementation, generic types are treated as second-class types that cannot be used in some important contexts. The generic types are present only during static type checking, after which every generic type in the program is erased by replacing it with a non-generic upper bound. For example, type annotations such as `List<T>` are erased to `List`, and ordinary type variables are erased to `Object` unless a bound is specified.

The core motivation for erasure is that you can use generified clients with non-generified libraries, and vice versa. This is often called *migration compatibility*. In the ideal world, everything would be generified on some designated day. In reality, even if programmers are only writing generic code, they must deal with non-generic

libraries written before Java 5. The authors of those libraries might never have the incentive to generify their code, or they might just take their time in getting to it.

So Java generics must not only support *backward compatibility*—existing code and class files are still legal, and continue to mean what they meant before—but must also support migration compatibility, so libraries can become generic at their own pace, and when a library does become generic, it doesn't break code and applications that depend upon it. After deciding this was the goal, the Java designers and the various groups working on the problem decided that erasure was the only feasible solution. Erasure enables this migration towards generics by allowing non-generic code to coexist with generic code.

For example, suppose an application uses two libraries, `X` and `Y`, and `Y` uses library `Z`. With the advent of Java 5, the creators of this application and these libraries will probably, eventually, migrate to generics. Each of them, however, will have different motivations and constraints as to when that migration happens. To achieve migration compatibility, each library and application must be independent of all the others in terms of whether generics are used. Thus, they cannot detect whether other libraries are or are not using generics. Ergo, the evidence that a particular library is using generics must be "erased."

Without some kind of migration path, all the libraries that had been built up over time stood the chance of being cut off from the developers that chose to move to Java generics. Libraries are arguably the part of a programming language that has the greatest productivity impact, so this was not an acceptable cost. Whether or not erasure was the best or only migration path is something that only time will tell.

The Problem with Erasure

So the primary justification for erasure is the transition process from non-generified code to generified code, and to incorporate generics into the language without breaking existing libraries. Erasure allows you to continue using existing non-generic client code unchanged, until clients are ready to rewrite code for generics. This is a noble motivation, because it doesn't suddenly break all existing code.

The cost of erasure is significant. Generic types cannot be used in operations that explicitly refer to runtime types, such as casts, `instanceof` operations, and `new` expressions. Because all type information about parameters is lost, when writing

generic code you must constantly remind yourself it only *appears* you have type information about a parameter.

Consider a piece of code like this:

```
class Foo<T> {
  T var;
}
```

It appears that when you create an instance of `Foo`:

```
Foo<Cat> f = new Foo<>();
```

the code in `class Foo` ought to know it is now working with a `Cat`. The syntax strongly suggests that the type `T` is substituted everywhere throughout the class, as it is in C++. But it isn't, and you must remind yourself, "No, it's just an `Object`," whenever you're writing the code for the class.

In addition, erasure and migration compatibility mean that using generics is not enforced when you might want it to be:

```
// generics/ErasureAndInheritance.java

class GenericBase<T> {
  private T element;
  public void set(T arg) { element = arg; }
  public T get() { return element; }
}

class Derived1<T> extends GenericBase<T> {}

class Derived2 extends GenericBase {} // No warning

// class Derived3 extends GenericBase<?> {}
// Strange error:
//   unexpected type
//   required: class or interface without bounds

public class ErasureAndInheritance {
  @SuppressWarnings("unchecked")
  public static void main(String[] args) {
    Derived2 d2 = new Derived2();
```

```
    Object obj = d2.get();
    d2.set(obj); // Warning here!
  }
}
```

`Derived2` inherits from `GenericBase` with no generic parameters, and the compiler doesn't issue a warning. The warning doesn't occur until `set()` is called.

To turn off the warning, Java provides an annotation, the one you see in the listing:

```
@SuppressWarnings("unchecked")
```

This is placed on the method that generates the warning, rather than the entire class. It's best to be as "focused" as possible when you turn off a warning, so you don't accidentally cloak a real problem by turning off warnings too broadly.

Presumably, the error produced by `Derived3` means the compiler expects a raw base class.

Add to this the extra effort of managing bounds when you treat your type parameter as more than just an `Object`, and you have far more effort for much less payoff than you get in parameterized types with languages like C++, Ada or Eiffel. This is not to say that those languages in general buy you more than Java does for the majority of programming problems, but rather that their parameterized type mechanisms are more flexible and powerful than Java's.

The Action at the Boundaries

Because of erasure, I find the most confusing aspect of generics is that you can represent things that have no meaning. For example:

```
// generics/ArrayMaker.java
import java.lang.reflect.*;
import java.util.*;

public class ArrayMaker<T> {
  private Class<T> kind;
  public ArrayMaker(Class<T> kind) { this.kind = kind; }
  @SuppressWarnings("unchecked")
  T[] create(int size) {
    return (T[])Array.newInstance(kind, size);
  }
  public static void main(String[] args) {
    ArrayMaker<String> stringMaker =
      new ArrayMaker<>(String.class);
    String[] stringArray = stringMaker.create(9);
    System.out.println(Arrays.toString(stringArray));
  }
}
/* Output:
[null, null, null, null, null, null, null, null, null]
*/
```

Even though `kind` is stored as `Class<T>`, erasure means it is actually just stored as a `Class`, with no parameter. So, when you do something with it, as in creating an array, `Array.newInstance()` doesn't actually have the type information that's implied in `kind`. It cannot produce the specific result, which must therefore be cast, which produces a warning you cannot satisfy.

Note that using `Array.newInstance()` is the recommended approach for creating arrays in generics.

It's different if we create a collection instead of an array:

```
// generics/ListMaker.java
import java.util.*;

public class ListMaker<T> {
  List<T> create() { return new ArrayList<>(); }
  public static void main(String[] args) {
    ListMaker<String> stringMaker = new ListMaker<>();
    List<String> stringList = stringMaker.create();
  }
}
```

The compiler gives no warnings, even though we know (from erasure) that the `<T>` in `new ArrayList<>()` inside `create()` is removed—at runtime there's no `<T>` inside the class, so it seems meaningless. But if you follow this idea and change the expression to `new ArrayList()`, the compiler gives a warning.

Is it really meaningless here? What if you were to put some objects in the `List` while creating it, like this:

```
// generics/FilledList.java
import java.util.*;
import java.util.function.*;
import onjava.*;

public class FilledList<T> extends ArrayList<T> {
  FilledList(Supplier<T> gen, int size) {
    Suppliers.fill(this, gen, size);
  }
  public FilledList(T t, int size) {
    for(int i = 0; i < size; i++)
      this.add(t);
  }
  public static void main(String[] args) {
    List<String> list = new FilledList<>("Hello", 4);
    System.out.println(list);
    // Supplier version:
    List<Integer> ilist = new FilledList<>(() -> 47, 4);
    System.out.println(ilist);
  }
}
/* Output:
[Hello, Hello, Hello, Hello]
```

```
[47, 47, 47, 47]
*/
```

Even though the compiler is unable to know anything about `T` inside `add()`, it can still ensure—at compile time—that what you put into the `FilledList` is of type `T`. Thus, even though erasure removes the information about the actual type inside a method or class, the compiler can still ensure internal consistency in the way that the type is used within the method or class.

Because erasure removes type information in the body of a method, what matters at run time is the *boundaries*: the points where objects enter and leave a method. These are the points at which the compiler performs type checks at compile time, and inserts casting code. Consider the following non-generic example:

```
// generics/SimpleHolder.java

public class SimpleHolder {
  private Object obj;
  public void set(Object obj) { this.obj = obj; }
  public Object get() { return obj; }
  public static void main(String[] args) {
    SimpleHolder holder = new SimpleHolder();
    holder.set("Item");
    String s = (String)holder.get();
  }
}
```

If we decompile the result with `javap -c SimpleHolder`, we get (after editing):

```
public void set(java.lang.Object);
   0:    aload_0
   1:    aload_1
   2:    putfield #2; //Field obj:Object;
   5:    return

public java.lang.Object get();
   0:    aload_0
   1:    getfield #2; //Field obj:Object;
   4:    areturn

public static void main(java.lang.String[]);
```

```
0:    new #3; //class SimpleHolder
3:    dup
4:    invokespecial #4; //Method "<init>":()V
7:    astore_1
8:    aload_1
9:    ldc #5; //String Item
11:   invokevirtual #6; //Method set:(Object;)V
14:   aload_1
15:   invokevirtual #7; //Method get:()Object;
18:   checkcast #8; //class java/lang/String
21:   astore_2
22:   return
```

The `set()` and `get()` methods store and produce the value, and the cast is checked at the point of the call to `get()`.

Now incorporate generics into the above code:

```
// generics/GenericHolder2.java

public class GenericHolder2<T> {
  private T obj;
  public void set(T obj) { this.obj = obj; }
  public T get() { return obj; }
  public static void main(String[] args) {
    GenericHolder2<String> holder =
      new GenericHolder2<>();
    holder.set("Item");
    String s = holder.get();
  }
}
```

The need for the cast from `get()` has disappeared, but we also know that the value passed to `set()` is type-checked at compile time. Here are the relevant bytecodes:

```
public void set(java.lang.Object);
   0:    aload_0
   1:    aload_1
   2:    putfield #2; //Field obj:Object;
   5:    return

public java.lang.Object get();
   0:    aload_0
   1:    getfield #2; //Field obj:Object;
   4:    areturn

public static void main(java.lang.String[]);
   0:    new #3; //class GenericHolder2
   3:    dup
   4:    invokespecial #4; //Method "<init>":()V
   7:    astore_1
   8:    aload_1
   9:    ldc #5; //String Item
   11:   invokevirtual #6; //Method set:(Object;)V
   14:   aload_1
   15:   invokevirtual #7; //Method get:()Object;
   18:   checkcast #8; //class java/lang/String
   21:   astore_2
   22:   return
```

The resulting code is identical. The extra work of checking the incoming type in `set()` is free, because it is performed by the compiler. And the cast for the outgoing value of `get()` is still there, but it's no less than you'd do yourself—and it's automatically inserted by the compiler, so the code you write (and read) is less noisy.

`get()` and `set()` produce the same bytecodes so this tells us that all the action in generics happens at the boundaries—the extra compile-time check for incoming values, and the inserted cast for outgoing values. It helps to counter the confusion of erasure to remember that "the boundaries are where the action takes place."

Compensating for Erasure

With erasure, we lose the ability to perform certain operations in generic code. Anything that requires knowing the exact type at run time won't work:

```
// generics/Erased.java
// {WillNotCompile}

public class Erased<T> {
  private final int SIZE = 100;
  public void f(Object arg) {

    // error: illegal generic type for instanceof
    if(arg instanceof T) {}

    // error: unexpected type
    T var = new T();

    // error: generic array creation
    T[] array = new T[SIZE];

    // warning: [unchecked] unchecked cast
    T[] array = (T[])new Object[SIZE];

  }
}
```

Occasionally you can program around these issues, but sometimes you must compensate for erasure by introducing a *type tag*. This means explicitly passing a `Class` object for your type to use it in type expressions.

For example, the attempt to use `instanceof` in the previous program fails because the type information was erased. A type tag enables a dynamic `isInstance()`:

```
// generics/ClassTypeCapture.java

class Building {}
class House extends Building {}

public class ClassTypeCapture<T> {
  Class<T> kind;
  public ClassTypeCapture(Class<T> kind) {
    this.kind = kind;
  }
  public boolean f(Object arg) {
    return kind.isInstance(arg);
  }
```

```
  public static void main(String[] args) {
    ClassTypeCapture<Building> ctt1 =
      new ClassTypeCapture<>(Building.class);
    System.out.println(ctt1.f(new Building()));
    System.out.println(ctt1.f(new House()));
    ClassTypeCapture<House> ctt2 =
      new ClassTypeCapture<>(House.class);
    System.out.println(ctt2.f(new Building()));
    System.out.println(ctt2.f(new House()));
  }
}
/* Output:
true
true
false
true
*/
```

The compiler ensures that the type tag matches the generic argument.

Creating Instances of Types

The attempt to create a `new T()` in `Erased.java` won't work, partly because of erasure, and partly because the compiler cannot verify that `T` has zero-argument constructor. But in C++ this operation is natural, straightforward, and safe (it's checked at compile time):

```
// generics/InstantiateGenericType.cpp
// C++, not Java!

template<class T> class Foo {
  T x; // Create a field of type T
  T* y; // Pointer to T
public:
  // Initialize the pointer:
  Foo() { y = new T(); }
};

class Bar {};
```

```
int main() {
  Foo<Bar> fb;
  Foo<int> fi; // ... and it works with primitives
}
```

The solution in Java is to pass in a factory object, and use that to make the new instance. A convenient factory object is just the `Class` object, so if you use a type tag, you can use `newInstance()` to create a new object of that type:

```
// generics/InstantiateGenericType.java
import java.util.function.*;
import java.lang.reflect.InvocationTargetException;

class ClassAsFactory<T> implements Supplier<T> {
  Class<T> kind;
  ClassAsFactory(Class<T> kind) {
    this.kind = kind;
  }
  @Override public T get() {
    try {
      return kind.getConstructor().newInstance();
    } catch(Exception e) {
      throw new RuntimeException(e);
    }
  }
}

class Employee {
  public Employee() {}
  @Override public String toString() {
    return "Employee";
  }
}

public class InstantiateGenericType {
  public static void main(String[] args) {
    ClassAsFactory<Employee> fe =
      new ClassAsFactory<>(Employee.class);
    System.out.println(fe.get());
    ClassAsFactory<Integer> fi =
      new ClassAsFactory<>(Integer.class);
    try {
```

```
      System.out.println(fi.get());
    } catch(Exception e) {
      System.out.println(e.getMessage());
    }
  }
}
/* Output:
Employee
java.lang.NoSuchMethodException:
java.lang.Integer.<init>()
*/
```

This compiles, but fails with `ClassAsFactory<Integer>` because `Integer` doesn't have a zero-argument constructor. Because the error is not caught at compile time, this approach is frowned upon by the language creators. They suggest instead that you use an explicit factory (`Supplier`) and constrain the type so it only takes a class that implements this factory. Here are two different ways to create the factory:

```
// generics/FactoryConstraint.java
import java.util.*;
import java.util.function.*;
import onjava.*;

class IntegerFactory implements Supplier<Integer> {
  private int i = 0;
  @Override public Integer get() {
    return ++i;
  }
}

class Widget {
  private int id;
  Widget(int n) { id = n; }
  @Override public String toString() {
    return "Widget " + id;
  }
  public static
  class Factory implements Supplier<Widget> {
    private int i = 0;
    @Override
    public Widget get() { return new Widget(++i); }
  }
```

```
}

class Fudge {
  private static int count = 1;
  private int n = count++;
  @Override public String toString() {
    return "Fudge " + n;
  }
}

class Foo2<T> {
  private List<T> x = new ArrayList<>();
  Foo2(Supplier<T> factory) {
    Suppliers.fill(x, factory, 5);
  }
  @Override public String toString() {
    return x.toString();
  }
}

public class FactoryConstraint {
  public static void main(String[] args) {
    System.out.println(
      new Foo2<>(new IntegerFactory()));
    System.out.println(
      new Foo2<>(new Widget.Factory()));
    System.out.println(
      new Foo2<>(Fudge::new));
  }
}
/* Output:
[1, 2, 3, 4, 5]
[Widget 1, Widget 2, Widget 3, Widget 4, Widget 5]
[Fudge 1, Fudge 2, Fudge 3, Fudge 4, Fudge 5]
*/
```

`IntegerFactory` is itself a factory by implementing `Supplier<Integer>`. `Widget` contains an inner class which is a factory. And notice that `Fudge` does not do anything factory-like, and yet passing `Fudge::new` still produces factory behavior, because the compiler translates a call to the functional method `::new` into a call to `get()`.

Another approach is the *Template Method* design pattern. In the following example, `create()` is the Template Method, overridden in the subclass to produce an object of that type:

```
// generics/CreatorGeneric.java

abstract class GenericWithCreate<T> {
  final T element;
  GenericWithCreate() { element = create(); }
  abstract T create();
}

class X {}

class XCreator extends GenericWithCreate<X> {
  @Override X create() { return new X(); }
  void f() {
    System.out.println(
      element.getClass().getSimpleName());
  }
}

public class CreatorGeneric {
  public static void main(String[] args) {
    XCreator xc = new XCreator();
    xc.f();
  }
}
/* Output:
X
*/
```

`GenericWithCreate` contains the `element` field, and forces its initialization via the zero-argument constructor, which in turn calls the `abstract create()` method. This way creation can be defined in the subclass, at the same time the type of `T` is established.

Arrays of Generics

As you saw in `Erased.java`, you can't create arrays of generics. The general solution is to use an `ArrayList` anywhere you are tempted to create an array of generics:

```
// generics/ListOfGenerics.java
import java.util.*;

public class ListOfGenerics<T> {
  private List<T> array = new ArrayList<>();
  public void add(T item) { array.add(item); }
  public T get(int index) { return array.get(index); }
}
```

Here you get the behavior of an array but the compile-time type safety afforded by generics.

At times, you still create an array of generic types (the `ArrayList`, for example, uses arrays internally). You can define a generic *reference* to an array in a way that makes the compiler happy:

```
// generics/ArrayOfGenericReference.java

class Generic<T> {}

public class ArrayOfGenericReference {
  static Generic<Integer>[] gia;
}
```

The compiler accepts this without producing warnings. But you can never create an array of that exact type (including the type parameters), so it's a little confusing. All arrays have the same structure (size of each array slot and array layout) regardless of the type they hold so it seems like you can create an array of `Object` and cast that to the desired array type. This does in fact compile, but it produces a `ClassCastException`:

```
// generics/ArrayOfGeneric.java

public class ArrayOfGeneric {
  static final int SIZE = 100;
  static Generic<Integer>[] gia;
  @SuppressWarnings("unchecked")
  public static void main(String[] args) {
    try {
      gia = (Generic<Integer>[])new Object[SIZE];
    } catch(ClassCastException e) {
      System.out.println(e.getMessage());
    }
    // Runtime type is the raw (erased) type:
    gia = (Generic<Integer>[])new Generic[SIZE];
    System.out.println(gia.getClass().getSimpleName());
    gia[0] = new Generic<>();
    //- gia[1] = new Object(); // Compile-time error
    // Discovers type mismatch at compile time:
    //- gia[2] = new Generic<Double>();
  }
}
/* Output:
[Ljava.lang.Object; cannot be cast to [LGeneric;
Generic[]
*/
```

The problem is that arrays keep track of their actual type, and that type is established at the point of creation of the array. So even though `gia` is cast to a `Generic<Integer>[]`, that information only exists at compile time (and without the `@SuppressWarnings` annotation, you'd get a warning for that cast). At run time, it's still an array of `Object`, and that causes problems. The only way to successfully create an array of a generic type is to create a new array of the erased type, and cast that.

Let's look at a slightly more sophisticated example. Consider a simple generic wrapper around an array:

```
// generics/GenericArray.java

public class GenericArray<T> {
  private T[] array;
  @SuppressWarnings("unchecked")
  public GenericArray(int sz) {
    array = (T[])new Object[sz];
  }
  public void put(int index, T item) {
    array[index] = item;
  }
  public T get(int index) { return array[index]; }
  // Method that exposes the underlying representation:
  public T[] rep() { return array; }
  public static void main(String[] args) {
    GenericArray<Integer> gai = new GenericArray<>(10);
    try {
      Integer[] ia = gai.rep();
    } catch(ClassCastException e) {
      System.out.println(e.getMessage());
    }
    // This is OK:
    Object[] oa = gai.rep();
  }
}
/* Output:
[Ljava.lang.Object; cannot be cast to
[Ljava.lang.Integer;
*/
```

As before, we can't say `T[] array = new T[sz]`, so we create an array of objects and cast it.

The `rep()` method returns a `T[]`, which in `main()` should be an `Integer[]` for `gai`, but if you call it and try to capture the result as an `Integer[]` reference, you get a `ClassCastException`, again because the actual runtime type is `Object[]`.

If you compile `GenericArray.java` after commenting out the `@SuppressWarnings` annotation, the compiler produces a warning:

```
GenericArray.java uses unchecked or unsafe operations.
Recompile with -Xlint:unchecked for details.
```

Here, we've gotten a single warning, and we believe it's about the cast. But to really make sure, compile with `-Xlint:unchecked`:

```
GenericArray.java:7: warning: [unchecked] unchecked cast
    array = (T[])new Object[sz];
                 ^
  required: T[]
  found:    Object[]
  where T is a type-variable:
    T extends Object declared in class GenericArray
1 warning
```

It is indeed complaining about that cast. Because warnings become noise, the best we can possibly do, once we verify that a particular warning is expected, is to turn it off using `@SuppressWarnings`. That way, when a warning does appear, we'll actually investigate it.

Because of erasure, the runtime type of the array can only be `Object[]`. If we immediately cast it to `T[]`, then at compile-time the actual type of the array is lost, and the compiler can miss out on some potential error checks. Because of this, it's better to use an `Object[]` inside the collection, and add a cast to `T` when you use an array element. Let's see how that would look with the `GenericArray.java` example:

```
// generics/GenericArray2.java

public class GenericArray2<T> {
  private Object[] array;
  public GenericArray2(int sz) {
    array = new Object[sz];
  }
  public void put(int index, T item) {
    array[index] = item;
  }
  @SuppressWarnings("unchecked")
  public T get(int index) { return (T)array[index]; }
  @SuppressWarnings("unchecked")
  public T[] rep() {
    return (T[])array; // Unchecked cast
```

```
  }
  public static void main(String[] args) {
    GenericArray2<Integer> gai =
      new GenericArray2<>(10);
    for(int i = 0; i < 10; i ++)
      gai.put(i, i);
    for(int i = 0; i < 10; i ++)
      System.out.print(gai.get(i) + " ");
    System.out.println();
    try {
      Integer[] ia = gai.rep();
    } catch(Exception e) {
      System.out.println(e);
    }
  }
}
/* Output:
0 1 2 3 4 5 6 7 8 9
java.lang.ClassCastException: [Ljava.lang.Object;
cannot be cast to [Ljava.lang.Integer;
*/
```

Initially, this doesn't look very different, just that the cast has moved. Without the `@SuppressWarnings` annotations, you still get "unchecked" warnings. However, the internal representation is now `Object[]` rather than `T[]`. When `get()` is called, it casts the object to `T`, which is in fact the correct type, so that is safe. However, if you call `rep()`, it again attempts to cast the `Object[]` to a `T[]`, still incorrect, and produces a warning at compile time and an exception at run time. Thus there's no way to subvert the type of the underlying array, which can only be `Object[]`. The advantage of treating `array` internally as `Object[]` instead of `T[]` is it's less likely you'll forget the runtime type of the array and accidentally introduce a bug (although the majority, and perhaps all, of such bugs would be rapidly detected at run time).

For new code, pass in a type token. In that case, the `GenericArray` looks like this:

```
// generics/GenericArrayWithTypeToken.java
import java.lang.reflect.*;

public class GenericArrayWithTypeToken<T> {
  private T[] array;
  @SuppressWarnings("unchecked")
  public
  GenericArrayWithTypeToken(Class<T> type, int sz) {
    array = (T[])Array.newInstance(type, sz);
  }
  public void put(int index, T item) {
    array[index] = item;
  }
  public T get(int index) { return array[index]; }
  // Expose the underlying representation:
  public T[] rep() { return array; }
  public static void main(String[] args) {
    GenericArrayWithTypeToken<Integer> gai =
      new GenericArrayWithTypeToken<>(
        Integer.class, 10);
    // This now works:
    Integer[] ia = gai.rep();
  }
}
```

The type token `Class<T>` is passed into the constructor to recover from the erasure, so we can create the actual type of array we need, although the warning from the cast must be suppressed with `@SuppressWarnings`. Once we get the actual type, we can return it and produce the desired results, as you see in `main()`. The runtime type of the array is the exact type `T[]`.

Unfortunately, if you look at the source code in the Java standard libraries, you'll see there are casts from `Object` arrays to parameterized types everywhere. For example, here's the copy-`ArrayList`-from-`Collection` constructor, after cleaning up and simplifying:

```
public ArrayList(Collection c) {
  size = c.size();
  elementData = (E[])new Object[size];
  c.toArray(elementData);
}
```

If you look through `ArrayList.java`, you'll find plenty of these casts. And what happens when we compile it?

```
Note: ArrayList.java uses unchecked or unsafe operations
Note: Recompile with -Xlint:unchecked for details.
```

Sure enough, the standard libraries produce lots of warnings. If you've worked with C, especially pre-ANSI C, you remember a particular effect of warnings: When you discover you can ignore them, you do. For that reason, it's best to not issue any kind of message from the compiler unless the programmer must do something about it.

In his weblog,[144] Neal Gafter (one of the lead developers for Java 5) points out that he was lazy when rewriting the Java libraries, and that we should not do what he did. Neal also points out that he could not fix some of the Java library code without breaking the existing interface. So even if certain idioms appear in the Java library sources, that's not necessarily the right way to do it. When you look at library code, you cannot assume it's an example to follow in your own code.

Note that the type token technique is recommended in the Java literature, such as Gilad Bracha's paper *Generics in the Java Programming Language*,[145] where he notes, "It's an idiom that's used extensively in the new APIs for manipulating annotations, for example." However, I've discovered some inconsistency in people's comfort level with this technique; some people strongly prefer the factory approach, which was presented earlier in this chapter.

Bounds

Bounds were briefly introduced earlier in the chapter. Bounds allow you to place constraints on parameter types used with generics. Although this can enforce rules

[144] Neal Gafter's Reply.
[145] See citation at the end of this chapter.

about types to which your generics are applied, a potentially more important effect is that you can call methods in your bound types.

Because erasure removes type information, the only methods you can call for an unbounded generic parameter are those available for `Object`. If, however, you are able to constrain that parameter to a subset of types, you can call the methods in that subset. To apply constraints, Java generics reuse the `extends` keyword.

It's important to understand that `extends` has a significantly different meaning in the context of generic bounds than it does ordinarily. This example shows the basics of bounds:

```
// generics/BasicBounds.java

interface HasColor { java.awt.Color getColor(); }

class WithColor<T extends HasColor> {
  T item;
  WithColor(T item) { this.item = item; }
  T getItem() { return item; }
  // The bound allows you to call a method:
  java.awt.Color color() { return item.getColor(); }
}

class Coord { public int x, y, z; }

// This fails. Class must be first, then interfaces:
// class WithColorCoord<T extends HasColor & Coord> {

// Multiple bounds:
class WithColorCoord<T extends Coord & HasColor> {
  T item;
  WithColorCoord(T item) { this.item = item; }
  T getItem() { return item; }
  java.awt.Color color() { return item.getColor(); }
  int getX() { return item.x; }
  int getY() { return item.y; }
  int getZ() { return item.z; }
}

interface Weight { int weight(); }
```

```
// As with inheritance, you can have only one
// concrete class but multiple interfaces:
class Solid<T extends Coord & HasColor & Weight> {
  T item;
  Solid(T item) { this.item = item; }
  T getItem() { return item; }
  java.awt.Color color() { return item.getColor(); }
  int getX() { return item.x; }
  int getY() { return item.y; }
  int getZ() { return item.z; }
  int weight() { return item.weight(); }
}

class Bounded
extends Coord implements HasColor, Weight {
  @Override
  public java.awt.Color getColor() { return null; }
  @Override public int weight() { return 0; }
}

public class BasicBounds {
  public static void main(String[] args) {
    Solid<Bounded> solid =
      new Solid<>(new Bounded());
    solid.color();
    solid.getY();
    solid.weight();
  }
}
```

You might observe that `BasicBounds.java` seems to contain redundancies that could be eliminated through inheritance. Here, each level of inheritance also adds bounds constraints:

```
// generics/InheritBounds.java

class HoldItem<T> {
  T item;
  HoldItem(T item) { this.item = item; }
  T getItem() { return item; }
}

class WithColor2<T extends HasColor>
extends HoldItem<T> {
  WithColor2(T item) { super(item); }
  java.awt.Color color() { return item.getColor(); }
}

class WithColorCoord2<T extends Coord & HasColor>
extends WithColor2<T> {
  WithColorCoord2(T item) {  super(item); }
  int getX() { return item.x; }
  int getY() { return item.y; }
  int getZ() { return item.z; }
}

class Solid2<T extends Coord & HasColor & Weight>
extends WithColorCoord2<T> {
  Solid2(T item) {  super(item); }
  int weight() { return item.weight(); }
}

public class InheritBounds {
  public static void main(String[] args) {
    Solid2<Bounded> solid2 =
      new Solid2<>(new Bounded());
    solid2.color();
    solid2.getY();
    solid2.weight();
  }
}
```

`HoldItem` holds an object, so this behavior is inherited into `WithColor2`, which also requires its parameter conforms to `HasColor`. `WithColorCoord2` and `Solid2` further extend the hierarchy and add bounds at each level. Now the methods are inherited and they're not repeated in each class.

Here's an example with more layers:

```
// generics/EpicBattle.java
// Bounds in Java generics
import java.util.*;

interface SuperPower {}

interface XRayVision extends SuperPower {
  void seeThroughWalls();
}

interface SuperHearing extends SuperPower {
  void hearSubtleNoises();
}

interface SuperSmell extends SuperPower {
  void trackBySmell();
}

class SuperHero<POWER extends SuperPower> {
  POWER power;
  SuperHero(POWER power) { this.power = power; }
  POWER getPower() { return power; }
}

class SuperSleuth<POWER extends XRayVision>
extends SuperHero<POWER> {
  SuperSleuth(POWER power) { super(power); }
  void see() { power.seeThroughWalls(); }
}

class
CanineHero<POWER extends SuperHearing & SuperSmell>
extends SuperHero<POWER> {
  CanineHero(POWER power) { super(power); }
  void hear() { power.hearSubtleNoises(); }
  void smell() { power.trackBySmell(); }
}

class SuperHearSmell
implements SuperHearing, SuperSmell {
```

```
  @Override public void hearSubtleNoises() {}
  @Override public void trackBySmell() {}
}

class DogPerson extends CanineHero<SuperHearSmell> {
  DogPerson() { super(new SuperHearSmell()); }
}

public class EpicBattle {
  // Bounds in generic methods:
  static <POWER extends SuperHearing>
  void useSuperHearing(SuperHero<POWER> hero) {
    hero.getPower().hearSubtleNoises();
  }
  static <POWER extends SuperHearing & SuperSmell>
  void superFind(SuperHero<POWER> hero) {
    hero.getPower().hearSubtleNoises();
    hero.getPower().trackBySmell();
  }
  public static void main(String[] args) {
    DogPerson dogPerson = new DogPerson();
    useSuperHearing(dogPerson);
    superFind(dogPerson);
    // You can do this:
    List<? extends SuperHearing> audioPeople;
    // But you can't do this:
    // List<? extends SuperHearing & SuperSmell> dogPs;
  }
}
```

Wildcards, which we shall investigate next, are limited to a single bound.

Wildcards

You've already seen some simple uses of *wildcards*—question marks in generic argument expressions—in the Collections chapter and more in the Reflection chapter. This section will explore the feature more deeply.

We'll start with an example that shows a particular behavior of arrays: You can assign an array of a derived type to an array reference of the base type:

```
// generics/CovariantArrays.java

class Fruit {}
class Apple extends Fruit {}
class Jonathan extends Apple {}
class Orange extends Fruit {}

public class CovariantArrays {
  public static void main(String[] args) {
    Fruit[] fruit = new Apple[10];
    fruit[0] = new Apple(); // OK
    fruit[1] = new Jonathan(); // OK
    // Runtime type is Apple[], not Fruit[] or Orange[]:
    try {
      // Compiler allows you to add Fruit:
      fruit[0] = new Fruit(); // ArrayStoreException
    } catch(Exception e) { System.out.println(e); }
    try {
      // Compiler allows you to add Oranges:
      fruit[0] = new Orange(); // ArrayStoreException
    } catch(Exception e) { System.out.println(e); }
  }
}
/* Output:
java.lang.ArrayStoreException: Fruit
java.lang.ArrayStoreException: Orange
*/
```

The first line in `main()` creates an array of `Apple` and assigns it to a reference to an array of `Fruit`. This makes sense—an `Apple` is a kind of `Fruit`, so an array of `Apple` should also be an array of `Fruit`.

However, if the actual array type is `Apple[]`, you can place an `Apple` or a subtype of `Apple` into the array, which in fact works at both compile time and run time. But you can also place a `Fruit` object into the array. This makes sense to the compiler, because it has a `Fruit[]` reference—why shouldn't it allow a `Fruit` object, or anything descended from `Fruit`, such as `Orange`, to be placed into the array? So at compile time, this is allowed. The runtime array mechanism, however, knows it's dealing with an `Apple[]` and throws an exception when a foreign type is placed into the array.

"Upcast" is a misnomer here. What you're really doing is assigning one array to another. The array behavior is to hold other objects, but because we are able to upcast, it's clear that the array objects can preserve the rules about the type of objects they contain. It's as if the arrays are conscious of what they are holding, so between the compile-time checks and the runtime checks, you can't abuse them.

This arrangement for arrays is not so terrible, because you *do* find out at run time that you've inserted an improper type. But one of the primary goals of generics is to move such error detection to compile time. So what happens when we try to use generic collections instead of arrays?

```
// generics/NonCovariantGenerics.java
// {WillNotCompile}
import java.util.*;

public class NonCovariantGenerics {
  // Compile Error: incompatible types:
  List<Fruit> flist = new ArrayList<Apple>();
}
```

Although you might at first read this as saying, "You can't assign a collection of `Apple` to a collection of `Fruit`," remember that generics are not just about collections. What it's really saying is, "You can't assign a generic *involving* `Apples` to a generic *involving* `Fruit`." If, as in the case of arrays, the compiler knew enough about the code to determine that collections were involved, perhaps it could give some leeway. But it doesn't know anything like that, so it refuses to allow the "upcast." But it really isn't an "upcast" anyway—a `List` of `Apple` is not a `List` of `Fruit`. A `List` of `Apple` will hold `Apples` and subtypes of `Apple`, and a `List` of `Fruit` will hold any kind of `Fruit`. Yes, including `Apples`, but that doesn't make it a `List` of `Apple`; it's still a `List` of `Fruit`. A `List` of `Apple` is not type-equivalent to a `List` of `Fruit`, even if an `Apple` is a type of `Fruit`.

The real issue is that we are talking about the type of the collection, rather than the type that the collection is holding. Unlike arrays, generics do not have built-in covariance. This is because arrays are completely defined in the language and can thus have both compile-time and runtime checks built in, but with generics, the compiler and runtime system cannot know what to do with your types and what the rules should be.

Sometimes, however, you'd like to establish some kind of upcasting relationship between the two. Wildcards produce this relationship.

```
// generics/GenericsAndCovariance.java
import java.util.*;

public class GenericsAndCovariance {
  public static void main(String[] args) {
    // Wildcards allow covariance:
    List<? extends Fruit> flist = new ArrayList<>();
    // Compile Error: can't add any type of object:
    // flist.add(new Apple());
    // flist.add(new Fruit());
    // flist.add(new Object());
    flist.add(null); // Legal but uninteresting
    // We know it returns at least Fruit:
    Fruit f = flist.get(0);
  }
}
```

The type of `flist` is now `List<? extends Fruit>`, which you can read as "a list of any type that's inherited from `Fruit`." This doesn't actually mean that the `List` will hold any type of `Fruit`, however. The wildcard refers to a definite type, so it means "some specific type which the `flist` reference doesn't specify." So the `List` that's assigned must hold some specified type such as `Fruit` or `Apple`, but to upcast to `flist`, that type is a "don't actually care."

The `List` must hold a specific `Fruit` or subtype of `Fruit`, but if you don't actually care what it is, what can you do with such a `List`? If you don't know what type the `List` is holding, how can you safely add an object? Just as with the "upcast" array in `CovariantArrays.java`, you can't, except that the compiler prevents it from happening rather than the runtime system. You discover the problem sooner.

You might argue that things have gone a bit overboard, because now you can't even add an `Apple` to a `List` you just said would hold `Apples`. Yes, but the compiler doesn't know that. A `List<? extends Fruit>` could legally point to a `List<Orange>`. Once you do this kind of "upcast," you lose the ability to pass anything in, even an `Object`.

On the other hand, if you call a method that returns `Fruit`, that's safe because you know that anything in the `List` must at least be of type `Fruit`, so the compiler allows

it.

How Smart is the Compiler?

Now, you might guess you are prevented from calling any methods that take arguments, but consider this:

```
// generics/CompilerIntelligence.java
import java.util.*;

public class CompilerIntelligence {
  public static void main(String[] args) {
    List<? extends Fruit> flist =
      Arrays.asList(new Apple());
    Apple a = (Apple)flist.get(0); // No warning
    flist.contains(new Apple()); // Argument is 'Object'
    flist.indexOf(new Apple()); // Argument is 'Object'
  }
}
```

Here, calls to `contains()` and `indexOf()` take `Apple` objects as arguments, and those are just fine. Does this mean that the compiler actually examines the code to see if a particular method modifies its object?

By looking at the documentation for `ArrayList`, we find that the compiler is not that smart. While `add()` takes an argument of the generic parameter type, `contains()` and `indexOf()` take arguments of type `Object`. So when you specify an `ArrayList<? extends Fruit>`, the argument for `add()` becomes `? extends Fruit`. From that description, the compiler cannot know which specific subtype of `Fruit` is required there, so it won't accept any type of `Fruit`. It doesn't matter if you upcast the `Apple` to a `Fruit` first—the compiler simply refuses to call a method (such as `add()`) if a wildcard is involved in the argument list.

With `contains()` and `indexOf()`, the arguments are of type `Object`, so there are no wildcards involved and the compiler allows the call. This means it's up to the generic class designer to decide which calls are "safe," and to use `Object` types for their arguments. To disallow a call when the type is used with wildcards, use the type parameter in the argument list.

A very simple `Holder` class will demonstrate:

```
// generics/Holder.java
import java.util.Objects;

public class Holder<T> {
  private T value;
  public Holder() {}
  public Holder(T val) { value = val; }
  public void set(T val) { value = val; }
  public T get() { return value; }
  @Override public boolean equals(Object o) {
    return o instanceof Holder &&
      Objects.equals(value, ((Holder)o).value);
  }
  @Override public int hashCode() {
    return Objects.hashCode(value);
  }
  public static void main(String[] args) {
    Holder<Apple> apple = new Holder<>(new Apple());
    Apple d = apple.get();
    apple.set(d);
    // Holder<Fruit> Fruit = apple; // Cannot upcast
    Holder<? extends Fruit> fruit = apple; // OK
    Fruit p = fruit.get();
    d = (Apple)fruit.get(); // Returns 'Object'
    try {
      Orange c = (Orange)fruit.get(); // No warning
    } catch(Exception e) { System.out.println(e); }
    // fruit.set(new Apple()); // Cannot call set()
    // fruit.set(new Fruit()); // Cannot call set()
    System.out.println(fruit.equals(d)); // OK
  }
}
/* Output:
java.lang.ClassCastException: Apple cannot be cast to
Orange
false
*/
```

`Holder` has a `set()` which takes a `T`, a `get()` which returns a `T`, and an `equals()` that takes an `Object`. As you've already seen, if you create a `Holder<Apple>`, you cannot upcast it to a `Holder<Fruit>`, but you can upcast to a `Holder<? extends Fruit>`. If you call `get()`, it only returns a `Fruit`—that's as much as it knows given

the "anything that extends `Fruit`" bound. If you know more about what's there, you can cast to a specific type of `Fruit` and there won't be any warning about it, but you risk a `ClassCastException`. The `set()` method won't work with either an `Apple` or a `Fruit`, because the `set()` argument is also "`? Extends Fruit`," which means it can be anything and the compiler can't verify type safety for "anything."

However, the `equals()` method works fine because it takes an `Object` instead of a `T` as an argument. Thus, the compiler is only paying attention to the types of objects that are passed and returned. It is not analyzing the code to see if you perform any actual writes or reads.

Java 7 introduced the `java.util.Objects` library for the purpose, among other things, of making it easier to create `equals()` and `hashCode()` methods. The canonical form of `equals()` shown here is described in the Appendix: Understanding `equals()` and `hashCode()`.

Contravariance

It's also possible to go the other way, and use *supertype wildcards*. Here, you say that the wildcard is bounded by any base class of a particular class, by specifying `<? super MyClass>` or even using a type parameter: `<? super T>` (although you cannot give a generic parameter a supertype bound; that is, you cannot say `<T super MyClass>`). This safely passes a typed object into a generic type. Thus, with supertype wildcards you can write into a `Collection`:

```
// generics/SuperTypeWildcards.java
import java.util.*;

public class SuperTypeWildcards {
  static void writeTo(List<? super Apple> apples) {
    apples.add(new Apple());
    apples.add(new Jonathan());
    // apples.add(new Fruit()); // Error
  }
}
```

The argument `apples` is a `List` of some type that is the base type of `Apple`; thus you know it is safe to add an `Apple` or a subtype of `Apple`. The *lower bound* is `Apple`, however, so you don't know it is safe to add `Fruit` to such a `List` because that would

allow the `List` to be opened up to the addition of non-`Apple` types, which violates static type safety.

This example provides a review of covariance and wildcards:

```
// generics/GenericReading.java
import java.util.*;

public class GenericReading {
  static List<Apple> apples =
    Arrays.asList(new Apple());
  static List<Fruit> fruit = Arrays.asList(new Fruit());
  static <T> T readExact(List<T> list) {
    return list.get(0);
  }
  // A static method adapts to each call:
  static void f1() {
    Apple a = readExact(apples);
    Fruit f = readExact(fruit);
    f = readExact(apples);
  }
  // A class type is established
  // when the class is instantiated:
  static class Reader<T> {
    T readExact(List<T> list) { return list.get(0); }
  }
  static void f2() {
    Reader<Fruit> fruitReader = new Reader<>();
    Fruit f = fruitReader.readExact(fruit);
    //- Fruit a = fruitReader.readExact(apples);
    // error: incompatible types: List<Apple>
    // cannot be converted to List<Fruit>
  }
  static class CovariantReader<T> {
    T readCovariant(List<? extends T> list) {
      return list.get(0);
    }
  }
  static void f3() {
    CovariantReader<Fruit> fruitReader =
      new CovariantReader<>();
    Fruit f = fruitReader.readCovariant(fruit);
```

```
    Fruit a = fruitReader.readCovariant(apples);
  }
  public static void main(String[] args) {
    f1(); f2(); f3();
  }
}
```

`readExact()` uses the precise type. If you use the precise type with no wildcards, you can both write and read that precise type into and out of a `List`. In addition, for the return value, the `static` generic method `readExact()` effectively "adapts" to each method call, and returns an `Apple` from a `List<Apple>` and a `Fruit` from a `List<Fruit>`, as you see in `f1()`. Thus, if you can get away with a `static` generic method, you don't necessarily need covariance if you're just reading.

With a generic class, however, the parameter is established for the class when you make an instance of that class. As shown in `f2()`, the `fruitReader` instance can read a piece of `Fruit` from a `List<Fruit>` because that is its exact type. But a `List<Apple>` should also produce `Fruit` objects, and the `fruitReader` doesn't allow this.

To fix the problem, the `CovariantReader.readCovariant()` method takes a `List<? extends T>`. It's safe to read a `T` from that list because you know that everything in that list is at least a `T`, and possibly something derived from a `T`. In `f3()` you see it's now possible to read a `Fruit` from a `List<Apple>`.

Unbounded Wildcards

The *unbounded wildcard* `<?>` appears to mean "anything," and so using an unbounded wildcard looks equivalent to using a raw type. Indeed, the compiler seems at first to agree with this assessment:

```
// generics/UnboundedWildcards1.java
import java.util.*;

public class UnboundedWildcards1 {
  static List list1;
  static List<?> list2;
  static List<? extends Object> list3;
  static void assign1(List list) {
    list1 = list;
    list2 = list;
    //- list3 = list;
    // warning: [unchecked] unchecked conversion
    // list3 = list;
    //         ^
    // required: List<? extends Object>
    // found:    List
  }
  static void assign2(List<?> list) {
    list1 = list;
    list2 = list;
    list3 = list;
  }
  static void assign3(List<? extends Object> list) {
    list1 = list;
    list2 = list;
    list3 = list;
  }
  public static void main(String[] args) {
    assign1(new ArrayList());
    assign2(new ArrayList());
    //- assign3(new ArrayList());
    // warning: [unchecked] unchecked method invocation:
    // method assign3 in class UnboundedWildcards1
    // is applied to given types
    // assign3(new ArrayList());
    //         ^
    // required: List<? extends Object>
    // found: ArrayList
    // warning: [unchecked] unchecked conversion
    // assign3(new ArrayList());
    //         ^
    // required: List<? extends Object>
```

```
    // found:    ArrayList
    // 2 warnings
    assign1(new ArrayList<>());
    assign2(new ArrayList<>());
    assign3(new ArrayList<>());
    // Both forms are acceptable as List<?>:
    List<?> wildList = new ArrayList();
    wildList = new ArrayList<>();
    assign1(wildList);
    assign2(wildList);
    assign3(wildList);
  }
}
```

There are many cases like the ones you see here where the compiler could care less whether you use a raw type or `<?>`. In those cases, `<?>` can be thought of as a decoration; and yet it is valuable because, in effect, it says, "I wrote this code with Java generics in mind, and I don't mean here I'm using a raw type, but that in this case the generic parameter can hold any type."

A second example shows an important use of unbounded wildcards. When you are dealing with multiple generic parameters, it's sometimes necessary to allow one parameter to be any type while establishing a particular type for the other parameter:

```
// generics/UnboundedWildcards2.java
import java.util.*;

public class UnboundedWildcards2 {
  static Map map1;
  static Map<?,?> map2;
  static Map<String,?> map3;
  static void assign1(Map map) { map1 = map; }
  static void assign2(Map<?,?> map) { map2 = map; }
  static void assign3(Map<String,?> map) { map3 = map; }
  public static void main(String[] args) {
    assign1(new HashMap());
    assign2(new HashMap());
    //- assign3(new HashMap());
    // warning: [unchecked] unchecked method invocation:
    // method assign3 in class UnboundedWildcards2
    // is applied to given types
```

```
    //     assign3(new HashMap());
    //             ^
    //   required: Map<String,?>
    //   found: HashMap
    // warning: [unchecked] unchecked conversion
    //     assign3(new HashMap());
    //             ^
    //   required: Map<String,?>
    //   found:    HashMap
    // 2 warnings
    assign1(new HashMap<>());
    assign2(new HashMap<>());
    assign3(new HashMap<>());
  }
}
```

But again, when you have all unbounded wildcards, as seen in `Map<?,?>`, the compiler doesn't seem to distinguish it from a raw `Map`. In addition, `UnboundedWildcards1.java` shows that the compiler treats `List<?>` and `List<? extends Object>` differently.

What's confusing is that the compiler doesn't always care about differences between, for example, `List` and `List<?>`, so they can seem like the same thing. Indeed, because a generic argument erases to its first bound, `List<?>` would seem equivalent to `List<Object>`, and `List` is effectively `List<Object>` as well—except neither of those statements is exactly true. `List` actually means "a raw `List` that holds any `Object` type," whereas `List<?>` means "a non-raw `List` of *some specific type*, but we just don't know what that type is."

When does the compiler actually care about differences between raw types and types involving unbounded wildcards? The following example uses the previously defined `Holder<T>` class. It contains methods that take `Holder` as an argument, but in various forms: as a raw type, with a specific type parameter, and with an unbounded wildcard parameter:

```
// generics/Wildcards.java
// Exploring the meaning of wildcards

public class Wildcards {
   // Raw argument:
  static void rawArgs(Holder holder, Object arg) {
    //- holder.set(arg);
    // warning: [unchecked] unchecked call to set(T)
    // as a member of the raw type Holder
    //     holder.set(arg);
    //                ^
    //   where T is a type-variable:
    //     T extends Object declared in class Holder
    // 1 warning

    // Can't do this; don't have any 'T':
    // T t = holder.get();

    // OK, but type information is lost:
    Object obj = holder.get();
  }
  // Like rawArgs(), but errors instead of warnings:
  static void
  unboundedArg(Holder<?> holder, Object arg) {
    //- holder.set(arg);
    // error: method set in class Holder<T>
    // cannot be applied to given types;
    //     holder.set(arg);
    //            ^
    //   required: CAP#1
    //   found: Object
    //   reason: argument mismatch;
    //     Object cannot be converted to CAP#1
    //   where T is a type-variable:
    //     T extends Object declared in class Holder
    //   where CAP#1 is a fresh type-variable:
    //     CAP#1 extends Object from capture of ?
    // 1 error

    // Can't do this; don't have any 'T':
    // T t = holder.get();
```

```
  // OK, but type information is lost:
  Object obj = holder.get();
}
static <T> T exact1(Holder<T> holder) {
  return holder.get();
}
static <T> T exact2(Holder<T> holder, T arg) {
  holder.set(arg);
  return holder.get();
}
static <T>
T wildSubtype(Holder<? extends T> holder, T arg) {
  //- holder.set(arg);
  // error: method set in class Holder<T#2>
  // cannot be applied to given types;
  //     holder.set(arg);
  //            ^
  //   required: CAP#1
  //   found: T#1
  //   reason: argument mismatch;
  //     T#1 cannot be converted to CAP#1
  //   where T#1,T#2 are type-variables:
  //     T#1 extends Object declared in method
  //     <T#1>wildSubtype(Holder<? extends T#1>,T#1)
  //     T#2 extends Object declared in class Holder
  //   where CAP#1 is a fresh type-variable:
  //     CAP#1 extends T#1 from
  //       capture of ? extends T#1
  // 1 error

  return holder.get();
}
static <T>
void wildSupertype(Holder<? super T> holder, T arg) {
  holder.set(arg);
  //- T t = holder.get();
  // error: incompatible types:
  // CAP#1 cannot be converted to T
  //     T t = holder.get();
  //                  ^
  //   where T is a type-variable:
  //     T extends Object declared in method
```

```
    //       <T>wildSupertype(Holder<? super T>,T)
    //   where CAP#1 is a fresh type-variable:
    //     CAP#1 extends Object super:
    //       T from capture of ? super T
    // 1 error

    // OK, but type information is lost:
    Object obj = holder.get();
  }
  public static void main(String[] args) {
    Holder raw = new Holder<>();
    // Or:
    raw = new Holder();
    Holder<Long> qualified = new Holder<>();
    Holder<?> unbounded = new Holder<>();
    Holder<? extends Long> bounded = new Holder<>();
    Long lng = 1L;

    rawArgs(raw, lng);
    rawArgs(qualified, lng);
    rawArgs(unbounded, lng);
    rawArgs(bounded, lng);

    unboundedArg(raw, lng);
    unboundedArg(qualified, lng);
    unboundedArg(unbounded, lng);
    unboundedArg(bounded, lng);

    //- Object r1 = exact1(raw);
    // warning: [unchecked] unchecked method invocation:
    // method exact1 in class Wildcards is applied
    // to given types
    //      Object r1 = exact1(raw);
    //                          ^
    //   required: Holder<T>
    //   found: Holder
    //   where T is a type-variable:
    //     T extends Object declared in
    //     method <T>exact1(Holder<T>)
    // warning: [unchecked] unchecked conversion
    //      Object r1 = exact1(raw);
    //                          ^
```

```
//   required: Holder<T>
//   found:    Holder
//   where T is a type-variable:
//     T extends Object declared in
//     method <T>exact1(Holder<T>)
// 2 warnings

Long r2 = exact1(qualified);
Object r3 = exact1(unbounded); // Must return Object
Long r4 = exact1(bounded);

//- Long r5 = exact2(raw, lng);
// warning: [unchecked] unchecked method invocation:
// method exact2 in class Wildcards is
// applied to given types
//     Long r5 = exact2(raw, lng);
//                      ^
//   required: Holder<T>,T
//   found: Holder,Long
//   where T is a type-variable:
//     T extends Object declared in
//       method <T>exact2(Holder<T>,T)
// warning: [unchecked] unchecked conversion
//     Long r5 = exact2(raw, lng);
//                       ^
//   required: Holder<T>
//   found:    Holder
//   where T is a type-variable:
//     T extends Object declared in
//       method <T>exact2(Holder<T>,T)
// 2 warnings

Long r6 = exact2(qualified, lng);

//- Long r7 = exact2(unbounded, lng);
// error: method exact2 in class Wildcards
// cannot be applied to given types;
//     Long r7 = exact2(unbounded, lng);
//               ^
//   required: Holder<T>,T
//   found: Holder<CAP#1>,Long
//   reason: inference variable T has
```

```
//     incompatible bounds
//     equality constraints: CAP#1
//     lower bounds: Long
//   where T is a type-variable:
//     T extends Object declared in
//       method <T>exact2(Holder<T>,T)
//   where CAP#1 is a fresh type-variable:
//     CAP#1 extends Object from capture of ?
// 1 error

//- Long r8 = exact2(bounded, lng);
// error: method exact2 in class Wildcards
// cannot be applied to given types;
//      Long r8 = exact2(bounded, lng);
//                ^
//   required: Holder<T>,T
//   found: Holder<CAP#1>,Long
//   reason: inference variable T
//     has incompatible bounds
//     equality constraints: CAP#1
//     lower bounds: Long
//   where T is a type-variable:
//     T extends Object declared in
//       method <T>exact2(Holder<T>,T)
//   where CAP#1 is a fresh type-variable:
//     CAP#1 extends Long from
//       capture of ? extends Long
// 1 error

//- Long r9 = wildSubtype(raw, lng);
// warning: [unchecked] unchecked method invocation:
// method wildSubtype in class Wildcards
// is applied to given types
//     Long r9 = wildSubtype(raw, lng);
//                           ^
//   required: Holder<? extends T>,T
//   found: Holder,Long
//   where T is a type-variable:
//     T extends Object declared in
//     method <T>wildSubtype(Holder<? extends T>,T)
// warning: [unchecked] unchecked conversion
//     Long r9 = wildSubtype(raw, lng);
```

```
//                            ^
//   required: Holder<? extends T>
//   found:    Holder
//   where T is a type-variable:
//     T extends Object declared in
//     method <T>wildSubtype(Holder<? extends T>,T)
// 2 warnings

Long r10 = wildSubtype(qualified, lng);
// OK, but can only return Object:
Object r11 = wildSubtype(unbounded, lng);
Long r12 = wildSubtype(bounded, lng);

//- wildSupertype(raw, lng);
// warning: [unchecked] unchecked method invocation:
//   method wildSupertype in class Wildcards
//   is applied to given types
//     wildSupertype(raw, lng);
//                   ^
//   required: Holder<? super T>,T
//   found: Holder,Long
//   where T is a type-variable:
//     T extends Object declared in
//     method <T>wildSupertype(Holder<? super T>,T)
// warning: [unchecked] unchecked conversion
//     wildSupertype(raw, lng);
//                   ^
//   required: Holder<? super T>
//   found:    Holder
//   where T is a type-variable:
//     T extends Object declared in
//     method <T>wildSupertype(Holder<? super T>,T)
// 2 warnings

wildSupertype(qualified, lng);

//- wildSupertype(unbounded, lng);
// error: method wildSupertype in class Wildcards
// cannot be applied to given types;
//     wildSupertype(unbounded, lng);
//     ^
//   required: Holder<? super T>,T
```

```
    //   found: Holder<CAP#1>,Long
    //   reason: cannot infer type-variable(s) T
    //     (argument mismatch; Holder<CAP#1>
    //     cannot be converted to Holder<? super T>)
    //   where T is a type-variable:
    //     T extends Object declared in
    //     method <T>wildSupertype(Holder<? super T>,T)
    //   where CAP#1 is a fresh type-variable:
    //     CAP#1 extends Object from capture of ?
    // 1 error

    //- wildSupertype(bounded, lng);
    // error: method wildSupertype in class Wildcards
    // cannot be applied to given types;
    //     wildSupertype(bounded, lng);
    //     ^
    //   required: Holder<? super T>,T
    //   found: Holder<CAP#1>,Long
    //   reason: cannot infer type-variable(s) T
    //     (argument mismatch; Holder<CAP#1>
    //     cannot be converted to Holder<? super T>)
    //   where T is a type-variable:
    //     T extends Object declared in
    //     method <T>wildSupertype(Holder<? super T>,T)
    //   where CAP#1 is a fresh type-variable:
    //     CAP#1 extends Long from capture of
    //     ? extends Long
    // 1 error
  }
}
```

In `rawArgs()`, the compiler knows that `Holder` is a generic type, so even though it is expressed as a raw type here, the compiler knows that passing an `Object` to `set()` is unsafe. It's a raw type so you can pass an object of any type into `set()` and that object is upcast to `Object`. So anytime you have a raw type, you give up compile-time checking. The call to `get()` shows the same issue: There's no `T`, so the result can only be an `Object`.

It's easy to start thinking that a raw `Holder` and a `Holder<?>` are roughly the same thing. But `unboundedArg()` emphasizes that they are different—it discovers the same kind of problems, but reports them as errors rather than warnings, because

the raw `Holder` will hold a combination of any types, whereas a `Holder<?>` holds a homogeneous collection of *some specific type*, and thus you can't just pass in an `Object`.

In `exact1()` and `exact2()`, you see the exact generic parameters used—no wildcards. You'll see that `exact2()` has different limitations than `exact1()`, because of the extra argument.

In `wildSubtype()`, the constraints on the type of `Holder` are relaxed to include a `Holder` of anything that `extends T`. Again, this means `T` could be `Fruit`, while `holder` could legitimately be a `Holder<Apple>`. To prevent putting an `Orange` in a `Holder<Apple>`, the call to `set()` (or any method that takes an argument of the type parameter) is disallowed. However, you still know that anything that comes out of a `Holder<? extends Fruit>` will at least be `Fruit`, so `get()` (or any method that produces a return value of the type parameter) is allowed.

Supertype wildcards are shown in `wildSupertype()`, which shows the opposite behavior of `wildSubtype()`: `holder` can be a collection that holds any type that's a base class of `T`. Thus, `set()` can accept a `T` because anything that works with a base type will polymorphically work with a derived type (thus a `T`). However, trying to call `get()` is not helpful, because the type held by `holder` can be any supertype at all, so the only safe one is `Object`.

This example also shows the limitations on what you can and can't do with an unbounded parameter in `unbounded()`: You can't `get()` or `set()` a `T` because you don't have a `T`.

In `main()` you see which of these methods can accept which types of arguments without errors and warnings. For migration compatibility, `rawArgs()` will take all the different variations of `Holder` without producing warnings. The `unboundedArg()` method is equally accepting of all types, although, as previously noted, it handles them differently inside the body of the method.

If you pass a raw `Holder` reference into a method that takes an "exact" generic type (no wildcards), you get a warning because the exact argument is expecting information that doesn't exist in the raw type. And if you pass an unbounded reference to `exact1()`, there's no type information to establish the return type.

`exact2()` has the most constraints because it wants exactly a `Holder<T>` and an argument of type `T`. It generates errors or warnings unless you give it the exact

arguments. Sometimes this is OK, but if it's overconstraining, you can use wildcards, depending on whether you get typed return values from your generic argument (as seen in `wildSubtype()`) or you pass typed arguments to your generic argument (as seen in `wildSupertype()`).

The benefit of using exact types instead of wildcard types is that you can do more with the generic parameters. But wildcards accepts a broader range of parameterized types as arguments. You must decide which trade-off is more appropriate for your needs on a case-by-case basis.

Capture Conversion

One situation in particular *requires* `<?>` rather than a raw type. If you pass a raw type to a method that uses `<?>`, it's possible for the compiler to infer the actual type parameter, so the method can turn around and call another method that uses the exact type. The following example demonstrates this technique, called *capture conversion* because the unspecified wildcard type is captured and converted to an exact type. Here, the comments about warnings only take effect when the `@SuppressWarnings` annotation is removed:

```
// generics/CaptureConversion.java

public class CaptureConversion {
  static <T> void f1(Holder<T> holder) {
    T t = holder.get();
    System.out.println(t.getClass().getSimpleName());
  }
  static void f2(Holder<?> holder) {
    f1(holder); // Call with captured type
  }
  @SuppressWarnings("unchecked")
  public static void main(String[] args) {
    Holder raw = new Holder<>(1);

    f1(raw);
    // warning: [unchecked] unchecked method invocation:
    // method f1 in class CaptureConversion
    // is applied to given types
    //     f1(raw);
```

```
    //       ^
    //   required: Holder<T>
    //   found: Holder
    //   where T is a type-variable:
    //     T extends Object declared in
    //     method <T>f1(Holder<T>)
    // warning: [unchecked] unchecked conversion
    //     f1(raw);
    //        ^
    //   required: Holder<T>
    //   found:    Holder
    //   where T is a type-variable:
    //     T extends Object declared in
    //     method <T>f1(Holder<T>)
    // 2 warnings

    f2(raw); // No warnings
    Holder rawBasic = new Holder();

    rawBasic.set(new Object());
    // warning: [unchecked] unchecked call to set(T)
    // as a member of the raw type Holder
    //     rawBasic.set(new Object());
    //                  ^
    //   where T is a type-variable:
    //     T extends Object declared in class Holder
    // 1 warning

    f2(rawBasic); // No warnings
    // Upcast to Holder<?>, still figures it out:
    Holder<?> wildcarded = new Holder<>(1.0);
    f2(wildcarded);
  }
}
/* Output:
Integer
Integer
Object
Double
*/
```

The type parameters in `f1()` are all exact, without wildcards or bounds. In `f2()`,

the `Holder` parameter is an unbounded wildcard, so it would seem to be effectively unknown. However, within `f2()`, `f1()` is called and `f1()` requires a known parameter. What's happening is that the parameter type is captured in the process of calling `f2()`, and used in the call to `f1()`.

You might wonder if this technique could be used for writing, but that would require you to pass a specific type along with the `Holder<?>`. Capture conversion only works in situations where, within the method, you must work with the exact type. Notice you can't return `T` from `f2()`, because `T` is unknown for `f2()`. Capture conversion is interesting, but limited.

Issues

This section addresses an assorted set of issues that appear when you use Java generics.

No Primitives as Type Parameters

As mentioned earlier in this chapter, one of the limitations in Java generics is you cannot use primitives as type parameters. So you cannot, for example, create an `ArrayList<int>`.

The solution is to use the primitive wrapper classes in conjunction with autoboxing. If you create an `ArrayList<Integer>` and use primitive `int`s with this collection, you'll discover that autoboxing does the conversion to and from `Integer` automatically—so it's almost as if you have an `ArrayList<int>`:

```
// generics/ListOfInt.java
// Autoboxing compensates for the inability
// to use primitives in generics
import java.util.*;
import java.util.stream.*;

public class ListOfInt {
  public static void main(String[] args) {
    List<Integer> li = IntStream.range(38, 48)
      .boxed() // Converts ints to Integers
      .collect(Collectors.toList());
    System.out.println(li);
  }
}
/* Output:
[38, 39, 40, 41, 42, 43, 44, 45, 46, 47]
*/
```

In general this solution works fine—you're able to successfully store and retrieve `int`s, and autoboxing hides the conversions. However, if performance is a problem, you can use a specialized version of the collections adapted for primitive types; one open-source version of this is `org.apache.commons.collections.primitives`.

Here's another approach, which creates a `Set` of `Byte`s:

```
// generics/ByteSet.java
import java.util.*;

public class ByteSet {
  Byte[] possibles = { 1,2,3,4,5,6,7,8,9 };
  Set<Byte> mySet =
    new HashSet<>(Arrays.asList(possibles));
  // But you can't do this:
  // Set<Byte> mySet2 = new HashSet<>(
  //   Arrays.<Byte>asList(1,2,3,4,5,6,7,8,9));
}
```

Autoboxing solves some problems, but not all.

In the following example, the `FillArray interface` contains generic methods that use `Suppliers` to fill arrays with objects (making the *class* generic wouldn't work here because the method is `static`). The `Supplier` implementations come from the

Arrays chapter, and in `main()` you see `FillArray.fill()` used to fill arrays with objects:

```
// generics/PrimitiveGenericTest.java
import onjava.*;
import java.util.*;
import java.util.function.*;

// Fill an array using a generator:
interface FillArray {
  static <T> T[] fill(T[] a, Supplier<T> gen) {
    Arrays.setAll(a, n -> gen.get());
    return a;
  }
  static int[] fill(int[] a, IntSupplier gen) {
    Arrays.setAll(a, n -> gen.getAsInt());
    return a;
  }
  static long[] fill(long[] a, LongSupplier gen) {
    Arrays.setAll(a, n -> gen.getAsLong());
    return a;
  }
  static double[] fill(double[] a, DoubleSupplier gen) {
    Arrays.setAll(a, n -> gen.getAsDouble());
    return a;
  }
}

public class PrimitiveGenericTest {
  public static void main(String[] args) {
    String[] strings = FillArray.fill(
      new String[5], new Rand.String(9));
    System.out.println(Arrays.toString(strings));
    int[] integers = FillArray.fill(
      new int[9], new Rand.Pint());
    System.out.println(Arrays.toString(integers));
  }
}
/* Output:
[btpenpccu, xszgvgmei, nneeloztd, vewcippcy, gpoalkljl]
[635, 8737, 3941, 4720, 6177, 8479, 6656, 3768, 4948]
*/
```

Autoboxing doesn't apply to arrays, so we have to create overloaded versions of `FillArray.fill()`, or create generators that produced the Wrapped output.

`FillArray` is only slightly more useful than `java.util.Arrays.setAll()` because it returns the filled array.

Implementing Parameterized Interfaces

A class cannot implement two variants of the same generic interface. Because of erasure, these are both the same interface. Here's a situation where this clash occurs:

```
// generics/MultipleInterfaceVariants.java
// {WillNotCompile}
package generics;

interface Payable<T> {}

class Employee implements Payable<Employee> {}

class Hourly extends Employee
implements Payable<Hourly> {}
```

`Hourly` won't compile because erasure reduces `Payable<Employee>` and `Payable<Hourly>` to the same class, `Payable`, and the above code would mean that you'd be implementing the same interface twice. If you remove the generic parameters from both uses of `Payable`—as the compiler does during erasure—the code compiles.

This issue becomes annoying when working with some of the more fundamental Java interfaces, such as `Comparable<T>`, as you'll see a little later in this section.

Casting and Warnings

Using a cast or `instanceof` with a generic type parameter doesn't have any effect. The following collection stores values internally as `Objects` and casts them back to `T` when you fetch them:

```
// generics/GenericCast.java
import java.util.*;
import java.util.stream.*;

class FixedSizeStack<T> {
  private final int size;
  private Object[] storage;
  private int index = 0;
  FixedSizeStack(int size) {
    this.size = size;
    storage = new Object[size];
  }
  public void push(T item) {
    if(index < size)
      storage[index++] = item;
  }
  @SuppressWarnings("unchecked")
  public T pop() {
    return index == 0 ? null : (T)storage[--index];
  }
  @SuppressWarnings("unchecked")
  Stream<T> stream() {
    return (Stream<T>)Arrays.stream(storage);
  }
}

public class GenericCast {
  static String[] letters =
    "ABCDEFGHIJKLMNOPQRS".split("");
  public static void main(String[] args) {
    FixedSizeStack<String> strings =
      new FixedSizeStack<>(letters.length);
    Arrays.stream("ABCDEFGHIJKLMNOPQRS".split(""))
      .forEach(strings::push);
    System.out.println(strings.pop());
    strings.stream()
      .map(s -> s + " ")
      .forEach(System.out::print);
  }
}
/* Output:
S
```

```
A B C D E F G H I J K L M N O P Q R S
*/
```

Without the `@SuppressWarnings` annotations, the compiler will produce an "unchecked cast" warning for `pop()` and `stream()`. Because of erasure, it can't know whether the cast is safe. `T` is erased to its first bound, `Object` by default, so `pop()` is actually just casting an `Object` to an `Object`.

There are times when generics do not eliminate the need to cast, and this generates a warning by the compiler which is inappropriate. For example:

```
// generics/NeedCasting.java
import java.io.*;
import java.util.*;

public class NeedCasting {
  @SuppressWarnings("unchecked")
  public void f(String[] args) throws Exception {
    ObjectInputStream in = new ObjectInputStream(
      new FileInputStream(args[0]));
    List<Widget> shapes = (List<Widget>)in.readObject();
  }
}
```

As you'll learn in the Appendix: Object Serialization, `readObject()` cannot know what it is reading, so it returns an `Object` that must be cast. But when you comment out the `@SuppressWarnings` annotation and compile the program, you get a warning:

```
NeedCasting.java uses unchecked or unsafe operations.
Recompile with -Xlint:unchecked for details.
```

And if you follow the instructions and recompile with `-Xlint:unchecked`:

```
NeedCasting.java:10: warning: [unchecked] unchecked cast
    List<Widget> shapes = (List<Widget>)in.readObject();
                                        ^
  required: List<Widget>
  found:    Object
1 warning
```

You're forced to cast, and yet you're told not to. To solve the problem, you must use a form of cast introduced in Java 5, the cast via a generic class:

```
// generics/ClassCasting.java
import java.io.*;
import java.util.*;

public class ClassCasting {
  @SuppressWarnings("unchecked")
  public void f(String[] args) throws Exception {
    ObjectInputStream in = new ObjectInputStream(
      new FileInputStream(args[0]));
      // Won't Compile:
//    List<Widget> lw1 =
//    List<>.class.cast(in.readObject());
    List<Widget> lw2 = List.class.cast(in.readObject());
  }
}
```

However, you can't cast to the actual type (`List<Widget>`). That is, you can't say:

```
List<Widget>.class.cast(in.readObject())
```

and even if you add another cast like this:

```
(List<Widget>)List.class.cast(in.readObject())
```

you'll still get a warning.

Overloading

This won't compile, even though it seems reasonable:

```
// generics/UseList.java
// {WillNotCompile}
import java.util.*;

public class UseList<W, T> {
  void f(List<T> v) {}
  void f(List<W> v) {}
}
```

Overloading the method produces the identical type signature because of erasure.

Instead, you must provide distinct method names when the erased arguments do not produce a unique argument list:

```
// generics/UseList2.java
import java.util.*;

public class UseList2<W, T> {
  void f1(List<T> v) {}
  void f2(List<W> v) {}
}
```

Fortunately, this kind of problem is detected by the compiler.

Base Class Hijacks an Interface

Suppose you have a `Pet` class that is `Comparable` to other `Pet` objects:

```
// generics/ComparablePet.java

public class ComparablePet
implements Comparable<ComparablePet> {
  @Override
  public int compareTo(ComparablePet arg) {
    return 0;
  }
}
```

It makes sense to try to narrow the comparison type for a subclass of `ComparablePet`. For example, a `Cat` should only be `Comparable` with other `Cats`:

```
// generics/HijackedInterface.java
// {WillNotCompile}

class Cat
  extends ComparablePet implements Comparable<Cat>{
  // error: Comparable cannot be inherited with
  // different arguments: <Cat> and <ComparablePet>
  // class Cat
  // ^
  // 1 error

  public int compareTo(Cat arg) { return 0; }
}
```

Unfortunately, this won't work. Once the `ComparablePet` argument is established for `Comparable`, no other implementing class can ever be compared to anything but a `ComparablePet`:

```
// generics/RestrictedComparablePets.java

class Hamster extends ComparablePet
implements Comparable<ComparablePet> {
  @Override
  public int compareTo(ComparablePet arg) {
    return 0;
  }
}

// Or just:

class Gecko extends ComparablePet {
  public int compareTo(ComparablePet arg) {
    return 0;
  }
}
```

`Hamster` shows it is possible to reimplement the same interface that is in `ComparablePet`, as long as it is exactly the same, including the parameter types. However, this is the same as just overriding the methods in the base class, as seen in `Gecko`.

Self-Bounded Types

There's one rather mind-bending idiom that appears periodically in Java generics. Here's what it looks like:

```
class SelfBounded<T extends SelfBounded<T>> { // ...
```

This has the dizzying effect of two mirrors pointed at each other, a kind of infinite reflection. The class `SelfBounded` takes a generic argument `T`, `T` is constrained by a bound, and that bound is `SelfBounded`, with `T` as an argument.

This is difficult to parse when you first see it, and it emphasizes that the `extends` keyword, when used with bounds, is definitely different than when it is used to create subclasses.

Curiously Recurring Generics

To understand what a self-bounded type means, let's start with a simpler version of the idiom, without the self-bound.

You can't inherit directly from a generic parameter. However, you *can* inherit from a class that uses that generic parameter in its own definition. That is, you can say:

```
// generics/CuriouslyRecurringGeneric.java

class GenericType<T> {}

public class CuriouslyRecurringGeneric
  extends GenericType<CuriouslyRecurringGeneric> {}
```

This could be called *curiously recurring generics* (CRG) after Jim Coplien's *Curiously Recurring Template Pattern* in C++. The "curiously recurring" part refers to the fact that your class appears, rather curiously, in its own base class.

To understand what this means, try saying it aloud: "I'm creating a new class that inherits from a generic type that takes my class name as its parameter." What can the generic base type accomplish when given the derived class name? Well, generics in Java are about arguments and return types, so it can produce a base class that uses the derived type for its arguments and return types. It can also use the derived type for field types, even though those are erased to `Object`. Here's a generic class that expresses this:

```
// generics/BasicHolder.java

public class BasicHolder<T> {
  T element;
  void set(T arg) { element = arg; }
  T get() { return element; }
  void f() {
    System.out.println(
      element.getClass().getSimpleName());
  }
}
```

It's an ordinary generic type with methods that both accept and produce objects of the parameter type, along with a method that operates on the stored field (although it only performs `Object` operations on that field).

We can use `BasicHolder` in a curiously recurring generic:

```
// generics/CRGWithBasicHolder.java

class Subtype extends BasicHolder<Subtype> {}

public class CRGWithBasicHolder {
  public static void main(String[] args) {
    Subtype
      st1 = new Subtype(),
      st2 = new Subtype();
    st1.set(st2);
    Subtype st3 = st1.get();
    st1.f();
  }
}
/* Output:
Subtype
*/
```

Notice something important here: The new class `Subtype` takes arguments and returns values of `Subtype`, not just the base class `BasicHolder`. This is the essence of CRG: *The base class substitutes the derived class for its parameters.* This means the generic base class becomes a kind of template for common functionality for all its derived classes, but this functionality will use the derived type for all of its arguments

and return values. That is, the exact type instead of the base type is used in the resulting class. So in `Subtype`, both the argument to `set()` and the return type of `get()` are exactly `Subtypes`.

Self-Bounding

The `BasicHolder` can use any type as its generic parameter, as seen here:

```
// generics/Unconstrained.java

class Other {}
class BasicOther extends BasicHolder<Other> {}

public class Unconstrained {
  public static void main(String[] args) {
    BasicOther b = new BasicOther();
    BasicOther b2 = new BasicOther();
    b.set(new Other());
    Other other = b.get();
    b.f();
  }
}
/* Output:
Other
*/
```

Self-bounding takes the extra step of *forcing* the generic to be used as its own bound argument. Look at how the resulting class can and can't be used:

```
// generics/SelfBounding.java

class SelfBounded<T extends SelfBounded<T>> {
  T element;
  SelfBounded<T> set(T arg) {
    element = arg;
    return this;
  }
  T get() { return element; }
}

class A extends SelfBounded<A> {}
```

```
class B extends SelfBounded<A> {} // Also OK

class C extends SelfBounded<C> {
  C setAndGet(C arg) { set(arg); return get(); }
}

class D {}
// Can't do this:
// class E extends SelfBounded<D> {}
// Compile error:
//   Type parameter D is not within its bound

// Alas, you can do this, so you cannot force the idiom:
class F extends SelfBounded {}

public class SelfBounding {
  public static void main(String[] args) {
    A a = new A();
    a.set(new A());
    a = a.set(new A()).get();
    a = a.get();
    C c = new C();
    c = c.setAndGet(new C());
  }
}
```

What self-bounding does is require the class in an inheritance relationship:

```
class A extends SelfBounded<A> {}
```

This forces you to pass the class you are defining as a parameter to the base class.

What's the added value in self-bounding the parameter? The type parameter must be the same as the class being defined. As you see in the definition of class `B`, you can also derive from a `SelfBounded` that uses a parameter of another `SelfBounded`, although the predominant use seems to be the one you see for class `A`. The attempt to define `E` shows you cannot use a type parameter that is not a `SelfBounded`.

Unfortunately, `F` compiles without warnings, so the self-bounding idiom is not enforceable. If it's really important, it can require an external tool to ensure that raw types are not used in place of parameterized types.

Notice you can remove the constraint and all the classes will still compile, but `E` will also compile:

```
// generics/NotSelfBounded.java

public class NotSelfBounded<T> {
  T element;
  NotSelfBounded<T> set(T arg) {
    element = arg;
    return this;
  }
  T get() { return element; }
}

class A2 extends NotSelfBounded<A2> {}
class B2 extends NotSelfBounded<A2> {}

class C2 extends NotSelfBounded<C2> {
  C2 setAndGet(C2 arg) { set(arg); return get(); }
}

class D2 {}
// Now this is OK:
class E2 extends NotSelfBounded<D2> {}
```

Clearly, the self-bounding constraint serves only to force the inheritance relationship. If you use self-bounding, you know that the type parameter used by the class is the same basic type as the class that's using that parameter. It forces anyone using that class to follow that form.

It's also possible to use self-bounding for generic methods:

```
// generics/SelfBoundingMethods.java

public class SelfBoundingMethods {
  static <T extends SelfBounded<T>> T f(T arg) {
    return arg.set(arg).get();
  }
  public static void main(String[] args) {
    A a = f(new A());
  }
}
```

This prevents the method from being applied to anything but a self-bounded argument of the form shown.

Argument Covariance

The value of self-bounding types is that they produce *covariant argument types*—method argument types vary to follow the subclasses.

Although self-bounding types also produce return types that are the same as the subclass type, this is not so important because *covariant return types* were introduced in Java 5:

```
// generics/CovariantReturnTypes.java

class Base {}
class Derived extends Base {}

interface OrdinaryGetter {
  Base get();
}

interface DerivedGetter extends OrdinaryGetter {
  // Overridden method return type can vary:
  @Override Derived get();
}

public class CovariantReturnTypes {
  void test(DerivedGetter d) {
    Derived d2 = d.get();
  }
}
```

The `get()` method in `DerivedGetter` overrides `get()` in `OrdinaryGetter` *and* returns a type that is derived from the type returned by `OrdinaryGetter.get()`. Although this is perfectly logical—a derived type method can return a more specific type than the base type method it's overriding—it was illegal in earlier versions of Java.

A self-bounded generic does in fact produce the exact derived type as a return value, as seen here with `get()`:

```
// generics/GenericsAndReturnTypes.java

interface GenericGetter<T extends GenericGetter<T>> {
  T get();
}

interface Getter extends GenericGetter<Getter> {}

public class GenericsAndReturnTypes {
  void test(Getter g) {
    Getter result = g.get();
    GenericGetter gg = g.get(); // Also the base type
  }
}
```

Notice this code would not have compiled unless covariant return types had been added in Java 5.

In non-generic code, however, the *argument* types cannot be made to vary with the subtypes:

```
// generics/OrdinaryArguments.java

class OrdinarySetter {
  void set(Base base) {
    System.out.println("OrdinarySetter.set(Base)");
  }
}

class DerivedSetter extends OrdinarySetter {
  void set(Derived derived) {
    System.out.println("DerivedSetter.set(Derived)");
  }
```

```
}

public class OrdinaryArguments {
  public static void main(String[] args) {
    Base base = new Base();
    Derived derived = new Derived();
    DerivedSetter ds = new DerivedSetter();
    ds.set(derived);
    // Compiles--overloaded, not overridden!:
    ds.set(base);
  }
}
/* Output:
DerivedSetter.set(Derived)
OrdinarySetter.set(Base)
*/
```

Both `set(derived)` and `set(base)` are legal, so `DerivedSetter.set()` is not overriding `OrdinarySetter.set()`, but instead it is *overloading* that method. The output shows there are two methods in `DerivedSetter`, so the base-class version is still available, thus verifying it was overloaded. Note that the use of `@Override` points out the issue via an error message.

However, with self-bounding types, there is only one method in the derived class, and that method takes the derived type as its argument, not the base type:

```
// generics/SelfBoundingAndCovariantArguments.java

interface
SelfBoundSetter<T extends SelfBoundSetter<T>> {
  void set(T arg);
}

interface Setter extends SelfBoundSetter<Setter> {}

public class SelfBoundingAndCovariantArguments {
  void
  testA(Setter s1, Setter s2, SelfBoundSetter sbs) {
    s1.set(s2);
    //- s1.set(sbs);
    // error: method set in interface SelfBoundSetter<T>
    // cannot be applied to given types;
```

```
    //     s1.set(sbs);
    //       ^
    //   required: Setter
    //   found: SelfBoundSetter
    //   reason: argument mismatch;
    // SelfBoundSetter cannot be converted to Setter
    //   where T is a type-variable:
    //     T extends SelfBoundSetter<T> declared in
    //     interface SelfBoundSetter
    // 1 error
  }
}
```

The compiler doesn't recognize the attempt to pass in the base type as an argument to `set()`, because there is no method with that signature. The argument has, in effect, been overridden.

Without self-bounding, the ordinary inheritance mechanism steps in and you get overloading, just as with the non-generic case:

```
// generics/PlainGenericInheritance.java

class GenericSetter<T> { // Not self-bounded
  void set(T arg) {
    System.out.println("GenericSetter.set(Base)");
  }
}

class DerivedGS extends GenericSetter<Base> {
  void set(Derived derived) {
    System.out.println("DerivedGS.set(Derived)");
  }
}

public class PlainGenericInheritance {
  public static void main(String[] args) {
    Base base = new Base();
    Derived derived = new Derived();
    DerivedGS dgs = new DerivedGS();
    dgs.set(derived);
    dgs.set(base); // Overloaded, not overridden!
  }
```

```
}
/* Output:
DerivedGS.set(Derived)
GenericSetter.set(Base)
*/
```

This code mimics `OrdinaryArguments.java`; in that example, `DerivedSetter` inherits from `OrdinarySetter` which contains a `set(Base)`. Here, `DerivedGS` inherits from `GenericSetter<Base>` which also contains a `set(Base)`, created by the generic. And just like `OrdinaryArguments.java`, the output shows that `DerivedGS` contains two overloaded versions of `set()`. Without self-bounding, you overload on argument types. If you use self-bounding, you only end up with one version of a method, which takes the exact argument type.

Dynamic Type Safety

Because you can pass generic collections to pre-Java 5 code, there's still the possibility that old-style code can corrupt your collections. Java 5 added a set of utilities in `java.util.Collections` to solve the type-checking problem in this situation: the `static` methods `checkedCollection()`, `checkedList()`, `checkedMap()`, `checkedSet()`, `checkedSortedMap()` and `checkedSortedSet()`. Each of these takes the collection to dynamically check as the first argument and the type to enforce as the second argument.

A checked collection will throw a `ClassCastException` when you try to *insert* an improper object, as opposed to a pre-generic (raw) collection which would inform you there was a problem when you pulled the object *out*. In the latter case, you know there's a problem but you don't know who the culprit is, but with checked collections you find out who tried to insert the bad object.

Let's look at the problem of "putting a cat in a list of dogs" using a checked collection. Here, `oldStyleMethod()` represents legacy code because it takes a raw `List`, and the `@SuppressWarnings("unchecked")` annotation is necessary to silence the resulting warning:

```
// generics/CheckedList.java
// Using Collection.checkedList()
import reflection.pets.*;
import java.util.*;

public class CheckedList {
  @SuppressWarnings("unchecked")
  static void oldStyleMethod(List probablyDogs) {
    probablyDogs.add(new Cat());
  }
  public static void main(String[] args) {
    List<Dog> dogs1 = new ArrayList<>();
    oldStyleMethod(dogs1); // Quietly accepts a Cat
    List<Dog> dogs2 = Collections.checkedList(
      new ArrayList<>(), Dog.class);
    try {
      oldStyleMethod(dogs2); // Throws an exception
    } catch(Exception e) {
      System.out.println("Expected: " + e);
    }
    // Derived types work fine:
    List<Pet> pets = Collections.checkedList(
      new ArrayList<>(), Pet.class);
    pets.add(new Dog());
    pets.add(new Cat());
  }
}
/* Output:
Expected: java.lang.ClassCastException: Attempt to
insert class reflection.pets.Cat element into collection
with element type class reflection.pets.Dog
*/
```

When you run the program you'll see that the insertion of a `Cat` goes unchallenged by `dogs1`, but `dogs2` immediately throws an exception upon the insertion of an incorrect type. You can also see it's fine to put derived-type objects into a checked collection that is checking for the base type.

Exceptions

Because of erasure, a `catch` clause cannot catch an exception of a generic type, because the exact type of the exception must be known at both compile time and run time. Also, a generic class can't directly or indirectly inherit from `Throwable` (this further prevents you from trying to define generic exceptions that can't be caught).

However, type parameters can be used in the `throws` clause of a method declaration. This means you can write generic code that varies with the type of a checked exception:

```
// generics/ThrowGenericException.java
import java.util.*;

interface Processor<T, E extends Exception> {
  void process(List<T> resultCollector) throws E;
}

class ProcessRunner<T, E extends Exception>
extends ArrayList<Processor<T, E>> {
  List<T> processAll() throws E {
    List<T> resultCollector = new ArrayList<>();
    for(Processor<T, E> processor : this)
      processor.process(resultCollector);
    return resultCollector;
  }
}

class Failure1 extends Exception {}

class Processor1
implements Processor<String, Failure1> {
  static int count = 3;
  @Override
  public void process(List<String> resultCollector)
  throws Failure1 {
    if(count-- > 1)
      resultCollector.add("Hep!");
    else
      resultCollector.add("Ho!");
    if(count < 0)
```

```
      throw new Failure1();
  }
}

class Failure2 extends Exception {}

class Processor2
implements Processor<Integer, Failure2> {
  static int count = 2;
  @Override
  public void process(List<Integer> resultCollector)
  throws Failure2 {
    if(count-- == 0)
      resultCollector.add(47);
    else {
      resultCollector.add(11);
    }
    if(count < 0)
       throw new Failure2();
  }
}

public class ThrowGenericException {
  public static void main(String[] args) {
    ProcessRunner<String, Failure1> runner =
      new ProcessRunner<>();
    for(int i = 0; i < 3; i++)
      runner.add(new Processor1());
    try {
      System.out.println(runner.processAll());
    } catch(Failure1 e) {
      System.out.println(e);
    }

    ProcessRunner<Integer, Failure2> runner2 =
      new ProcessRunner<>();
    for(int i = 0; i < 3; i++)
      runner2.add(new Processor2());
    try {
      System.out.println(runner2.processAll());
    } catch(Failure2 e) {
      System.out.println(e);
```

```
    }
  }
}
/* Output:
[Hep!, Hep!, Ho!]
Failure2
*/
```

A `Processor` performs a `process()` and might throw an exception of type `E`. The result of the `process()` is stored in the `List<T> resultCollector` (this is called a *collecting parameter*). A `ProcessRunner` has a `processAll()` method that executes every `Process` object it holds, and returns the `resultCollector`.

Unless you can parameterize the thrown exceptions, you can't write this code generically because of the checked exceptions.

Mixins

The term *mixin* has acquired numerous meanings, but the fundamental concept is that of mixing capabilities from multiple classes to produce a resulting class that represents all the types of the mixins. This is often something you do at the last minute, which makes it convenient to easily assemble classes.

One value of mixins is they consistently apply characteristics and behaviors across multiple classes. As a bonus, if you change something in a mixin class, those changes are applied across all the classes where the mixin is used. Because of this, mixins have part of the flavor of *aspect-oriented programming* (AOP), and aspects are often suggested to solve the mixin problem.

Mixins in C++

One of the strongest arguments made for multiple inheritance in C++ is for mixins. A more elegant approach to mixins uses parameterized types, whereby a mixin is a class that inherits from its type parameter. In C++, you can easily create mixins because C++ remembers the type of its template parameters.

Here's a C++ example with two mixin types: one that mixes in the property of having a time stamp, and another that mixes in a serial number for each object instance:

```
// generics/Mixins.cpp
#include <string>
#include <ctime>
#include <iostream>
using namespace std;

template<class T> class TimeStamped : public T {
  long timeStamp;
public:
  TimeStamped() { timeStamp = time(0); }
  long getStamp() { return timeStamp; }
};

template<class T> class SerialNumbered : public T {
  long serialNumber;
  static long counter;
public:
  SerialNumbered() { serialNumber = counter++; }
  long getSerialNumber() { return serialNumber; }
};

// Define and initialize the static storage:
template<class T> long SerialNumbered<T>::counter = 1;

class Basic {
  string value;
public:
  void set(string val) { value = val; }
  string get() { return value; }
};

int main() {
  TimeStamped<SerialNumbered<Basic>> mixin1, mixin2;
  mixin1.set("test string 1");
  mixin2.set("test string 2");
  cout << mixin1.get() << " " << mixin1.getStamp() <<
    " " << mixin1.getSerialNumber() << endl;
  cout << mixin2.get() << " " << mixin2.getStamp() <<
    " " << mixin2.getSerialNumber() << endl;
}
/* Output:
test string 1 1452987605 1
```

```
test string 2 1452987605 2
*/
```

In `main()`, the resulting type of `mixin1` and `mixin2` has all the methods of the mixed-in types. You can think of a mixin as a function that maps existing classes to new subclasses. Notice how trivial it is to create a mixin using this technique; basically, you just say, "Here's what I want," and it happens:

```
TimeStamped<SerialNumbered<Basic>> mixin1, mixin2;
```

Very unfortunately, Java generics don't permit this. Erasure forgets the base-class type, so:

> A generic class cannot inherit directly from a generic parameter.

This highlights one of the big problems I have with any number of Java language design decisions (and the marketing that went along with those features): There's a lot of promise, but when you actually try to do something interesting, you discover you can't.

Mixing with Interfaces

A commonly suggested solution is to use interfaces to produce the effect of mixins, like this:

```
// generics/Mixins.java
import java.util.*;

interface TimeStamped { long getStamp(); }

class TimeStampedImp implements TimeStamped {
  private final long timeStamp;
  TimeStampedImp() {
    timeStamp = new Date().getTime();
  }
  @Override
  public long getStamp() { return timeStamp; }
}
```

```
interface SerialNumbered { long getSerialNumber(); }

class SerialNumberedImp implements SerialNumbered {
  private static long counter = 1;
  private final long serialNumber = counter++;
  @Override
  public long getSerialNumber() { return serialNumber; }
}

interface Basic {
  void set(String val);
  String get();
}

class BasicImp implements Basic {
  private String value;
  @Override
  public void set(String val) { value = val; }
  @Override
  public String get() { return value; }
}

class Mixin extends BasicImp
implements TimeStamped, SerialNumbered {
  private TimeStamped timeStamp = new TimeStampedImp();
  private SerialNumbered serialNumber =
    new SerialNumberedImp();
  @Override public long getStamp() {
    return timeStamp.getStamp();
  }
  @Override public long getSerialNumber() {
    return serialNumber.getSerialNumber();
  }
}

public class Mixins {
  public static void main(String[] args) {
    Mixin mixin1 = new Mixin(), mixin2 = new Mixin();
    mixin1.set("test string 1");
    mixin2.set("test string 2");
    System.out.println(mixin1.get() + " " +
```

```
      mixin1.getStamp() +  " " +
      mixin1.getSerialNumber());
    System.out.println(mixin2.get() + " " +
      mixin2.getStamp() +  " " +
      mixin2.getSerialNumber());
  }
}
/* Output:
test string 1 1611503367257 1
test string 2 1611503367258 2
*/
```

The `Mixin` class is basically using *delegation*, so each mixed-in type requires a field in `Mixin`, and you must write all the necessary methods in `Mixin` to forward calls to the appropriate object. This example uses trivial classes, but with a more complex mixin the code grows rapidly.[146]

Using the Decorator Pattern

When you look at the way it is used, the concept of a mixin seems closely related to the *Decorator* design pattern. Decorators are often used when, to satisfy every possible combination, simple subclassing produces so many classes it becomes impractical.

The Decorator pattern uses layered objects to dynamically and transparently add responsibilities to individual objects. The pattern specifies that all objects that wrap around your initial object have the same basic interface. Something is decoratable, and you layer on functionality by wrapping other classes around the decoratable. This makes the decorators transparent—there are a set of common messages you can send to an object whether or not it is decorated. A decorating class can also add methods, but as you shall see, this is limited.

Decorators are implemented using composition and formal structures (the decoratable/decorator hierarchy), whereas mixins are inheritance-based. Think of parameterized-type-based mixins as a generic decorator mechanism that does not require the inheritance structure of the Decorator design pattern.

The previous example can be recast using Decorator:

[146] Note that some programming environments, such as Eclipse and IntelliJ Idea, will automatically generate delegation code.

```
// generics/decorator/Decoration.java
// {java generics.decorator.Decoration}
package generics.decorator;
import java.util.*;

class Basic {
  private String value;
  public void set(String val) { value = val; }
  public String get() { return value; }
}

class Decorator extends Basic {
  protected Basic basic;
  Decorator(Basic basic) { this.basic = basic; }
  @Override
  public void set(String val) { basic.set(val); }
  @Override
  public String get() { return basic.get(); }
}

class TimeStamped extends Decorator {
  private final long timeStamp;
  TimeStamped(Basic basic) {
    super(basic);
    timeStamp = new Date().getTime();
  }
  public long getStamp() { return timeStamp; }
}

class SerialNumbered extends Decorator {
  private static long counter = 1;
  private final long serialNumber = counter++;
  SerialNumbered(Basic basic) { super(basic); }
  public long getSerialNumber() { return serialNumber; }
}

public class Decoration {
  public static void main(String[] args) {
    TimeStamped t = new TimeStamped(new Basic());
    TimeStamped t2 = new TimeStamped(
      new SerialNumbered(new Basic()));
    //- t2.getSerialNumber(); // Not available
```

```
    SerialNumbered s = new SerialNumbered(new Basic());
    SerialNumbered s2 = new SerialNumbered(
      new TimeStamped(new Basic()));
    //- s2.getStamp(); // Not available
  }
}
```

The class resulting from a mixin contains all the methods of interest, but the type of the object that results from using decorators is the last type it was decorated with. That is, although it's *possible* to add more than one layer, the final layer is the actual type, so only the final layer's methods are visible, whereas the type of the mixin is *all* the types that get mixed together. So a significant drawback to Decorator is it only effectively works with one layer of decoration (the final one), and the mixin approach is arguably more natural. Thus, Decorator is only a limited solution to the problem addressed by mixins.

Mixins with Dynamic Proxies

It's possible to use a dynamic proxy to create a mechanism that more closely models mixins than does the Decorator (see the Reflection chapter for an explanation of how Java's dynamic proxies work). With a dynamic proxy, the *dynamic* type of the resulting class is the combined types that were mixed.

Because of the constraints of dynamic proxies, each class that is mixed in must be the implementation of an interface:

```
// generics/DynamicProxyMixin.java
import java.lang.reflect.*;
import java.util.*;
import onjava.*;
import static onjava.Tuple.*;

class MixinProxy implements InvocationHandler {
  Map<String, Object> delegatesByMethod;
  @SuppressWarnings("unchecked")
  MixinProxy(Tuple2<Object, Class<?>>... pairs) {
    delegatesByMethod = new HashMap<>();
    for(Tuple2<Object, Class<?>> pair : pairs) {
      for(Method method : pair.a2.getMethods()) {
```

```
        String methodName = method.getName();
        // The first interface in the map
        // implements the method.
        if(!delegatesByMethod.containsKey(methodName))
          delegatesByMethod.put(methodName, pair.a1);
      }
    }
  }
  @Override
  public Object invoke(Object proxy, Method method,
    Object[] args) throws Throwable {
    String methodName = method.getName();
    Object delegate = delegatesByMethod.get(methodName);
    return method.invoke(delegate, args);
  }
  @SuppressWarnings("unchecked")
  public static Object newInstance(Tuple2... pairs) {
    Class[] interfaces = new Class[pairs.length];
    for(int i = 0; i < pairs.length; i++) {
      interfaces[i] = (Class)pairs[i].a2;
    }
    ClassLoader cl =
      pairs[0].a1.getClass().getClassLoader();
    return Proxy.newProxyInstance(
      cl, interfaces, new MixinProxy(pairs));
  }
}

public class DynamicProxyMixin {
  public static void main(String[] args) {
    @SuppressWarnings("unchecked")
    Object mixin = MixinProxy.newInstance(
      tuple(new BasicImp(), Basic.class),
      tuple(new TimeStampedImp(), TimeStamped.class),
      tuple(new SerialNumberedImp(),
          SerialNumbered.class));
    Basic b = (Basic)mixin;
    TimeStamped t = (TimeStamped)mixin;
    SerialNumbered s = (SerialNumbered)mixin;
    b.set("Hello");
    System.out.println(b.get());
    System.out.println(t.getStamp());
```

```
    System.out.println(s.getSerialNumber());
  }
}
/* Output:
Hello
1611503350927
1
*/
```

Because only the dynamic type, and not the static type, includes all the mixed-in types, this is still not as nice as the C++ approach, because you're forced to downcast to the appropriate type before you can call methods for it. However, it is significantly closer to a true mixin.

There is a fair amount of work done towards the support of mixins for Java, including the creation of at least one language add-on, the Jam language, specifically for supporting mixins.

Latent Typing

The beginning of this chapter introduced the idea of writing code to apply as generally as possible. To do this, we need ways to loosen the constraints on the types our code works with, without losing the benefits of static type checking. We can then write code for use in more situations—that is, more "generic" code.

Java generics appear to take a further step in this direction. When you are writing or using generics that simply hold objects, the code works with any type (except for primitives, although autoboxing can fix this). Or, put another way, "holder" generics are able to say, "I don't care what type you are." Code that doesn't care what type it works with can indeed be applied everywhere, and is thus quite "generic."

You've seen the problem that arises when you perform manipulations on generic types (other than calling `Object` methods). Erasure forces you to specify the bounds of your generic types to safely call specific methods for the generic objects in your code. This is a significant limitation to the concept of "generic" because you must constrain your generic types so they inherit from particular classes or implement particular interfaces. In some cases you might end up using an ordinary class or

interface instead, because a bounded generic might be no different from specifying a class or interface.

One solution that some programming languages provide is called *latent typing* or *structural typing*. A more whimsical term is *duck typing*, as in, "If it walks like a duck and talks like a duck, you might as well treat it like a duck." Duck typing has become a fairly popular term, possibly because it doesn't carry the historical baggage that other terms do.

Generic code typically only calls a few methods on a generic type, and a language with latent typing loosens the constraint (and produces more generic code) by only requiring that a subset of methods be implemented, *not* a particular class or interface. Because of this, latent typing cuts across class hierarchies, calling methods that are not part of a common interface. So a piece of code might say, in effect, "I don't care what type you are as long as you can `speak()` and `sit()`." By not requiring a specific type, your code is more generic.

Latent typing is a code organization and reuse mechanism. With it you can write a piece of code that is easier to reuse than without it. Code organization and reuse are the foundational levers of all computer programming: Write it once, use it more than once, and keep the code in one place. Because I am not required to name an exact interface that my code operates upon, with latent typing I can write less code and apply it more easily in more places.

Languages that support latent typing include Python (from *www.Python.org*), C++, Ruby, SmallTalk, and Go. Python is a dynamically typed language (virtually all the type checking happens at run time), while C++ and Go are statically typed languages (the type checking happens at compile time), so latent typing does not require either static or dynamic type checking.

Latent Typing in Python

If we take the description of latent typing and express it in Python, it looks like this:

```
# generics/DogsAndRobots.py

class Dog:
    def speak(self):
        print("Arf!")
    def sit(self):
        print("Sitting")
    def reproduce(self):
        pass

class Robot:
    def speak(self):
        print("Click!")
    def sit(self):
        print("Clank!")
    def oilChange(self):
        pass

def perform(anything):
    anything.speak()
    anything.sit()

a = Dog()
b = Robot()
perform(a)
perform(b)

output = """
Arf!
Sitting
Click!
Clank!
"""
```

Python uses indentation to determine scope (so no curly braces are needed), and a colon to begin a new scope. A `#` indicates a comment to the end of the line, like `//` in Java. The methods of a class explicitly specify the equivalent of the `this` reference as the first argument, called `self` by convention. Constructor calls do not require any sort of "new" keyword. And Python allows regular (non-member) functions, as evidenced by `perform()`.

In `perform(anything)`, notice there is no type information for `anything`, and

`anything` is just an identifier. It must execute the operations that `perform()` asks of it, so an interface is implied. But you never explicitly write out that interface—it's *latent*. `perform()` doesn't care about the type of its argument, so I can pass any object to it as long as it supports the `speak()` and `sit()` methods. If you pass an object to `perform()` that does not support these operations, you'll get a runtime exception.

The `output` assignment uses triple quotes to create a string with embedded newlines.

Latent Typing in C++

We can produce the same effect in C++:

```
// generics/DogsAndRobots.cpp
#include <iostream>
using namespace std;

class Dog {
public:
  void speak() { cout << "Arf!" << endl; }
  void sit() { cout << "Sitting" << endl; }
  void reproduce() {}
};

class Robot {
public:
  void speak() { cout << "Click!" << endl; }
  void sit() { cout << "Clank!" << endl; }
  void oilChange() {}
};

template<class T> void perform(T anything) {
  anything.speak();
  anything.sit();
}

int main() {
  Dog d;
  Robot r;
  perform(d);
  perform(r);
```

```
}
/* Output:
Arf!
Sitting
Click!
Clank!
*/
```

In both Python and C++, `Dog` and `Robot` have nothing in common—they just happen to have two methods with identical signatures. From a type standpoint, they are completely distinct types. However, `perform()` doesn't care about the specific type of its argument, and latent typing allows it to accept both types of object.

The C++ compiler ensures it can actually send those messages. It gives you an error message if you try to pass the wrong type (these error messages have historically been terrible and verbose, and are the primary reason that C++ templates have a poor reputation). Although they do it at different times—C++ at compile time, and Python at run time—both languages ensure that types cannot be misused and are thus considered *strongly typed*.[147] Latent typing does not compromise strong typing.

Latent Typing in Go

Here's the same program written in Go:

```
// generics/dogsandrobots.go
package main
import "fmt"

type Dog struct {}
func (this Dog) speak() { fmt.Printf("Arf!\n")}
func (this Dog) sit() { fmt.Printf("Sitting\n")}
func (this Dog) reproduce() {}

type Robot struct {}
func (this Robot) speak() { fmt.Printf("Click!\n") }
func (this Robot) sit() { fmt.Printf("Clank!\n") }
func (this Robot) oilChange() {}
```

[147] Because you can use casts, which effectively disable the type system, some people argue that C++ is weakly typed, but that's extreme. It's probably safer to say that C++ is "strongly typed with a trap door."

```
func perform(speaker interface { speak(); sit() }) {
  speaker.speak();
  speaker.sit();
}

func main() {
  perform(Dog{})
  perform(Robot{})
}
/* Output:
Arf!
Sitting
Click!
Clank!
*/
```

Go has no `class` keyword, but you can create the equivalent of basic classes using the above form: what you would ordinarily define as a class, you instead define as a `struct`, within which dwell your data fields (there are none here). For each method, you start with the `func` keyword, then—to attach the method to your class—you put parentheses containing the object reference, which can be any identifier but I use `this` here to remind you that it's like the `this` in C++ or Java. Then you define the rest of the function as you do for any other function in Go.

There's also no inheritance in Go, so this form of "object-orientedness" is relatively primitive, and probably the main thing that keeps me from spending more time with the language. Composition, however, is straightforward.

The `perform()` function uses latent typing: the exact type of the argument is unimportant as long as it contains a `speak()` and `sit()` method. The `interface` is define here anonymously, inline, as seen in the argument list to `perform()`.

`main()` demonstrates that `perform()` is indeed indifferent to the exact type of its argument, as long as `speak()` and `sit()` can be called on that argument. However, just like C++ template functions, the types are checked at compile time.

The syntax `Dog{}` and `Robot{}` creates anonymous `Dog` and `Robot structs`.

Direct Latent Typing in Java

Because generics were added to Java late in the game, there was no chance that any kind of latent typing could be implemented, so Java has no support for this feature. As a result, it initially seems that Java's generic mechanism is "less generic" than a language that supports latent typing (The implementation of Java's generics using erasure is sometimes called *second-class* generic types). For example, if we try to implement the dogs-and-robots example before Java 8, we must use a class or an interface and specify it in a bounds expression:

```
// generics/Performs.java

public interface Performs {
  void speak();
  void sit();
}


// generics/DogsAndRobots.java
// No (direct) latent typing in Java
import reflection.pets.*;

class PerformingDog extends Dog implements Performs {
  @Override
  public void speak() { System.out.println("Woof!"); }
  @Override
  public void sit() { System.out.println("Sitting"); }
  public void reproduce() {}
}

class Robot implements Performs {
  @Override
  public void speak() { System.out.println("Click!"); }
  @Override
  public void sit() { System.out.println("Clank!"); }
  public void oilChange() {}
}

class Communicate {
  public static <T extends Performs>
  void perform(T performer) {
```

```
    performer.speak();
    performer.sit();
  }
}

public class DogsAndRobots {
  public static void main(String[] args) {
    Communicate.perform(new PerformingDog());
    Communicate.perform(new Robot());
  }
}
/* Output:
Woof!
Sitting
Click!
Clank!
*/
```

However, note that `perform()` does not need generics to work. It can be specified to accept a `Performs` object:

```
// generics/SimpleDogsAndRobots.java
// Removing the generic; code still works

class CommunicateSimply {
  static void perform(Performs performer) {
    performer.speak();
    performer.sit();
  }
}

public class SimpleDogsAndRobots {
  public static void main(String[] args) {
    CommunicateSimply.perform(new PerformingDog());
    CommunicateSimply.perform(new Robot());
  }
}
/* Output:
Woof!
Sitting
Click!
Clank!
*/
```

Here, generics were not necessary because the classes were already forced to implement the `Performs` interface.

Compensating for the Lack of (Direct) Latent Typing

Although Java does not directly support latent typing, this does not mean your generic code cannot be applied across different type hierarchies. You can create truly generic code, but it takes some extra effort.

Reflection

One approach you can use is reflection. Here's a reflective `perform()` that implements latent typing:

```
// generics/LatentReflection.java
// Using reflection for latent typing
import java.lang.reflect.*;

// Does not implement Performs:
class Mime {
  public void walkAgainstTheWind() {}
  public void sit() {
    System.out.println("Pretending to sit");
  }
  public void pushInvisibleWalls() {}
  @Override public String toString() { return "Mime"; }
}

// Does not implement Performs:
class SmartDog {
  public void speak() { System.out.println("Woof!"); }
  public void sit() { System.out.println("Sitting"); }
  public void reproduce() {}
}

class CommunicateReflectively {
  public static void perform(Object speaker) {
```

```
    Class<?> spkr = speaker.getClass();
    try {
      try {
        Method speak = spkr.getMethod("speak");
        speak.invoke(speaker);
      } catch(NoSuchMethodException e) {
        System.out.println(speaker + " cannot speak");
      }
      try {
        Method sit = spkr.getMethod("sit");
        sit.invoke(speaker);
      } catch(NoSuchMethodException e) {
        System.out.println(speaker + " cannot sit");
      }
    } catch(SecurityException |
            IllegalAccessException |
            IllegalArgumentException |
            InvocationTargetException e) {
      throw new RuntimeException(speaker.toString(), e);
    }
  }
}

public class LatentReflection {
  public static void main(String[] args) {
    CommunicateReflectively.perform(new SmartDog());
    CommunicateReflectively.perform(new Robot());
    CommunicateReflectively.perform(new Mime());
  }
}
/* Output:
Woof!
Sitting
Click!
Clank!
Mime cannot speak
Pretending to sit
*/
```

Here, the classes are completely disjoint and have no base classes (other than `Object`) or interfaces in common. Through reflection, `CommunicateReflectively.perform()` is able to dynamically establish whether the desired methods are available and call

them. It is even able to deal with the fact that `Mime` only has one of the necessary methods, and partially fulfills its goal.

Applying a Method to a Sequence

Reflection has useful possibilities, but it relegates all the type checking to run time, and is thus undesirable in many situations. If you can achieve compile-time type checking, that's usually more desirable. But can you have both compile-time type checking *and* latent typing?

Let's look at an example that explores the problem. Suppose you create an `apply()` method that applies any method to every object in a sequence. This is a situation where interfaces don't seem to fit. You want to apply any method to a collection of objects, and interfaces constrain you too much to describe "any method." How do you do this in Java?

Initially, we can solve the problem with reflection, which turns out to be fairly elegant using varargs:

```
// generics/Apply.java
import java.lang.reflect.*;
import java.util.*;

public class Apply {
  public static <T, S extends Iterable<T>>
  void apply(S seq, Method f, Object... args) {
    try {
      for(T t: seq)
        f.invoke(t, args);
    } catch(IllegalAccessException |
            IllegalArgumentException |
            InvocationTargetException e) {
      // Failures are programmer errors
      throw new RuntimeException(e);
    }
  }
}
```

Exceptions are converted to `RuntimeExceptions` because there's not much of a way to recover from exceptions—they really do represent programmer errors here.

Why don't we just use a Java 8 method reference (this is shown later) instead of the reflective `Method f`? Notice that `invoke()`, and thus `apply()`, have the advantage that they can take any number of arguments. There are situations when that flexibility might be essential.

To test `Apply`, we'll first create a `Shape` class:

```
// generics/Shape.java

public class Shape {
  private static long counter = 0;
  private final long id = counter++;
  @Override public String toString() {
    return getClass().getSimpleName() + " " + id;
  }
  public void rotate() {
    System.out.println(this + " rotate");
  }
  public void resize(int newSize) {
    System.out.println(this + " resize " + newSize);
  }
}
```

Followed by a subclass:

```
// generics/Square.java
public class Square extends Shape {}
```

Using these, we can test `Apply`:

```
// generics/ApplyTest.java
import java.util.*;
import java.util.function.*;
import onjava.*;

public class ApplyTest {
  public static
  void main(String[] args) throws Exception {
    List<Shape> shapes =
      Suppliers.create(ArrayList::new, Shape::new, 3);
    Apply.apply(shapes,
      Shape.class.getMethod("rotate"));
```

```
    Apply.apply(shapes,
      Shape.class.getMethod("resize", int.class), 7);

    List<Square> squares =
      Suppliers.create(ArrayList::new, Square::new, 3);
    Apply.apply(squares,
      Shape.class.getMethod("rotate"));
    Apply.apply(squares,
      Shape.class.getMethod("resize", int.class), 7);

    Apply.apply(new FilledList<>(Shape::new, 3),
      Shape.class.getMethod("rotate"));
    Apply.apply(new FilledList<>(Square::new, 3),
      Shape.class.getMethod("rotate"));

    SimpleQueue<Shape> shapeQ = Suppliers.fill(
      new SimpleQueue<>(), SimpleQueue::add,
      Shape::new, 3);
    Suppliers.fill(shapeQ, SimpleQueue::add,
      Square::new, 3);
    Apply.apply(shapeQ,
      Shape.class.getMethod("rotate"));
  }
}
/* Output:
Shape 0 rotate
Shape 1 rotate
Shape 2 rotate
Shape 0 resize 7
Shape 1 resize 7
Shape 2 resize 7
Square 3 rotate
Square 4 rotate
Square 5 rotate
Square 3 resize 7
Square 4 resize 7
Square 5 resize 7
Shape 6 rotate
Shape 7 rotate
Shape 8 rotate
Square 9 rotate
Square 10 rotate
```

```
Square 11 rotate
Shape 12 rotate
Shape 13 rotate
Shape 14 rotate
Square 15 rotate
Square 16 rotate
Square 17 rotate
*/
```

In `Apply`, we get lucky because there happens to be an `Iterable` interface built into Java, used by the Java collections library. Because of this, the `apply()` method can accept anything that implements the `Iterable` interface, which includes all the `Collection` classes such as `List`. But it can also accept anything else, as long as you make it `Iterable`—for example, the `SimpleQueue` class defined here and used above in `main()`:

```
// generics/SimpleQueue.java
// A different kind of Iterable collection
import java.util.*;

public class SimpleQueue<T> implements Iterable<T> {
  private LinkedList<T> storage = new LinkedList<>();
  public void add(T t) { storage.offer(t); }
  public T get() { return storage.poll(); }
  @Override public Iterator<T> iterator() {
    return storage.iterator();
  }
}
```

As elegant as the reflection solution seems, we must observe that reflection (although improved significantly in recent versions of Java) is usually slower than a non-reflection implementation because so much is happening at run time. This should not stop you from trying the solution, but it's certainly a point for consideration.

You'd almost certainly use the Java 8 functional approach first, and only resort to reflection if it solved a special need. Here's `ApplyTest.java` rewritten to take advantage of Java 8 streams and functional tools:

```
// generics/ApplyFunctional.java
import java.util.*;
import java.util.stream.*;
import java.util.function.*;
import onjava.*;

public class ApplyFunctional {
  public static void main(String[] args) {
    Stream.of(
      Stream.generate(Shape::new).limit(2),
      Stream.generate(Square::new).limit(2))
      .flatMap(c -> c) // flatten into one stream
      .peek(Shape::rotate)
      .forEach(s -> s.resize(7));

    new FilledList<>(Shape::new, 2)
      .forEach(Shape::rotate);
    new FilledList<>(Square::new, 2)
      .forEach(Shape::rotate);

    SimpleQueue<Shape> shapeQ = Suppliers.fill(
      new SimpleQueue<>(), SimpleQueue::add,
      Shape::new, 2);
    Suppliers.fill(shapeQ, SimpleQueue::add,
      Square::new, 2);
    shapeQ.forEach(Shape::rotate);
  }
}
/* Output:
Shape 0 rotate
Shape 0 resize 7
Shape 1 rotate
Shape 1 resize 7
Square 2 rotate
Square 2 resize 7
Square 3 rotate
Square 3 resize 7
Shape 4 rotate
Shape 5 rotate
Square 6 rotate
Square 7 rotate
Shape 8 rotate
```

```
Shape 9 rotate
Square 10 rotate
Square 11 rotate
*/
```

Because of Java 8, there's no need for `Apply.apply()`.

We start by generating two `Streams`: one of `Shape` and one of `Square`, and flattening them into a single stream. Although Java is missing the `flatten()` often found in functional languages, we can produce the same result with `flatMap(c -> c)`, which uses a identity mapping to reduce the operation to only "flatten."

We use `peek()` for the call to `rotate()` because `peek()` performs an operation (here, for its side effect) and passes the object on unchanged.

Notice how much cleaner the call to `forEach()` is with the `FilledLists` and `shapeQ` than `Apply.apply()`.

The result is far preferable to the previous approach, just in code simplicity and readability alone. Also there is now no possibility of an exception being thrown from `main()`.

Assisted Latent Typing in Java 8

The prior statements about Java's lack of support for latent typing are strictly true before Java 8. However, unbound method references in Java 8 allow us to produce a form of latent typing that satisfies the requirement of creating a single piece of code that works across unrelated types. Java wasn't originally designed to do this so the result is—as you might expect by now—slightly more awkward than in other languages. But it is now *possible*, which I consider just short of amazing.

I have not encountered this technique elsewhere, so I shall call it *assisted latent typing*.

We'll rewrite `DogsAndRobots.java` to demonstrate this technique. To make things look as similar as possible to the original example, I've simply added an `A` to each of the original class names:

```
// generics/DogsAndRobotMethodReferences.java
// "Assisted Latent Typing"
import reflection.pets.*;
import java.util.function.*;

class PerformingDogA extends Dog {
  public void speak() { System.out.println("Woof!"); }
  public void sit() { System.out.println("Sitting"); }
  public void reproduce() {}
}

class RobotA {
  public void speak() { System.out.println("Click!"); }
  public void sit() { System.out.println("Clank!"); }
  public void oilChange() {}
}

class CommunicateA {
  public static <P> void perform(P performer,
    Consumer<P> action1, Consumer<P> action2) {
    action1.accept(performer);
    action2.accept(performer);
  }
}

public class DogsAndRobotMethodReferences {
  public static void main(String[] args) {
    CommunicateA.perform(new PerformingDogA(),
      PerformingDogA::speak, PerformingDogA::sit);
    CommunicateA.perform(new RobotA(),
      RobotA::speak, RobotA::sit);
    CommunicateA.perform(new Mime(),
      Mime::walkAgainstTheWind,
      Mime::pushInvisibleWalls);
  }
}
/* Output:
Woof!
Sitting
Click!
Clank!
*/
```

`PerformingDogA` and `RobotA` are the same as they are in `DogsAndRobots.java`, except they do not inherit the common interface `Performs`, so they have no commonality.

`CommunicateA.perform()` is generified on `P` which has *no constraints*. It can be anything as long as there are `Consumer<P>`s available for it—here, those `Consumer<P>`s represent unbound method references for `P` methods that take no arguments. When you call `Consumer`'s `accept()` method, it binds the method reference to the performer object and calls that method. Because of the "magic" described in the Functional Programming chapter, we can pass any signature-conforming unbound method references to `CommunicateA.perform()`.

The reason for calling it "assisted" is because you must explicitly give `perform()` the method references to use; it can't just call the methods by name.

Although passing the unbound method references might seem like a lot of extra handholding, the ultimate goal of latent typing is achieved. We have created a single piece of code, `CommunicateA.perform()`, which works on any types that have signature-conformant method references. Notice this is somewhat different than latent typing in other languages we've seen, because those languages require not just the signature to conform, but also the method names. Thus, this technique arguably produces even *more* generic code.

Just to prove the point, I've also thrown in a `Mime` from `LatentReflection.java`.

Generic Methods to Use with `Suppliers`

With assisted latent typing, we can define the `Suppliers` class used in other parts of this chapter. This class contains utility methods that use generators to fill `Collections`. It makes sense to "generify" these operations:

```
// onjava/Suppliers.java
// A utility to use with Suppliers
package onjava;
import java.util.*;
import java.util.function.*;
import java.util.stream.*;

public class Suppliers {
  // Create a collection and fill it:
  public static <T, C extends Collection<T>> C
  create(Supplier<C> factory, Supplier<T> gen, int n) {
    return Stream.generate(gen)
      .limit(n)
      .collect(factory, C::add, C::addAll);
  }
  // Fill an existing collection:
  public static <T, C extends Collection<T>>
  C fill(C coll, Supplier<T> gen, int n) {
    Stream.generate(gen)
      .limit(n)
      .forEach(coll::add);
    return coll;
  }
  // Use an unbound method reference to
  // produce a more general method:
  public static <H, A> H fill(H holder,
    BiConsumer<H, A> adder, Supplier<A> gen, int n) {
    Stream.generate(gen)
      .limit(n)
      .forEach(a -> adder.accept(holder, a));
    return holder;
  }
}
```

`create()` makes a new `Collection` subtype for you, while the first version of `fill()` puts elements into an existing subtype of `Collection`. Notice the exact type of container passed in is also returned, so the type information is not lost.[148]

The first two methods are generically constrained to work with `Collection` subtypes. The second version of `fill()` works with a `holder` of any type. It takes an

[148] Once again, I got help from Brian Goetz.

additional argument: the unbound method reference `adder`. `fill()` uses assisted latent typing to make it work with any `holder` type that has a method to add elements. Because this unbound method `adder` must take an argument (the element to add to the `holder`), `adder` must be a `BiConsumer<H, A>` where `H` is the type of the `holder` object to bind to, and `A` is the type of element being added. The call to `accept()` invokes the unbound method `adder` on the object `holder` with the argument `a`.

The `Suppliers` utility is tested within a little simulation that also uses `RandomList`, defined earlier in the chapter:

```
// generics/BankTeller.java
// A very simple bank teller simulation
import java.util.*;
import onjava.*;

class Customer {
  private static long counter = 1;
  private final long id = counter++;
  @Override public String toString() {
    return "Customer " + id;
  }
}

class Teller {
  private static long counter = 1;
  private final long id = counter++;
  @Override public String toString() {
    return "Teller " + id;
  }
}

class Bank {
  private List<BankTeller> tellers =
    new ArrayList<>();
  public void put(BankTeller bt) {
    tellers.add(bt);
  }
}

public class BankTeller {
```

```
  public static void serve(Teller t, Customer c) {
    System.out.println(t + " serves " + c);
  }
  public static void main(String[] args) {
    // Demonstrate create():
    RandomList<Teller> tellers =
      Suppliers.create(
        RandomList::new, Teller::new, 4);
    // Demonstrate fill():
    List<Customer> customers = Suppliers.fill(
      new ArrayList<>(), Customer::new, 12);
    customers.forEach(c ->
      serve(tellers.select(), c));
    // Demonstrate assisted latent typing:
    Bank bank = Suppliers.fill(
      new Bank(), Bank::put, BankTeller::new, 3);
    // Can also use second version of fill():
    List<Customer> customers2 = Suppliers.fill(
      new ArrayList<>(),
      List::add, Customer::new, 12);
  }
}
/* Output:
Teller 3 serves Customer 1
Teller 2 serves Customer 2
Teller 3 serves Customer 3
Teller 1 serves Customer 4
Teller 1 serves Customer 5
Teller 3 serves Customer 6
Teller 1 serves Customer 7
Teller 2 serves Customer 8
Teller 3 serves Customer 9
Teller 3 serves Customer 10
Teller 2 serves Customer 11
Teller 4 serves Customer 12
*/
```

You can see that `create()` generates a new `Collection` object, while `fill()` adds to an existing `Collection`. The second version of `fill()` is shown not only working with the new and unrelated type `Bank`, but also with a `List`—thus the first version of `fill()` is not technically necessary but provides shorter syntax when working with `Collections`.

Summary: Is Casting Really So Bad?

Having worked to explain C++ templates since their inception, I have probably been putting forward the following argument longer than most people. Only recently have I stopped to wonder how often this argument is valid—how many times does the problem I'm about to describe really slip through the cracks?

The argument goes like this. One of the most compelling places to use a generic type mechanism is with collection classes such as the `Lists`, `Sets`, `Maps`, etc. you saw in Collections and you shall see more of in the Appendix: Collection Topics. Before Java 5, when you put an object into a collection, it was upcast to `Object`, so you'd lose the type information. When you wanted to pull it back out to do something with it, you had to cast it back down to the proper type. My example was a `List` of `Cat` (a variation of this using apples and oranges is shown at the beginning of the Collections chapter). Without the Java 5 generic version of the collection, you put `Objects` in and you get `Objects` out, so it's easily possible to put a `Dog` in a `List` of `Cat`.

However, pre-generic Java wouldn't let you *misuse* the objects you put into a collection. If you threw a `Dog` into a collection of `Cats`, then tried to treat everything in the collection as a `Cat`, you'd get a `RuntimeException` when you pulled the `Dog` reference out of the `Cat` collection and tried to cast it to a `Cat`. You'd still discover the problem, but at run time rather than compile time.

In earlier times, I went on to argue:

> This is more than just an annoyance. It's something that can create difficult-to-find bugs. If one part (or several parts) of a program inserts objects into a collection, and you discover only in a separate part of the program through an exception that a bad object was placed in the collection, then you must find out where the bad insert occurred.

However, upon further examination of the argument, I began to wonder about it. First, how often does it happen? I don't remember this kind of thing ever happening to me, and when I asked people at conferences, I didn't hear anyone say it had happened to them. Another book used an example of a list called `files` that contained `String` objects—in this example it seemed perfectly natural to add a `File` object to `files`, so

a better name for the object might be `fileNames`. No matter how much type checking Java provides, it's still possible to write obscure programs, and a badly written program that compiles is still a badly written program. Perhaps most people use well-named collections such as `cats` that provide a visual warning to the programmer who would try to add a non-`Cat`. And even if it did happen, how long would the issue really stay buried? It would seem that as soon as you started running tests with real data, you'd see an exception pretty quickly.

One author even asserted that such a bug could "remain buried for years." But I do not recall any deluge of reports of people having great difficulty finding "dog in cat list" bugs, or even producing them very often. With Concurrent Programming, it is easy and common for bugs to appear extremely rarely, and only give you a vague idea of what's wrong. So is the "dog in cat list" argument really the reason this very significant and fairly complex feature was added to Java?

I believe the *intent* of the general-purpose language feature called "generics" is *expressiveness*, not just creating type-safe collections. Type-safe collections come as a side effect of the ability to create more general-purpose code.

So even though the "dog in cat list" argument is often used to justify generics, it is questionable. And as I've asserted throughout this chapter, I do not believe this is what the *concept* of generics is really about. Instead, generics are as their name implies—a way to write more "generic" code that is less constrained by the types it can work with, so a single piece of code can be applied to more types. As you have seen in this chapter, it is fairly easy to write truly generic "holder" classes (which the Java collections are). To write generic code that manipulates its generic types requires extra effort, on the part of both the class creator *and* the class consumer, who must understand the concept and implementation of such code. That extra effort reduces the ease of use of the feature, and can thus make it less applicable in places where it might otherwise have added value.

Also note that because generics were back-engineered into Java instead of designed into the language from the start, some of the collections cannot be made as robust as they should be. For example, look at `Map`, in particular the methods `containsKey(Object key)` and `get(Object key)`. If these classes had been designed with pre-existing generics, these methods would have used parameterized types instead of `Object`, thus affording the compile-time checking that generics are meant to provide. In C++ `maps`, for example, the key type is always checked at compile time.

It is clear that introducing any kind of generic mechanism in a later version of a language, after that language has come into general use, is a very, very messy proposition, and one that cannot be accomplished without pain. In C++, templates were introduced in the initial ISO version of the language (although even that caused some distress because there was an earlier non-template version in use before the first Standard C++ appeared), so in effect templates were *always* a part of the language. In Java, generics were not introduced until almost 10 years after the language was first released, so the issues of migrating to generics are considerable, and have made a significant impact on the design of generics. The result is that you, the programmer, will suffer because of the lack of vision exhibited by the Java designers when they created version 1.0. When Java was first created, the designers knew about C++ templates, and they even considered including them in the language, but for one reason or another decided to leave them out (indications are they were in a hurry). As a result, both the language and the programmers that use it will suffer. Only time will show the ultimate impact that Java's approach to generics have on the language.

Some languages have incorporated cleaner and less impactful approaches to parameterized types. It's not impossible to imagine such a language becoming a successor to Java, because it takes exactly the approach that C++ did with C: Use what's there and improve upon it.

Further Reading

- The introductory document for generics is *Generics in the Java Programming Language*, by Gilad Bracha—Search from here[149].
- Angelika Langer's Java Generics FAQs[150] is a very helpful resource.
- You can find out more about wildcards in *Adding Wildcards to the Java Programming Language*, by Torgerson, Ernst, Hansen, von der Ahe, Bracha and Gafter, located at *www.jot.fm/issues/issue_2004_12/article5*.
- Neal Gafter's opinions on Java problems (erasure in particular) can be found here[151].

[149] http://java.oracle.com
[150] http://www.angelikalanger.com/GenericsFAQ/JavaGenericsFAQ.html
[151] http://www.infoq.com/articles/neal-gafter-on-java

Arrays

> At the end of the Housekeeping chapter, you learned to define and initialize an array.

The simple view of arrays is that you create and populate them, you select elements from them using `int` indexes, and they don't change their size. Most of the time that's all you must know, but sometimes you must perform more sophisticated operations on arrays, and you might also evaluate an array vs. a more flexible `Collection`. This chapter looks at arrays in more depth.

Note: As Java `Collections` and `Streams` have added more high-level capabilities, the need to use arrays in day-to-day programming has diminished, so you can safely skim or even skip this chapter for the time being. Ultimately, however, there will come a time when—even if you avoid using arrays yourself—you will need to read and understand someone else's array code. At that time, this chapter will still be here, waiting for you.

Why Arrays are Special

There are other ways to hold objects, so what makes an array special?

There are three issues that distinguish arrays from other types of `Collections`: efficiency, type, and the ability to hold primitives. The array is Java's most efficient way to store and randomly access a sequence of object references. The array is a simple linear sequence, which makes element access fast. The cost of this speed is that the size of an array object is fixed and cannot be changed for the lifetime of that array.

Speed is not usually an issue, and if it is, the way you hold and retrieve objects is rarely the culprit. You should always start with an `ArrayList` (from Collections), which uses and manages an array internally. When necessary, it automatically allocates more array space, creating a new array and moving all the references from

the old array to the new array. This flexibility has overhead, so an `ArrayList` is less efficient than an array. In the rare cases where this is an issue, you can use arrays directly.

Both arrays and `Collections` guarantee you can't abuse them. Whether you're using an array or a `Collection`, you'll get a `RuntimeException` if you exceed the bounds, indicating a programmer error.

Before generics, the other `Collection` classes dealt with objects as if they had no specific type. That is, they treated them as type `Object`, the root class of all classes in Java. Arrays are superior to *pre-generic* `Collections` because you create an array to hold a specific type. This means you get compile-time type checking to prevent you from inserting the wrong type or mistaking the type that you're extracting. Of course, Java prevents you from sending inappropriate messages to objects at either compile time or run time. So it's not riskier one way or the other; it's just nicer if the compiler points it out to you, and there's less likelihood that the end user will get surprised by an exception.

An array can hold primitives, whereas a pre-generic `Collection` could not. With generics, however, `Collections` can specify and check the type of objects they hold. With autoboxing, `Collections` can act as if they are able to hold primitives because the conversion is automatic. Here's a comparison between arrays and generic `Collections`:

```
// arrays/CollectionComparison.java
import java.util.*;
import onjava.*;
import static onjava.ArrayShow.*;

class BerylliumSphere {
  private static long counter;
  private final long id = counter++;
  @Override public String toString() {
    return "Sphere " + id;
  }
}

public class CollectionComparison {
  public static void main(String[] args) {
    BerylliumSphere[] spheres =
```

```
      new BerylliumSphere[10];
    for(int i = 0; i < 5; i++)
      spheres[i] = new BerylliumSphere();
    show(spheres);
    System.out.println(spheres[4]);

    List<BerylliumSphere> sphereList = Suppliers.create(
      ArrayList::new, BerylliumSphere::new, 5);
    System.out.println(sphereList);
    System.out.println(sphereList.get(4));

    int[] integers = { 0, 1, 2, 3, 4, 5 };
    show(integers);
    System.out.println(integers[4]);

    List<Integer> intList = new ArrayList<>(
      Arrays.asList(0, 1, 2, 3, 4, 5));
    intList.add(97);
    System.out.println(intList);
    System.out.println(intList.get(4));
  }
}
/* Output:
[Sphere 0, Sphere 1, Sphere 2, Sphere 3, Sphere 4,
null, null, null, null, null]
Sphere 4
[Sphere 5, Sphere 6, Sphere 7, Sphere 8, Sphere 9]
Sphere 9
[0, 1, 2, 3, 4, 5]
4
[0, 1, 2, 3, 4, 5, 97]
4
*/
```

`Suppliers.create()` was defined in the Generics chapter.

Both ways of holding objects are type-checked, and the only apparent difference is that arrays use `[ ]` for accessing elements, and a `List` uses methods such as `add()` and `get()`. The similarity between arrays and the `ArrayList` is intentional, so it's conceptually easy to switch between the two. But as you saw in the Collections chapter, `Collections` have significantly more functionality than arrays.

With the advent of autoboxing, `Collections` are nearly as easy to use for primitives as arrays. The only remaining advantage to arrays is efficiency. However, when you're solving a more general problem, arrays can be too restrictive, and in those cases you use a `Collection` class.

A Utility for Displaying Arrays

Throughout this chapter we must display arrays. Java provides `Arrays.toString()` to convert an array into a readable string, which we can then display on the console. However, this is visually noisy so we'll create a little library to do the work:

```
// onjava/ArrayShow.java
package onjava;
import java.util.*;

public interface ArrayShow {
  static void show(Object[] a) {
    System.out.println(Arrays.toString(a));
  }
  static void show(boolean[] a) {
    System.out.println(Arrays.toString(a));
  }
  static void show(byte[] a) {
    System.out.println(Arrays.toString(a));
  }
  static void show(char[] a) {
    System.out.println(Arrays.toString(a));
  }
  static void show(short[] a) {
    System.out.println(Arrays.toString(a));
  }
  static void show(int[] a) {
    System.out.println(Arrays.toString(a));
  }
  static void show(long[] a) {
    System.out.println(Arrays.toString(a));
  }
  static void show(float[] a) {
    System.out.println(Arrays.toString(a));
  }
```

```
  static void show(double[] a) {
    System.out.println(Arrays.toString(a));
  }
  // Start with a description:
  static void show(String info, Object[] a) {
    System.out.print(info + ": ");
    show(a);
  }
  static void show(String info, boolean[] a) {
    System.out.print(info + ": ");
    show(a);
  }
  static void show(String info, byte[] a) {
    System.out.print(info + ": ");
    show(a);
  }
  static void show(String info, char[] a) {
    System.out.print(info + ": ");
    show(a);
  }
  static void show(String info, short[] a) {
    System.out.print(info + ": ");
    show(a);
  }
  static void show(String info, int[] a) {
    System.out.print(info + ": ");
    show(a);
  }
  static void show(String info, long[] a) {
    System.out.print(info + ": ");
    show(a);
  }
  static void show(String info, float[] a) {
    System.out.print(info + ": ");
    show(a);
  }
  static void show(String info, double[] a) {
    System.out.print(info + ": ");
    show(a);
  }
}
```

The first version works for `Object` arrays, including arrays of wrapped primitives. All the overloaded versions are necessary for the various different primitive types.

The second overloaded group is so you can prefix the array display with an information `String`.

For simplicity, you'll normally import this statically.

Arrays are First-Class Objects

Regardless of what type of array you're working with, the array identifier is actually a reference to a true object that's created on the heap. This is the object that holds the references to the other objects, and it can be created either implicitly, as part of the array initialization syntax, or explicitly with a `new` expression. Part of the array object (in fact, the only field or method you can access) is the read-only `length` member that tells how many elements can be stored in that array object. The `[]` syntax is the only other access you have to the array object.

The following example summarizes the various ways to initialize an array, and how to assign array references to different array objects. It also shows that arrays of objects and arrays of primitives are almost identical in use. The only difference is that arrays of objects hold references, but arrays of primitives hold the primitive values directly.

```
// arrays/ArrayOptions.java
// Initialization & re-assignment of arrays
import java.util.*;
import static onjava.ArrayShow.*;

public class ArrayOptions {
  public static void main(String[] args) {
    // Arrays of objects:
    BerylliumSphere[] a; // Uninitialized local
    BerylliumSphere[] b = new BerylliumSphere[5];

    // The references inside the array are
    // automatically initialized to null:
    show("b", b);
    BerylliumSphere[] c = new BerylliumSphere[4];
```

```
for(int i = 0; i < c.length; i++)
  if(c[i] == null) // Can test for null reference
    c[i] = new BerylliumSphere();

// Aggregate initialization:
BerylliumSphere[] d = {
  new BerylliumSphere(),
  new BerylliumSphere(),
  new BerylliumSphere()
};

// Dynamic aggregate initialization:
a = new BerylliumSphere[]{
  new BerylliumSphere(), new BerylliumSphere(),
};
// (Trailing comma is optional)

System.out.println("a.length = " + a.length);
System.out.println("b.length = " + b.length);
System.out.println("c.length = " + c.length);
System.out.println("d.length = " + d.length);
a = d;
System.out.println("a.length = " + a.length);

// Arrays of primitives:
int[] e; // Null reference
int[] f = new int[5];

// The primitives inside the array are
// automatically initialized to zero:
show("f", f);
int[] g = new int[4];
for(int i = 0; i < g.length; i++)
  g[i] = i*i;
int[] h = { 11, 47, 93 };

//  Compile error: variable e not initialized:
//- System.out.println("e.length = " + e.length);
System.out.println("f.length = " + f.length);
System.out.println("g.length = " + g.length);
System.out.println("h.length = " + h.length);
e = h;
```

```
    System.out.println("e.length = " + e.length);
    e = new int[]{ 1, 2 };
    System.out.println("e.length = " + e.length);
  }
}
/* Output:
b: [null, null, null, null, null]
a.length = 2
b.length = 5
c.length = 4
d.length = 3
a.length = 3
f: [0, 0, 0, 0, 0]
f.length = 5
g.length = 4
h.length = 3
e.length = 3
e.length = 2
*/
```

The array `a` is an uninitialized local variable, and the compiler prevents you from doing anything with this reference until you've properly initialized it. The array `b` is initialized to point to an array of `BerylliumSphere` references, but no actual `BerylliumSphere` objects are ever placed in that array. However, you can still ask what the size of the array is because `b` is pointing to a legitimate object. This brings up a slight drawback: You can't find out how many elements are actually *in* the array because `length` tells you only how many elements *can* be placed in the array; that is, the size of the array object, not the number of elements it actually holds. However, when you create an array object, its references are automatically initialized to `null`, so you test whether a particular array slot has an object in it by checking to see whether it's `null`. Similarly, an array of primitives is automatically initialized to zero for numeric types, `(char)0` for `char`, and `false` for `boolean`.

Array `c` shows the creation of the array object followed by the assignment of `BerylliumSphere` objects to all the slots in the array. Array `d` shows the "aggregate initialization" syntax that causes the array object to be created (implicitly with `new` on the heap, just like for array `c`) *and* initialized with `BerylliumSphere` objects, all in one statement.

The next array initialization can be thought of as a "dynamic aggregate initialization."

The aggregate initialization used by `d` must be used at the point of `d`'s definition, but with the second syntax you can create and initialize an array object anywhere. For example, suppose `hide()` is a method that takes an array of `BerylliumSphere` objects. You call it by saying:

```
hide(d);
```

You can also dynamically create the array you pass as the argument:

```
hide(new BerylliumSphere[]{
  new BerylliumSphere(),
  new BerylliumSphere()
});
```

In many situations this syntax is a more convenient way to write code.

The expression:

```
a = d;
```

shows how you can take a reference that's attached to one array object and assign it to another array object, just as you can do with any other type of object reference. Now both `a` and `d` are pointing to the same array object on the heap.

The second part of `ArrayOptions.java` shows that primitive arrays work just like object arrays *except* that primitive arrays hold the primitive values directly.

Returning an Array

Suppose you write a method that returns not one element, but many elements. Languages like C and C++ make this difficult because you can't just return an array, only a pointer to an array. This introduces problems because it becomes messy to control the lifetime of the array, which leads to memory leaks.

In Java, you just return the array. You never worry about responsibility for that array—it is around as long as you need it, and the garbage collector will clean it up when you're done.

Here, we return an array of `String`:

```
// arrays/IceCreamFlavors.java
// Returning arrays from methods
import java.util.*;
import static onjava.ArrayShow.*;

public class IceCreamFlavors {
  private static SplittableRandom rand =
    new SplittableRandom(47);
  static final String[] FLAVORS = {
    "Chocolate", "Strawberry", "Vanilla Fudge Swirl",
    "Mint Chip", "Mocha Almond Fudge", "Rum Raisin",
    "Praline Cream", "Mud Pie"
  };
  public static String[] flavorSet(int n) {
    if(n > FLAVORS.length)
      throw new IllegalArgumentException("Set too big");
    String[] results = new String[n];
    boolean[] picked = new boolean[FLAVORS.length];
    for(int i = 0; i < n; i++) {
      int t;
      do
        t = rand.nextInt(FLAVORS.length);
      while(picked[t]);
      results[i] = FLAVORS[t];
      picked[t] = true;
    }
    return results;
  }
  public static void main(String[] args) {
    for(int i = 0; i < 7; i++)
      show(flavorSet(3));
  }
}
/* Output:
[Praline Cream, Mint Chip, Vanilla Fudge Swirl]
[Strawberry, Vanilla Fudge Swirl, Mud Pie]
[Chocolate, Strawberry, Vanilla Fudge Swirl]
[Rum Raisin, Praline Cream, Chocolate]
[Mint Chip, Rum Raisin, Mocha Almond Fudge]
[Mocha Almond Fudge, Mud Pie, Vanilla Fudge Swirl]
[Mocha Almond Fudge, Mud Pie, Mint Chip]
*/
```

`flavorSet()` creates an array of `String` called `results`. The size of this array is `n`, determined by the argument you pass into the method. Then it chooses flavors randomly from the array `FLAVORS` and places them into `results`, which it returns. Returning an array is just like returning any other object—it's a reference. It's not important that the array was created within `flavorSet()`, or that the array was created anyplace else, for that matter. The garbage collector takes care of cleaning up the array when you're done with it, and the array will persist for as long as you need it.

If you must return several elements of different types, you can instead use the `tuple` utilities introduced in Generics.

As an aside, when `flavorSet()` chooses flavors randomly, it ensures that a particular choice hasn't already been selected. This is performed in a `do` loop that keeps making random choices until it finds one not already in the `picked` array. (A `String` comparison would also show whether the random choice is already in the `results` array.) If it's successful, it adds the entry and finds the next one (`i` gets incremented). The output shows that `flavorSet()` chooses flavors in a random order each time.

Up until now, random numbers have been generated using the `java.util.Random` class that has been with the language since version 1.0, and has even been updated to provide Java 8 streams. Now we introduce the Java 8 `SplittableRandom`, which not only works with parallel operations (which you'll learn about eventually), but provides higher-quality random numbers. We'll use `SplittableRandom` throughout the rest of the book.

Multidimensional Arrays

To create a multidimensional array of primitives, you delimit each vector in the array using curly braces:

```
// arrays/MultidimensionalPrimitiveArray.java
import java.util.*;

public class MultidimensionalPrimitiveArray {
  public static void main(String[] args) {
    int[][] a = {
      { 1, 2, 3, },
      { 4, 5, 6, },
    };
    System.out.println(Arrays.deepToString(a));
  }
}
/* Output:
[[1, 2, 3], [4, 5, 6]]
*/
```

Each nested set of curly braces moves you into the next level of the array.

This example uses the `Arrays.deepToString()` method, which turns multidimensional arrays into `String`s, as shown in the output.

You can also allocate an array using `new`. Here's a three-dimensional array allocated in a `new` expression:

```
// arrays/ThreeDWithNew.java
import java.util.*;

public class ThreeDWithNew {
  public static void main(String[] args) {
    // 3-D array with fixed length:
    int[][][] a = new int[2][2][4];
    System.out.println(Arrays.deepToString(a));
  }
}
/* Output:
[[[0, 0, 0, 0], [0, 0, 0, 0]], [[0, 0, 0, 0], [0, 0, 0,
0]]]
*/
```

Primitive array values are automatically initialized if you don't give them an explicit initialization value. Arrays of objects are initialized to `null`.

Each vector in the arrays that make up the matrix can be of any length (this is called a *ragged array*):

```
// arrays/RaggedArray.java
import java.util.*;

public class RaggedArray {
  static int val = 1;
  public static void main(String[] args) {
    SplittableRandom rand = new SplittableRandom(47);
    // 3-D array with varied-length vectors:
    int[][][] a = new int[rand.nextInt(7)][][];
    for(int i = 0; i < a.length; i++) {
      a[i] = new int[rand.nextInt(5)][];
      for(int j = 0; j < a[i].length; j++) {
        a[i][j] = new int[rand.nextInt(5)];
        Arrays.setAll(a[i][j], n -> val++);     // [1]
      }
    }
    System.out.println(Arrays.deepToString(a));
  }
}
/* Output:
[[[1], []], [[2, 3, 4, 5], [6]], [[7, 8, 9], [10, 11,
12], []]]
*/
```

The first `new` creates an array with a random-length first element and the rest undetermined. The second `new` inside the `for` loop fills out the elements but leaves the third index undetermined until you hit the third `new`.

- **[1]** Java 8 added `Arrays.setAll()` which uses a generator to produce values inserted into the array. This generator conforms to the functional interface `IntUnaryOperator` with a single non-`default` method `applyAsInt(int operand)`. `Arrays.setAll()` passes the current array index as the `operand`, so one option is to provide a lambda of `n -> n` to show the index in the array (it's easy to try in the above code). Here, we ignore the index and simply insert the value of an incremented counter.

Arrays of non-primitive objects can also be defined as ragged arrays. Here, we collect many `new` expressions using curly braces:

```
// arrays/MultidimensionalObjectArrays.java
import java.util.*;

public class MultidimensionalObjectArrays {
  public static void main(String[] args) {
    BerylliumSphere[][] spheres = {
      { new BerylliumSphere(), new BerylliumSphere() },
      { new BerylliumSphere(), new BerylliumSphere(),
        new BerylliumSphere(), new BerylliumSphere() },
      { new BerylliumSphere(), new BerylliumSphere(),
        new BerylliumSphere(), new BerylliumSphere(),
        new BerylliumSphere(), new BerylliumSphere(),
        new BerylliumSphere(), new BerylliumSphere() },
    };
    System.out.println(Arrays.deepToString(spheres));
  }
}
/* Output:
[[Sphere 0, Sphere 1], [Sphere 2, Sphere 3, Sphere 4,
Sphere 5], [Sphere 6, Sphere 7, Sphere 8, Sphere 9,
Sphere 10, Sphere 11, Sphere 12, Sphere 13]]
*/
```

Autoboxing works with array initializers:

```
// arrays/AutoboxingArrays.java
import java.util.*;

public class AutoboxingArrays {
  public static void main(String[] args) {
    Integer[][] a = { // Autoboxing:
      { 1, 2, 3, 4, 5, 6, 7, 8, 9, 10 },
      { 21, 22, 23, 24, 25, 26, 27, 28, 29, 30 },
      { 51, 52, 53, 54, 55, 56, 57, 58, 59, 60 },
      { 71, 72, 73, 74, 75, 76, 77, 78, 79, 80 },
    };
    System.out.println(Arrays.deepToString(a));
  }
}
/* Output:
[[1, 2, 3, 4, 5, 6, 7, 8, 9, 10], [21, 22, 23, 24, 25,
26, 27, 28, 29, 30], [51, 52, 53, 54, 55, 56, 57, 58,
```

```
59, 60], [71, 72, 73, 74, 75, 76, 77, 78, 79, 80]]
*/
```

Here's how an array of non-primitive objects can be built up piece-by-piece:

```
// arrays/AssemblingMultidimensionalArrays.java
// Creating multidimensional arrays
import java.util.*;

public class AssemblingMultidimensionalArrays {
  public static void main(String[] args) {
    Integer[][] a;
    a = new Integer[3][];
    for(int i = 0; i < a.length; i++) {
      a[i] = new Integer[3];
      for(int j = 0; j < a[i].length; j++)
        a[i][j] = i * j; // Autoboxing
    }
    System.out.println(Arrays.deepToString(a));
  }
}
/* Output:
[[0, 0, 0], [0, 1, 2], [0, 2, 4]]
*/
```

The `i * j` is only there to put an interesting value into the `Integer`.

The `Arrays.deepToString()` method works with both primitive arrays and object arrays:

```
// arrays/MultiDimWrapperArray.java
// Multidimensional arrays of "wrapper" objects
import java.util.*;

public class MultiDimWrapperArray {
  public static void main(String[] args) {
    Integer[][] a1 = { // Autoboxing
      { 1, 2, 3, },
      { 4, 5, 6, },
    };
    Double[][][] a2 = { // Autoboxing
      { { 1.1, 2.2 }, { 3.3, 4.4 } },
```

```
      { { 5.5, 6.6 }, { 7.7, 8.8 } },
      { { 9.9, 1.2 }, { 2.3, 3.4 } },
    };
    String[][] a3 = {
      { "The", "Quick", "Sly", "Fox" },
      { "Jumped", "Over" },
      { "The", "Lazy", "Brown", "Dog", "&", "friend" },
    };
    System.out.println(
      "a1: " + Arrays.deepToString(a1));
    System.out.println(
      "a2: " + Arrays.deepToString(a2));
    System.out.println(
      "a3: " + Arrays.deepToString(a3));
  }
}
/* Output:
a1: [[1, 2, 3], [4, 5, 6]]
a2: [[[1.1, 2.2], [3.3, 4.4]], [[5.5, 6.6], [7.7,
8.8]], [[9.9, 1.2], [2.3, 3.4]]]
a3: [[The, Quick, Sly, Fox], [Jumped, Over], [The,
Lazy, Brown, Dog, &, friend]]
*/
```

Again, in the `Integer` and `Double` arrays, autoboxing creates the wrapper objects for you.

Arrays and Generics

In general, arrays and generics do not mix well. You cannot instantiate arrays of parameterized types:

```
Peel<Banana>[] peels = new Peel<Banana>[10]; // Illegal
```

Erasure removes the parameter type information, and arrays must know the exact type they hold, to enforce type safety.

However, you can parameterize the type of the array itself:

```
// arrays/ParameterizedArrayType.java

class ClassParameter<T> {
  public T[] f(T[] arg) { return arg; }
}

class MethodParameter {
  public static <T> T[] f(T[] arg) { return arg; }
}

public class ParameterizedArrayType {
  public static void main(String[] args) {
    Integer[] ints = { 1, 2, 3, 4, 5 };
    Double[] doubles = { 1.1, 2.2, 3.3, 4.4, 5.5 };
    Integer[] ints2 =
      new ClassParameter<Integer>().f(ints);
    Double[] doubles2 =
      new ClassParameter<Double>().f(doubles);
    ints2 = MethodParameter.f(ints);
    doubles2 = MethodParameter.f(doubles);
  }
}
```

It's convenient to use a parameterized method instead of a parameterized class. You don't instantiate a class with a parameter for each different type you apply it to, and you can make it `static`. You can't always choose to use a parameterized method instead of a parameterized class, but it's usually preferable.

It's not precisely correct to say you cannot create arrays of generic types. True, the compiler won't let you *instantiate* an array of a generic type. However, it will let you create a reference to such an array. For example:

```
List<String>[] ls;
```

This passes through the compiler without complaint. And although you cannot create an actual array object that holds generics, you can create an array of the non-generified type and cast it:

```
// arrays/ArrayOfGenerics.java
import java.util.*;

public class ArrayOfGenerics {
  @SuppressWarnings("unchecked")
  public static void main(String[] args) {
    List<String>[] ls;
    List[] la = new List[10];
    ls = (List<String>[])la; // Unchecked cast
    ls[0] = new ArrayList<>();

    //- ls[1] = new ArrayList<Integer>();
    // error: incompatible types: ArrayList<Integer>
    // cannot be converted to List<String>
    //     ls[1] = new ArrayList<Integer>();
    //             ^

    // The problem: List<String> is a subtype of Object
    Object[] objects = ls; // So assignment is OK
    // Compiles and runs without complaint:
    objects[1] = new ArrayList<>();

    // However, if your needs are straightforward it is
    // possible to create an array of generics, albeit
    // with an "unchecked cast" warning:
    List<BerylliumSphere>[] spheres =
      (List<BerylliumSphere>[])new List[10];
    Arrays.setAll(spheres, n -> new ArrayList<>());
  }
}
```

Once you have a reference to a `List<String>[]`, you see you get some compile-time checking. The problem is that arrays are covariant, so a `List<String>[]` is also an `Object[]`, and you can use this to assign an `ArrayList<Integer>` into your array, with no error at either compile time or run time.

If you know you're not going to upcast and your needs are relatively simple, however, it is possible to create an array of generics, which will provide basic compile-time type checking. However, a generic `Collection` will virtually always be a better choice than an array of generics.

In general you'll find that generics are effective at the *boundaries* of a class or method.

In the interiors, erasure usually makes generics unusable. So you cannot, for example, create an array of a generic type:

```
// arrays/ArrayOfGenericType.java

public class ArrayOfGenericType<T> {
  T[] array; // OK
  @SuppressWarnings("unchecked")
  public ArrayOfGenericType(int size) {
    // error: generic array creation:
    //- array = new T[size];
    array = (T[])new Object[size]; // unchecked cast
  }
  // error: generic array creation:
  //- public <U> U[] makeArray() { return new U[10]; }
}
```

Erasure gets in the way again—this example attempts to create arrays of types that have been erased, and are thus unknown types. You can create an array of `Object`, and cast it, but without the `@SuppressWarnings` annotation you get an "unchecked" warning at compile time because the array doesn't really hold or dynamically check for type `T`. That is, if I create a `String[]`, Java will enforce, at both compile time and run time, that I can only place `String` objects in that array. However, if I create an `Object[]`, I can put anything into that array except primitive types.

`Arrays.fill()`

When experimenting with arrays, and with programs in general, it's helpful to easily generate arrays filled with test data. The Java standard library `Arrays` class includes a trivial `fill()` method that duplicates a single value into each location, or in the case of objects, copies the same reference into each location:

```
// arrays/FillingArrays.java
// Using Arrays.fill()
import java.util.*;
import static onjava.ArrayShow.*;

public class FillingArrays {
  public static void main(String[] args) {
    int size = 6;
    boolean[] a1 = new boolean[size];
    byte[] a2 = new byte[size];
    char[] a3 = new char[size];
    short[] a4 = new short[size];
    int[] a5 = new int[size];
    long[] a6 = new long[size];
    float[] a7 = new float[size];
    double[] a8 = new double[size];
    String[] a9 = new String[size];
    Arrays.fill(a1, true);
    show("a1", a1);
    Arrays.fill(a2, (byte)11);
    show("a2", a2);
    Arrays.fill(a3, 'x');
    show("a3", a3);
    Arrays.fill(a4, (short)17);
    show("a4", a4);
    Arrays.fill(a5, 19);
    show("a5", a5);
    Arrays.fill(a6, 23);
    show("a6", a6);
    Arrays.fill(a7, 29);
    show("a7", a7);
    Arrays.fill(a8, 47);
    show("a8", a8);
    Arrays.fill(a9, "Hello");
    show("a9", a9);
    // Manipulating ranges:
    Arrays.fill(a9, 3, 5, "World");
    show("a9", a9);
  }
}
/* Output:
a1: [true, true, true, true, true, true]
```

```
a2: [11, 11, 11, 11, 11, 11]
a3: [x, x, x, x, x, x]
a4: [17, 17, 17, 17, 17, 17]
a5: [19, 19, 19, 19, 19, 19]
a6: [23, 23, 23, 23, 23, 23]
a7: [29.0, 29.0, 29.0, 29.0, 29.0, 29.0]
a8: [47.0, 47.0, 47.0, 47.0, 47.0, 47.0]
a9: [Hello, Hello, Hello, Hello, Hello, Hello]
a9: [Hello, Hello, Hello, World, World, Hello]
*/
```

You can either fill the entire array or, as the last two statements show, fill a range of elements. You can only call `Arrays.fill()` with a single data value so the results are not especially useful.

`Arrays.setAll()`

Introduced in `RaggedArray.java` and used again in `ArrayOfGenerics.java`, `Arrays.setAll()` was added in Java 8. It takes a generator and produces different values, optionally based on the index element of the array (with access to the current index, your generator can read the array value and modify it). The overloaded signatures of the `static Arrays.setAll()` are:

- `void setAll(int[] a, IntUnaryOperator gen)`
- `void setAll(long[] a, IntToLongFunction gen)`
- `void setAll(double[] a, IntToDoubleFunction gen)`
- `<T> void setAll(T[] a, IntFunction<? extends T> gen)`

There are special versions for `int`, `long`, and `double`, and everything else is handled by the generic version. The generators are not `Suppliers` because those take no arguments, and these must take the `int` array index as an argument.

This very simple `setAll()` example uses trivial lambda expressions and method references:

```
// arrays/SimpleSetAll.java
import java.util.*;
import static onjava.ArrayShow.*;

class Bob {
  final int id;
  Bob(int n) { id = n; }
  @Override public String toString() {
    return "Bob" + id;
  }
}

public class SimpleSetAll {
  public static final int SZ = 8;
  static int val = 1;
  static char[] chars = "abcdefghijklmnopqrstuvwxyz"
    .toCharArray();
  static char getChar(int n) { return chars[n]; }
  public static void main(String[] args) {
    int[] ia = new int[SZ];
    long[] la = new long[SZ];
    double[] da = new double[SZ];
    Arrays.setAll(ia, n -> n);                // [1]
    Arrays.setAll(la, n -> n);
    Arrays.setAll(da, n -> n);
    show(ia);
    show(la);
    show(da);
    Arrays.setAll(ia, n -> val++);            // [2]
    Arrays.setAll(la, n -> val++);
    Arrays.setAll(da, n -> val++);
    show(ia);
    show(la);
    show(da);

    Bob[] ba = new Bob[SZ];
    Arrays.setAll(ba, Bob::new);              // [3]
    show(ba);

    Character[] ca = new Character[SZ];
    Arrays.setAll(ca, SimpleSetAll::getChar); // [4]
    show(ca);
```

```
  }
}
/* Output:
[0, 1, 2, 3, 4, 5, 6, 7]
[0, 1, 2, 3, 4, 5, 6, 7]
[0.0, 1.0, 2.0, 3.0, 4.0, 5.0, 6.0, 7.0]
[1, 2, 3, 4, 5, 6, 7, 8]
[9, 10, 11, 12, 13, 14, 15, 16]
[17.0, 18.0, 19.0, 20.0, 21.0, 22.0, 23.0, 24.0]
[Bob0, Bob1, Bob2, Bob3, Bob4, Bob5, Bob6, Bob7]
[a, b, c, d, e, f, g, h]
*/
```

- **[1]** Here, we just insert the array index as the value. This is automatically converted for the `long` and `double` versions.
- **[2]** The function only needs to accept the index and produce an appropriate result. Here we ignore the index value and produce the result using `val`.
- **[3]** The method reference works because `Bob`'s constructor takes an `int` argument. As long as the function we pass takes an `int` argument and produces the expected result, it does the job.
- **[4]** To deal with a primitive type other than `int`, `long`, or `double`, make the array of the wrapper type for your primitive. Then the generic version of `setAll()` is used. Notice that `getChar()` produces the primitive type, so this is autoboxed to `Character`.

Incremental Generators

Here is a package of methods to produce incremental values for different types.

These are written as inner classes to produce easy-to-remember names; for example, to use the `Integer` tool you say `new Count.Integer()`, and if you want the primitive `int` tool you say `new Count.Pint()` (the primitive names could not be used directly, so they are all preceded by a `P` for "primitive"—my first choice was to use the primitive name followed by a trailing underscore, such as `int_` and `double_`, but that violates Java naming conventions). Each generator for a wrapper class also implements its associated `Supplier` with a `get()` method. To work

with `Arrays.setAll()`, an overloaded `get(int n)` method takes (and ignores) its argument so it accepts the index value passed by `setAll()`.

Note that, by using the name of a wrapper class as an inner class name, we must qualify the name of the actual wrapper with `java.lang`:

```
// onjava/Count.java
// Generate incremental values of different types
package onjava;
import java.util.*;
import java.util.function.*;
import static onjava.ConvertTo.*;

public interface Count {
  class Boolean
  implements Supplier<java.lang.Boolean> {
    private boolean b = true;
    @Override public java.lang.Boolean get() {
      b = !b;
      return java.lang.Boolean.valueOf(b);
    }
    public java.lang.Boolean get(int n) {
      return get();
    }
    public java.lang.Boolean[] array(int sz) {
      java.lang.Boolean[] result =
        new java.lang.Boolean[sz];
      Arrays.setAll(result, n -> get());
      return result;
    }
  }
  class Pboolean {
    private boolean b = true;
    public boolean get() {
      b = !b;
      return b;
    }
    public boolean get(int n) { return get(); }
    public boolean[] array(int sz) {
      return primitive(new Boolean().array(sz));
    }
  }
```

```
class Byte
implements Supplier<java.lang.Byte> {
  private byte b;
  @Override
  public java.lang.Byte get() { return b++; }
  public java.lang.Byte get(int n) {
    return get();
  }
  public java.lang.Byte[] array(int sz) {
    java.lang.Byte[] result =
      new java.lang.Byte[sz];
    Arrays.setAll(result, n -> get());
    return result;
  }
}
class Pbyte {
  private byte b;
  public byte get() { return b++; }
  public byte get(int n) { return get(); }
  public byte[] array(int sz) {
    return primitive(new Byte().array(sz));
  }
}
char[] CHARS =
  "abcdefghijklmnopqrstuvwxyz".toCharArray();
class Character
implements Supplier<java.lang.Character> {
  private int i;
  @Override public java.lang.Character get() {
    i = (i + 1) % CHARS.length;
    return CHARS[i];
  }
  public java.lang.Character get(int n) {
    return get();
  }
  public java.lang.Character[] array(int sz) {
    java.lang.Character[] result =
      new java.lang.Character[sz];
    Arrays.setAll(result, n -> get());
    return result;
  }
}
```

```
  class Pchar {
    private int i;
    public char get() {
      i = (i + 1) % CHARS.length;
      return CHARS[i];
    }
    public char get(int n) { return get(); }
    public char[] array(int sz) {
      return primitive(new Character().array(sz));
    }
  }
  class Short
  implements Supplier<java.lang.Short> {
    short s;
    @Override
    public java.lang.Short get() { return s++; }
    public java.lang.Short get(int n) {
      return get();
    }
    public java.lang.Short[] array(int sz) {
      java.lang.Short[] result =
        new java.lang.Short[sz];
      Arrays.setAll(result, n -> get());
      return result;
    }
  }
  class Pshort {
    short s;
    public short get() { return s++; }
    public short get(int n) { return get(); }
    public short[] array(int sz) {
      return primitive(new Short().array(sz));
    }
  }
  class Integer
  implements Supplier<java.lang.Integer> {
    int i;
    @Override
    public java.lang.Integer get() { return i++; }
    public java.lang.Integer get(int n) {
      return get();
    }
```

```
  public java.lang.Integer[] array(int sz) {
    java.lang.Integer[] result =
      new java.lang.Integer[sz];
    Arrays.setAll(result, n -> get());
    return result;
  }
}
class Pint implements IntSupplier {
  int i;
  public int get() { return i++; }
  public int get(int n) { return get(); }
  @Override
  public int getAsInt() { return get(); }
  public int[] array(int sz) {
    return primitive(new Integer().array(sz));
  }
}
class Long
implements Supplier<java.lang.Long> {
  private long l;
  @Override
  public java.lang.Long get() { return l++; }
  public java.lang.Long get(int n) {
    return get();
  }
  public java.lang.Long[] array(int sz) {
    java.lang.Long[] result =
      new java.lang.Long[sz];
    Arrays.setAll(result, n -> get());
    return result;
  }
}
class Plong implements LongSupplier {
  private long l;
  public long get() { return l++; }
  public long get(int n) { return get(); }
  @Override
  public long getAsLong() { return get(); }
  public long[] array(int sz) {
    return primitive(new Long().array(sz));
  }
}
```

```
  class Float
  implements Supplier<java.lang.Float> {
    private int i;
    @Override public java.lang.Float get() {
      return java.lang.Float.valueOf(i++);
    }
    public java.lang.Float get(int n) {
      return get();
    }
    public java.lang.Float[] array(int sz) {
      java.lang.Float[] result =
        new java.lang.Float[sz];
      Arrays.setAll(result, n -> get());
      return result;
    }
  }
  class Pfloat {
    private int i;
    public float get() { return i++; }
    public float get(int n) { return get(); }
    public float[] array(int sz) {
      return primitive(new Float().array(sz));
    }
  }
  class Double
  implements Supplier<java.lang.Double> {
    private int i;
    @Override public java.lang.Double get() {
      return java.lang.Double.valueOf(i++);
    }
    public java.lang.Double get(int n) {
      return get();
    }
    public java.lang.Double[] array(int sz) {
      java.lang.Double[] result =
        new java.lang.Double[sz];
      Arrays.setAll(result, n -> get());
      return result;
    }
  }
  class Pdouble implements DoubleSupplier {
    private int i;
```

```
    public double get() { return i++; }
    public double get(int n) { return get(); }
    @Override
    public double getAsDouble() { return get(0); }
    public double[] array(int sz) {
      return primitive(new Double().array(sz));
    }
  }
}
```

For the three primitive types `int`, `long` and `double` where special `Supplier` interfaces are available, `Pint`, `Plong` and `Pdouble` implement those interfaces.

Here is a test for `Count`, which also gives examples of how to use it:

```
// arrays/TestCount.java
// Test counting generators
import java.util.*;
import java.util.stream.*;
import onjava.*;
import static onjava.ArrayShow.*;

public class TestCount {
  static final int SZ = 5;
  public static void main(String[] args) {
    System.out.println("Boolean");
    Boolean[] a1 = new Boolean[SZ];
    Arrays.setAll(a1, new Count.Boolean()::get);
    show(a1);
    a1 = Stream.generate(new Count.Boolean())
      .limit(SZ + 1).toArray(Boolean[]::new);
    show(a1);
    a1 = new Count.Boolean().array(SZ + 2);
    show(a1);
    boolean[] a1b =
      new Count.Pboolean().array(SZ + 3);
    show(a1b);

    System.out.println("Byte");
    Byte[] a2 = new Byte[SZ];
    Arrays.setAll(a2, new Count.Byte()::get);
    show(a2);
```

```
a2 = Stream.generate(new Count.Byte())
  .limit(SZ + 1).toArray(Byte[]::new);
show(a2);
a2 = new Count.Byte().array(SZ + 2);
show(a2);
byte[] a2b = new Count.Pbyte().array(SZ + 3);
show(a2b);

System.out.println("Character");
Character[] a3 = new Character[SZ];
Arrays.setAll(a3, new Count.Character()::get);
show(a3);
a3 = Stream.generate(new Count.Character())
  .limit(SZ + 1).toArray(Character[]::new);
show(a3);
a3 = new Count.Character().array(SZ + 2);
show(a3);
char[] a3b = new Count.Pchar().array(SZ + 3);
show(a3b);

System.out.println("Short");
Short[] a4 = new Short[SZ];
Arrays.setAll(a4, new Count.Short()::get);
show(a4);
a4 = Stream.generate(new Count.Short())
  .limit(SZ + 1).toArray(Short[]::new);
show(a4);
a4 = new Count.Short().array(SZ + 2);
show(a4);
short[] a4b = new Count.Pshort().array(SZ + 3);
show(a4b);

System.out.println("Integer");
int[] a5 = new int[SZ];
Arrays.setAll(a5, new Count.Integer()::get);
show(a5);
Integer[] a5b =
  Stream.generate(new Count.Integer())
    .limit(SZ + 1).toArray(Integer[]::new);
show(a5b);
a5b = new Count.Integer().array(SZ + 2);
show(a5b);
```

```
a5 = IntStream.generate(new Count.Pint())
  .limit(SZ + 1).toArray();
show(a5);
a5 = new Count.Pint().array(SZ + 3);
show(a5);

System.out.println("Long");
long[] a6 = new long[SZ];
Arrays.setAll(a6, new Count.Long()::get);
show(a6);
Long[] a6b = Stream.generate(new Count.Long())
  .limit(SZ + 1).toArray(Long[]::new);
show(a6b);
a6b = new Count.Long().array(SZ + 2);
show(a6b);
a6 = LongStream.generate(new Count.Plong())
  .limit(SZ + 1).toArray();
show(a6);
a6 = new Count.Plong().array(SZ + 3);
show(a6);

System.out.println("Float");
Float[] a7 = new Float[SZ];
Arrays.setAll(a7, new Count.Float()::get);
show(a7);
a7 = Stream.generate(new Count.Float())
  .limit(SZ + 1).toArray(Float[]::new);
show(a7);
a7 = new Count.Float().array(SZ + 2);
show(a7);
float[] a7b = new Count.Pfloat().array(SZ + 3);
show(a7b);

System.out.println("Double");
double[] a8 = new double[SZ];
Arrays.setAll(a8, new Count.Double()::get);
show(a8);
Double[] a8b =
  Stream.generate(new Count.Double())
    .limit(SZ + 1).toArray(Double[]::new);
show(a8b);
a8b = new Count.Double().array(SZ + 2);
```

```
    show(a8b);
    a8 = DoubleStream.generate(new Count.Pdouble())
      .limit(SZ + 1).toArray();
    show(a8);
    a8 = new Count.Pdouble().array(SZ + 3);
    show(a8);
  }
}
/* Output:
Boolean
[false, true, false, true, false]
[false, true, false, true, false, true]
[false, true, false, true, false, true, false]
[false, true, false, true, false, true, false, true]
Byte
[0, 1, 2, 3, 4]
[0, 1, 2, 3, 4, 5]
[0, 1, 2, 3, 4, 5, 6]
[0, 1, 2, 3, 4, 5, 6, 7]
Character
[b, c, d, e, f]
[b, c, d, e, f, g]
[b, c, d, e, f, g, h]
[b, c, d, e, f, g, h, i]
Short
[0, 1, 2, 3, 4]
[0, 1, 2, 3, 4, 5]
[0, 1, 2, 3, 4, 5, 6]
[0, 1, 2, 3, 4, 5, 6, 7]
Integer
[0, 1, 2, 3, 4]
[0, 1, 2, 3, 4, 5]
[0, 1, 2, 3, 4, 5, 6]
[0, 1, 2, 3, 4, 5]
[0, 1, 2, 3, 4, 5, 6, 7]
Long
[0, 1, 2, 3, 4]
[0, 1, 2, 3, 4, 5]
[0, 1, 2, 3, 4, 5, 6]
[0, 1, 2, 3, 4, 5]
[0, 1, 2, 3, 4, 5, 6, 7]
Float
```

```
[0.0, 1.0, 2.0, 3.0, 4.0]
[0.0, 1.0, 2.0, 3.0, 4.0, 5.0]
[0.0, 1.0, 2.0, 3.0, 4.0, 5.0, 6.0]
[0.0, 1.0, 2.0, 3.0, 4.0, 5.0, 6.0, 7.0]
Double
[0.0, 1.0, 2.0, 3.0, 4.0]
[0.0, 1.0, 2.0, 3.0, 4.0, 5.0]
[0.0, 1.0, 2.0, 3.0, 4.0, 5.0, 6.0]
[0.0, 1.0, 2.0, 3.0, 4.0, 5.0]
[0.0, 1.0, 2.0, 3.0, 4.0, 5.0, 6.0, 7.0]
*/
```

Notice the primitive array types `int[]`, `long[]` and `double[]` can be filled directly using `Arrays.setAll()`, but all other primitives require arrays of their wrapper types.

The wrapped arrays created via `Stream.generate()` show the overloaded use of `toArray()`, where you provide it with the constructor for the type of array to create.

Random Generators

We can follow the structure of `Count.java` to create a tool that produces random values:

```
// onjava/Rand.java
// Generate random values of different types
package onjava;
import java.util.*;
import java.util.function.*;
import static onjava.ConvertTo.*;

public interface Rand {
  int MOD = 10_000;
  class Boolean
  implements Supplier<java.lang.Boolean> {
    SplittableRandom r = new SplittableRandom(47);
    @Override public java.lang.Boolean get() {
      return r.nextBoolean();
    }
    public java.lang.Boolean get(int n) {
```

```
    return get();
  }
  public java.lang.Boolean[] array(int sz) {
    java.lang.Boolean[] result =
      new java.lang.Boolean[sz];
    Arrays.setAll(result, n -> get());
    return result;
  }
}
class Pboolean {
  public boolean[] array(int sz) {
    return primitive(new Boolean().array(sz));
  }
}
class Byte
implements Supplier<java.lang.Byte> {
  SplittableRandom r = new SplittableRandom(47);
  @Override public java.lang.Byte get() {
    return (byte)r.nextInt(MOD);
  }
  public java.lang.Byte get(int n) {
    return get();
  }
  public java.lang.Byte[] array(int sz) {
    java.lang.Byte[] result =
      new java.lang.Byte[sz];
    Arrays.setAll(result, n -> get());
    return result;
  }
}
class Pbyte {
  public byte[] array(int sz) {
    return primitive(new Byte().array(sz));
  }
}
class Character
implements Supplier<java.lang.Character> {
  SplittableRandom r = new SplittableRandom(47);
  @Override public java.lang.Character get() {
    return (char)r.nextInt('a', 'z' + 1);
  }
  public java.lang.Character get(int n) {
```

```
    return get();
  }
  public java.lang.Character[] array(int sz) {
    java.lang.Character[] result =
      new java.lang.Character[sz];
    Arrays.setAll(result, n -> get());
    return result;
  }
}
class Pchar {
  public char[] array(int sz) {
    return primitive(new Character().array(sz));
  }
}
class Short
implements Supplier<java.lang.Short> {
  SplittableRandom r = new SplittableRandom(47);
  @Override public java.lang.Short get() {
    return (short)r.nextInt(MOD);
  }
  public java.lang.Short get(int n) {
    return get();
  }
  public java.lang.Short[] array(int sz) {
    java.lang.Short[] result =
      new java.lang.Short[sz];
    Arrays.setAll(result, n -> get());
    return result;
  }
}
class Pshort {
  public short[] array(int sz) {
    return primitive(new Short().array(sz));
  }
}
class Integer
implements Supplier<java.lang.Integer> {
  SplittableRandom r = new SplittableRandom(47);
  @Override public java.lang.Integer get() {
    return r.nextInt(MOD);
  }
  public java.lang.Integer get(int n) {
```

```
    return get();
  }
  public java.lang.Integer[] array(int sz) {
    int[] primitive = new Pint().array(sz);
    java.lang.Integer[] result =
      new java.lang.Integer[sz];
    for(int i = 0; i < sz; i++)
      result[i] = primitive[i];
    return result;
  }
}
class Pint implements IntSupplier {
  SplittableRandom r = new SplittableRandom(47);
  @Override public int getAsInt() {
    return r.nextInt(MOD);
  }
  public int get(int n) { return getAsInt(); }
  public int[] array(int sz) {
    return r.ints(sz, 0, MOD).toArray();
  }
}
class Long
implements Supplier<java.lang.Long> {
  SplittableRandom r = new SplittableRandom(47);
  @Override public java.lang.Long get() {
    return r.nextLong(MOD);
  }
  public java.lang.Long get(int n) {
    return get();
  }
  public java.lang.Long[] array(int sz) {
    long[] primitive = new Plong().array(sz);
    java.lang.Long[] result =
      new java.lang.Long[sz];
    for(int i = 0; i < sz; i++)
      result[i] = primitive[i];
    return result;
  }
}
class Plong implements LongSupplier {
  SplittableRandom r = new SplittableRandom(47);
  @Override public long getAsLong() {
```

```
    return r.nextLong(MOD);
  }
  public long get(int n) { return getAsLong(); }
  public long[] array(int sz) {
    return r.longs(sz, 0, MOD).toArray();
  }
}
class Float
implements Supplier<java.lang.Float> {
  SplittableRandom r = new SplittableRandom(47);
  @Override public java.lang.Float get() {
    return (float)trim(r.nextDouble());
  }
  public java.lang.Float get(int n) {
    return get();
  }
  public java.lang.Float[] array(int sz) {
    java.lang.Float[] result =
      new java.lang.Float[sz];
    Arrays.setAll(result, n -> get());
    return result;
  }
}
class Pfloat {
  public float[] array(int sz) {
    return primitive(new Float().array(sz));
  }
}
static double trim(double d) {
  return
    ((double)Math.round(d * 1000.0)) / 100.0;
}
class Double
implements Supplier<java.lang.Double> {
  SplittableRandom r = new SplittableRandom(47);
  @Override public java.lang.Double get() {
    return trim(r.nextDouble());
  }
  public java.lang.Double get(int n) {
    return get();
  }
  public java.lang.Double[] array(int sz) {
```

```
    double[] primitive =
      new Rand.Pdouble().array(sz);
    java.lang.Double[] result =
      new java.lang.Double[sz];
    for(int i = 0; i < sz; i++)
      result[i] = primitive[i];
    return result;
  }
}
class Pdouble implements DoubleSupplier {
  SplittableRandom r = new SplittableRandom(47);
  @Override public double getAsDouble() {
    return trim(r.nextDouble());
  }
  public double get(int n) {
    return getAsDouble();
  }
  public double[] array(int sz) {
    double[] result = r.doubles(sz).toArray();
    Arrays.setAll(result,
      n -> result[n] = trim(result[n]));
    return result;
  }
}
class String
implements Supplier<java.lang.String> {
  SplittableRandom r = new SplittableRandom(47);
  private int strlen = 7; // Default length
  public String() {}
  public String(int strLength) {
    strlen = strLength;
  }
  @Override public java.lang.String get() {
    return r.ints(strlen, 'a', 'z' + 1)
      .collect(StringBuilder::new,
               StringBuilder::appendCodePoint,
               StringBuilder::append).toString();
  }
  public java.lang.String get(int n) {
    return get();
  }
  public java.lang.String[] array(int sz) {
```

```
      java.lang.String[] result =
        new java.lang.String[sz];
      Arrays.setAll(result, n -> get());
      return result;
    }
  }
}
```

For all primitive generators except for `int`, `long` and `double`, only arrays are generated rather than the full set of operations seen in `Count`. This is just a design choice, because the book doesn't need the extra features.

Here's a test for all the `Rand` tools:

```
// arrays/TestRand.java
// Test random generators
import java.util.*;
import java.util.stream.*;
import onjava.*;
import static onjava.ArrayShow.*;

public class TestRand {
  static final int SZ = 5;
  public static void main(String[] args) {
    System.out.println("Boolean");
    Boolean[] a1 = new Boolean[SZ];
    Arrays.setAll(a1, new Rand.Boolean()::get);
    show(a1);
    a1 = Stream.generate(new Rand.Boolean())
      .limit(SZ + 1).toArray(Boolean[]::new);
    show(a1);
    a1 = new Rand.Boolean().array(SZ + 2);
    show(a1);
    boolean[] a1b =
      new Rand.Pboolean().array(SZ + 3);
    show(a1b);

    System.out.println("Byte");
    Byte[] a2 = new Byte[SZ];
    Arrays.setAll(a2, new Rand.Byte()::get);
    show(a2);
    a2 = Stream.generate(new Rand.Byte())
```

```
    .limit(SZ + 1).toArray(Byte[]::new);
show(a2);
a2 = new Rand.Byte().array(SZ + 2);
show(a2);
byte[] a2b = new Rand.Pbyte().array(SZ + 3);
show(a2b);

System.out.println("Character");
Character[] a3 = new Character[SZ];
Arrays.setAll(a3, new Rand.Character()::get);
show(a3);
a3 = Stream.generate(new Rand.Character())
  .limit(SZ + 1).toArray(Character[]::new);
show(a3);
a3 = new Rand.Character().array(SZ + 2);
show(a3);
char[] a3b = new Rand.Pchar().array(SZ + 3);
show(a3b);

System.out.println("Short");
Short[] a4 = new Short[SZ];
Arrays.setAll(a4, new Rand.Short()::get);
show(a4);
a4 = Stream.generate(new Rand.Short())
  .limit(SZ + 1).toArray(Short[]::new);
show(a4);
a4 = new Rand.Short().array(SZ + 2);
show(a4);
short[] a4b = new Rand.Pshort().array(SZ + 3);
show(a4b);

System.out.println("Integer");
int[] a5 = new int[SZ];
Arrays.setAll(a5, new Rand.Integer()::get);
show(a5);
Integer[] a5b =
  Stream.generate(new Rand.Integer())
    .limit(SZ + 1).toArray(Integer[]::new);
show(a5b);
a5b = new Rand.Integer().array(SZ + 2);
show(a5b);
a5 = IntStream.generate(new Rand.Pint())
```

```
    .limit(SZ + 1).toArray();
show(a5);
a5 = new Rand.Pint().array(SZ + 3);
show(a5);

System.out.println("Long");
long[] a6 = new long[SZ];
Arrays.setAll(a6, new Rand.Long()::get);
show(a6);
Long[] a6b = Stream.generate(new Rand.Long())
  .limit(SZ + 1).toArray(Long[]::new);
show(a6b);
a6b = new Rand.Long().array(SZ + 2);
show(a6b);
a6 = LongStream.generate(new Rand.Plong())
  .limit(SZ + 1).toArray();
show(a6);
a6 = new Rand.Plong().array(SZ + 3);
show(a6);

System.out.println("Float");
Float[] a7 = new Float[SZ];
Arrays.setAll(a7, new Rand.Float()::get);
show(a7);
a7 = Stream.generate(new Rand.Float())
  .limit(SZ + 1).toArray(Float[]::new);
show(a7);
a7 = new Rand.Float().array(SZ + 2);
show(a7);
float[] a7b = new Rand.Pfloat().array(SZ + 3);
show(a7b);

System.out.println("Double");
double[] a8 = new double[SZ];
Arrays.setAll(a8, new Rand.Double()::get);
show(a8);
Double[] a8b =
  Stream.generate(new Rand.Double())
    .limit(SZ + 1).toArray(Double[]::new);
show(a8b);
a8b = new Rand.Double().array(SZ + 2);
show(a8b);
```

```
    a8 = DoubleStream.generate(new Rand.Pdouble())
      .limit(SZ + 1).toArray();
    show(a8);
    a8 = new Rand.Pdouble().array(SZ + 3);
    show(a8);

    System.out.println("String");
    String[] s = new String[SZ - 1];
    Arrays.setAll(s, new Rand.String()::get);
    show(s);
    s = Stream.generate(new Rand.String())
      .limit(SZ).toArray(String[]::new);
    show(s);
    s = new Rand.String().array(SZ + 1);
    show(s);

    Arrays.setAll(s, new Rand.String(4)::get);
    show(s);
    s = Stream.generate(new Rand.String(4))
      .limit(SZ).toArray(String[]::new);
    show(s);
    s = new Rand.String(4).array(SZ + 1);
    show(s);
  }
}
/* Output:
Boolean
[true, false, true, true, true]
[true, false, true, true, true, false]
[true, false, true, true, true, false, false]
[true, false, true, true, true, false, false, true]
Byte
[123, 33, 101, 112, 33]
[123, 33, 101, 112, 33, 31]
[123, 33, 101, 112, 33, 31, 0]
[123, 33, 101, 112, 33, 31, 0, -72]
Character
[b, t, p, e, n]
[b, t, p, e, n, p]
[b, t, p, e, n, p, c]
[b, t, p, e, n, p, c, c]
Short
```

```
[635, 8737, 3941, 4720, 6177]
[635, 8737, 3941, 4720, 6177, 8479]
[635, 8737, 3941, 4720, 6177, 8479, 6656]
[635, 8737, 3941, 4720, 6177, 8479, 6656, 3768]
Integer
[635, 8737, 3941, 4720, 6177]
[635, 8737, 3941, 4720, 6177, 8479]
[635, 8737, 3941, 4720, 6177, 8479, 6656]
[635, 8737, 3941, 4720, 6177, 8479]
[635, 8737, 3941, 4720, 6177, 8479, 6656, 3768]
Long
[6882, 3765, 692, 9575, 4439]
[6882, 3765, 692, 9575, 4439, 2638]
[6882, 3765, 692, 9575, 4439, 2638, 4011]
[6882, 3765, 692, 9575, 4439, 2638]
[6882, 3765, 692, 9575, 4439, 2638, 4011, 9610]
Float
[4.83, 2.89, 2.9, 1.97, 3.01]
[4.83, 2.89, 2.9, 1.97, 3.01, 0.18]
[4.83, 2.89, 2.9, 1.97, 3.01, 0.18, 0.99]
[4.83, 2.89, 2.9, 1.97, 3.01, 0.18, 0.99, 8.28]
Double
[4.83, 2.89, 2.9, 1.97, 3.01]
[4.83, 2.89, 2.9, 1.97, 3.01, 0.18]
[4.83, 2.89, 2.9, 1.97, 3.01, 0.18, 0.99]
[4.83, 2.89, 2.9, 1.97, 3.01, 0.18]
[4.83, 2.89, 2.9, 1.97, 3.01, 0.18, 0.99, 8.28]
String
[btpenpc, cuxszgv, gmeinne, eloztdv]
[btpenpc, cuxszgv, gmeinne, eloztdv, ewcippc]
[btpenpc, cuxszgv, gmeinne, eloztdv, ewcippc, ygpoalk]
[btpe, npcc, uxsz, gvgm, einn, eelo]
[btpe, npcc, uxsz, gvgm, einn]
[btpe, npcc, uxsz, gvgm, einn, eelo]
*/
```

Notice that (with the exception of the `String` section), this code is identical to that in `TestCount.java`, with `Count` replaced by `Rand`.

Generics and Primitive Arrays

Earlier in this chapter we were reminded that generics don't work with primitives. It's not uncommon to encounter situations where we must convert from arrays of primitives to arrays of wrapped types, and also to convert in the other direction. Here's a converter that performs both operations for all types:

```
// onjava/ConvertTo.java
package onjava;

public interface ConvertTo {
  static boolean[] primitive(Boolean[] in) {
    boolean[] result = new boolean[in.length];
    for(int i = 0; i < in.length; i++)
      result[i] = in[i]; // Autounboxing
    return result;
  }
  static char[] primitive(Character[] in) {
    char[] result = new char[in.length];
    for(int i = 0; i < in.length; i++)
      result[i] = in[i];
    return result;
  }
  static byte[] primitive(Byte[] in) {
    byte[] result = new byte[in.length];
    for(int i = 0; i < in.length; i++)
      result[i] = in[i];
    return result;
  }
  static short[] primitive(Short[] in) {
    short[] result = new short[in.length];
    for(int i = 0; i < in.length; i++)
      result[i] = in[i];
    return result;
  }
  static int[] primitive(Integer[] in) {
    int[] result = new int[in.length];
    for(int i = 0; i < in.length; i++)
      result[i] = in[i];
    return result;
  }
```

```
  static long[] primitive(Long[] in) {
    long[] result = new long[in.length];
    for(int i = 0; i < in.length; i++)
      result[i] = in[i];
    return result;
  }
  static float[] primitive(Float[] in) {
    float[] result = new float[in.length];
    for(int i = 0; i < in.length; i++)
      result[i] = in[i];
    return result;
  }
  static double[] primitive(Double[] in) {
    double[] result = new double[in.length];
    for(int i = 0; i < in.length; i++)
      result[i] = in[i];
    return result;
  }
  // Convert from primitive array to wrapped array:
  static Boolean[] boxed(boolean[] in) {
    Boolean[] result = new Boolean[in.length];
    for(int i = 0; i < in.length; i++)
      result[i] = in[i]; // Autoboxing
    return result;
  }
  static Character[] boxed(char[] in) {
    Character[] result = new Character[in.length];
    for(int i = 0; i < in.length; i++)
      result[i] = in[i];
    return result;
  }
  static Byte[] boxed(byte[] in) {
    Byte[] result = new Byte[in.length];
    for(int i = 0; i < in.length; i++)
      result[i] = in[i];
    return result;
  }
  static Short[] boxed(short[] in) {
    Short[] result = new Short[in.length];
    for(int i = 0; i < in.length; i++)
      result[i] = in[i];
    return result;
```

```
  }
  static Integer[] boxed(int[] in) {
    Integer[] result = new Integer[in.length];
    for(int i = 0; i < in.length; i++)
      result[i] = in[i];
    return result;
  }
  static Long[] boxed(long[] in) {
    Long[] result = new Long[in.length];
    for(int i = 0; i < in.length; i++)
      result[i] = in[i];
    return result;
  }
  static Float[] boxed(float[] in) {
    Float[] result = new Float[in.length];
    for(int i = 0; i < in.length; i++)
      result[i] = in[i];
    return result;
  }
  static Double[] boxed(double[] in) {
    Double[] result = new Double[in.length];
    for(int i = 0; i < in.length; i++)
      result[i] = in[i];
    return result;
  }
}
```

Each version of `primitive()` creates an appropriate primitive array of the correct length, then copies the elements from the `in` array of wrapper types. If any of the wrapped-array elements is `null`, you'll get an exception (this is reasonable—what value would you substitute that has meaning?). Notice how autounboxing takes place during assignment.

Here's a test for each method in `ConvertTo`:

```
// arrays/TestConvertTo.java
import java.util.*;
import onjava.*;
import static onjava.ArrayShow.*;
import static onjava.ConvertTo.*;

public class TestConvertTo {
  static final int SIZE = 6;
  public static void main(String[] args) {
    Boolean[] a1 = new Boolean[SIZE];
    Arrays.setAll(a1, new Rand.Boolean()::get);
    boolean[] a1p = primitive(a1);
    show("a1p", a1p);
    Boolean[] a1b = boxed(a1p);
    show("a1b", a1b);

    Byte[] a2 = new Byte[SIZE];
    Arrays.setAll(a2, new Rand.Byte()::get);
    byte[] a2p = primitive(a2);
    show("a2p", a2p);
    Byte[] a2b = boxed(a2p);
    show("a2b", a2b);

    Character[] a3 = new Character[SIZE];
    Arrays.setAll(a3, new Rand.Character()::get);
    char[] a3p = primitive(a3);
    show("a3p", a3p);
    Character[] a3b = boxed(a3p);
    show("a3b", a3b);

    Short[] a4 = new Short[SIZE];
    Arrays.setAll(a4, new Rand.Short()::get);
    short[] a4p = primitive(a4);
    show("a4p", a4p);
    Short[] a4b = boxed(a4p);
    show("a4b", a4b);

    Integer[] a5 = new Integer[SIZE];
    Arrays.setAll(a5, new Rand.Integer()::get);
    int[] a5p = primitive(a5);
    show("a5p", a5p);
    Integer[] a5b = boxed(a5p);
```

```
    show("a5b", a5b);

    Long[] a6 = new Long[SIZE];
    Arrays.setAll(a6, new Rand.Long()::get);
    long[] a6p = primitive(a6);
    show("a6p", a6p);
    Long[] a6b = boxed(a6p);
    show("a6b", a6b);

    Float[] a7 = new Float[SIZE];
    Arrays.setAll(a7, new Rand.Float()::get);
    float[] a7p = primitive(a7);
    show("a7p", a7p);
    Float[] a7b = boxed(a7p);
    show("a7b", a7b);

    Double[] a8 = new Double[SIZE];
    Arrays.setAll(a8, new Rand.Double()::get);
    double[] a8p = primitive(a8);
    show("a8p", a8p);
    Double[] a8b = boxed(a8p);
    show("a8b", a8b);
  }
}
/* Output:
a1p: [true, false, true, true, true, false]
a1b: [true, false, true, true, true, false]
a2p: [123, 33, 101, 112, 33, 31]
a2b: [123, 33, 101, 112, 33, 31]
a3p: [b, t, p, e, n, p]
a3b: [b, t, p, e, n, p]
a4p: [635, 8737, 3941, 4720, 6177, 8479]
a4b: [635, 8737, 3941, 4720, 6177, 8479]
a5p: [635, 8737, 3941, 4720, 6177, 8479]
a5b: [635, 8737, 3941, 4720, 6177, 8479]
a6p: [6882, 3765, 692, 9575, 4439, 2638]
a6b: [6882, 3765, 692, 9575, 4439, 2638]
a7p: [4.83, 2.89, 2.9, 1.97, 3.01, 0.18]
a7b: [4.83, 2.89, 2.9, 1.97, 3.01, 0.18]
a8p: [4.83, 2.89, 2.9, 1.97, 3.01, 0.18]
a8b: [4.83, 2.89, 2.9, 1.97, 3.01, 0.18]
*/
```

In each case, the original array is created for the wrapper type and filled using `Arrays.setAll()` as we did in `TestCounter.java` (this also verifies that `Arrays.setAll()` works with the wrapper arrays for `Integer`, `Long`, and `Double`). Then `ConvertTo.primitive()` converts the wrapper array to its corresponding primitive array, and `ConvertTo.boxed()` converts it back.

Modifying Existing Array Elements

The generator function passed to `Arrays.setAll()` can modify existing array elements by using the array index it receives:

```
// arrays/ModifyExisting.java
import java.util.*;
import onjava.*;
import static onjava.ArrayShow.*;

public class ModifyExisting {
  public static void main(String[] args) {
    double[] da = new double[7];
    Arrays.setAll(da, new Rand.Double()::get);
    show(da);
    Arrays.setAll(da, n -> da[n] / 100);       // [1]
    show(da);
  }
}
/* Output:
[4.83, 2.89, 2.9, 1.97, 3.01, 0.18, 0.99]
[0.0483, 0.028900000000000002, 0.028999999999999998,
0.0197, 0.0301, 0.0018, 0.009899999999999999]
*/
```

- **[1]** Lambdas are particularly useful here because the array will always be within the scope of the lambda expression.

An Aside On Parallelism

We are soon forced to encounter the topic of parallelism. For example, the word "parallel" is used in a number of Java library methods. You might have heard

something like "parallel programs run faster," and it makes sense—why have only a single processor working on your program when you can have many? You can easily be forgiven for thinking you should just take advantage of anything with "parallel" in it.

It would be lovely if it were that simple. Unfortunately, by taking this approach you can quite easily write code that *runs slower* than the non-parallel version. And until you understand all the issues quite deeply, it could very well seem like parallel programming is more art than science.

Here's the short version: Write code the easy and simple way. Don't start wrestling with parallelism unless it becomes a problem.

You'll still encounter parallelism. In this chapter we'll introduce some of the Java library methods written for parallel execution. So you must understand it enough for basic discussions, and to avoid the pitfalls. After you read the Concurrent Programming chapter, you'll understand it more deeply (but, alas, never enough. It turns out to be impossible to understand this topic enough).

There are also some situations where the parallel implementation is either the only one, or the best or logical choice regardless of whether you're explicitly trying to be parallel or not, or even if you only have a single processor. It's the one to use all the time, so you must understand the issues around it.

Strategies

It's probably best to think about parallelism in terms of data. For a *lot* of data (and with extra processors available), parallel *might* help. But it might not, and you could also make things worse.

Throughout the rest of the book, we will encounter different situations:

1. The only option provided is the parallel one. That's easy because we have no choice but to use it. This is rare.
2. There are multiple options but the parallel version (often the most recent version) is designed to be used everywhere (even in code that otherwise doesn't care about parallelism), as in case #1. We'll use the parallel version as intended.
3. Cases #1 and #2 don't happen that often. Instead, you'll encounter two versions of the algorithm, one for parallel usage and one for normal usage. I'll describe

the parallel one but won't use it in normal code because of all the possible problems it could produce.

I recommend you adopt this approach for your own code.

For further insights into why this is a hard problem, see Doug Lea's Article[152].

`parallelSetAll()`

The `Stream` approach produces elegant code. For example, suppose we'd like to create an array of `long` filled with values counted up from zero:

```
// arrays/CountUpward.java
import java.util.*;
import java.util.stream.*;
import static onjava.ArrayShow.*;

public class CountUpward {
  static long[] fillCounted(int size) {
    return LongStream.iterate(0, i -> i + 1)
      .limit(size).toArray();
  }
  public static void main(String[] args) {
    long[] l1 = fillCounted(20); // No problem
    show(l1);
    // On my machine, this runs out of heap space:
    //- long[] l2 = fillCounted(10_000_000);
  }
}
/* Output:
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15,
16, 17, 18, 19]
*/
```

`Streams` actually work up to nearly ten million, but then start running out of heap space. Regular `setAll()` works but it's nice if we can do it faster with such large numbers.

We can initialize bigger arrays using `setAll()`. If speed becomes an issue, `Arrays.parallelSetAll()` will (probably) perform the initialization faster (keeping in mind the problems described in An Aside On Parallelism):

[152] http://gee.cs.oswego.edu/dl/html/StreamParallelGuidance.html

```
// arrays/ParallelSetAll.java
import java.util.*;
import onjava.*;

public class ParallelSetAll {
  static final int SIZE = 10_000_000;
  static void intArray() {
    int[] ia = new int[SIZE];
    Arrays.setAll(ia, new Rand.Pint()::get);
    Arrays.parallelSetAll(ia, new Rand.Pint()::get);
  }
  static void longArray() {
    long[] la = new long[SIZE];
    Arrays.setAll(la, new Rand.Plong()::get);
    Arrays.parallelSetAll(la, new Rand.Plong()::get);
  }
  public static void main(String[] args) {
    intArray();
    longArray();
  }
}
```

The array allocations and initializations are performed in separate methods, because if both arrays are allocated in `main()`, it runs out of memory (on my machine, anyway. Also, there are ways of telling Java to allocate more memory on startup).

`Arrays` Utilities

You've seen `fill()` and `setAll()`/`parallelSetAll()` from `java.util.Arrays`. That class contains a number of other useful `static` utility methods that we shall explore. Here's an overview:

- `asList()`: Takes any sequence or array and turns it into a `List Collection`—this method was covered in the Collections chapter.
- `copyOf()`: Makes a new copy of an existing array, with a new length.
- `copyOfRange()`: Makes a new copy of a section of an existing array.
- `equals()`: Compare two arrays for equality.
- `deepEquals()`: Equality comparison for multidimensional arrays.

- `stream()`: Produce a `Stream` of the array elements.
- `hashCode()`: Produce the hash value of an array (you'll learn what this means in the Appendix: Understanding `equals()` and `hashCode()`).
- `deepHashCode()`: Hash value for a multidimensional array.
- `sort()`: Sort an array.
- `parallelSort()`: Sort an array in parallel, to increase speed.
- `binarySearch()`: Find an element in a sorted array.
- `parallelPrefix()`: Accumulate using the supplied function, in parallel (for speed). Basically, a `reduce()` for arrays.
- `spliterator()`: Produces a `Spliterator` from the array; this is an advanced part of `Streams` not covered in this book.
- `toString()`: Produce a `String` representation for an array. You've seen this used regularly throughout the chapter.
- `deepToString()`: Produce a `String` for a multidimensional array. You've seen this used regularly throughout the chapter.

All these methods are overloaded for all the primitive types and for `Objects`.

Copying an Array

`copyOf()` and `copyOfRange()` copy arrays far more quickly than if you use a `for` loop to perform the copy by hand. These methods are overloaded to handle all types. We start by copying arrays of `int` and `Integer`:

```
// arrays/ArrayCopying.java
// Demonstrate Arrays.copy() and Arrays.copyOf()
import java.util.*;
import onjava.*;
import static onjava.ArrayShow.*;

class Sup { // Superclass
  private int id;
  Sup(int n) { id = n; }
  @Override public String toString() {
    return getClass().getSimpleName() + id;
  }
```

```
}

class Sub extends Sup { // Subclass
  Sub(int n) { super(n); }
}

public class ArrayCopying {
  public static final int SZ = 15;
  public static void main(String[] args) {
    int[] a1 = new int[SZ];
    Arrays.setAll(a1, new Count.Integer()::get);
    show("a1", a1);
    int[] a2 = Arrays.copyOf(a1, a1.length);   // [1]
    // Prove they are distinct arrays:
    Arrays.fill(a1, 1);
    show("a1", a1);
    show("a2", a2);

    // Create a shorter result:
    a2 = Arrays.copyOf(a2, a2.length/2);       // [2]
    show("a2", a2);
    // Allocate more space:
    a2 = Arrays.copyOf(a2, a2.length + 5);
    show("a2", a2);

    // Also copies wrapped arrays:
    Integer[] a3 = new Integer[SZ];            // [3]
    Arrays.setAll(a3, new Count.Integer()::get);
    Integer[] a4 = Arrays.copyOfRange(a3, 4, 12);
    show("a4", a4);

    Sub[] d = new Sub[SZ/2];
    Arrays.setAll(d, Sub::new);
    // Produce Sup[] from Sub[]:
    Sup[] b =
      Arrays.copyOf(d, d.length, Sup[].class); // [4]
    show(b);

    // This "downcast" works fine:
    Sub[] d2 =
      Arrays.copyOf(b, b.length, Sub[].class); // [5]
    show(d2);
```

```
    // Bad "downcast" compiles but throws exception:
    Sup[] b2 = new Sup[SZ/2];
    Arrays.setAll(b2, Sup::new);
    try {
      Sub[] d3 = Arrays.copyOf(
        b2, b2.length, Sub[].class);           // [6]
    } catch(Exception e) {
      System.out.println(e);
    }
  }
}
/* Output:
a1: [0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14]
a1: [1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1]
a2: [0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14]
a2: [0, 1, 2, 3, 4, 5, 6]
a2: [0, 1, 2, 3, 4, 5, 6, 0, 0, 0, 0, 0]
a4: [4, 5, 6, 7, 8, 9, 10, 11]
[Sub0, Sub1, Sub2, Sub3, Sub4, Sub5, Sub6]
[Sub0, Sub1, Sub2, Sub3, Sub4, Sub5, Sub6]
java.lang.ArrayStoreException
*/
```

- **[1]** Here's the basic way to make a copy; just give it the size of the result. This is helpful if you're writing an algorithm that needs to resize storage. After the copy, we set all the elements of `a1` to 1, to demonstrate that this doesn't change anything in `a2`.
- **[2]** By changing the result size (the last argument), we can shorten or lengthen the resulting array.
- **[3]** Both `copyOf()` and `copyOfRange()` also work with wrapped types. `copyOfRange()` requires a start and end index.
- **[4]** Both `copyOf()` and `copyOfRange()` have a version that creates an array of a different type, by adding the destination type at the end of the method call. I first though this might be a way to produce a wrapped array from a primitive array, and vice-versa. But that doesn't work. What it's actually for is "upcasting" and "downcasting" arrays. That is, if you have an array of a subtype (derived type) and you'd like an array of the base type instead, these methods will produce the desired array.

- **[5]** You can even successfully "downcast" and produce an array of the subtype from an array of the supertype. This version works fine because we just "upcast."
- **[6]** This "array cast" will compile, but if the types are incompatible, you'll get a runtime exception. Here, forcing the base type to pretend it's a derived type is illegal because there are probably data and methods in the derived objects that are not in the base objects.

The example shows that both primitive arrays and object arrays can be copied. However, if you copy arrays of objects, then only the references get copied—there's no duplication of the objects themselves. This is called a *shallow copy* (see the Appendix: Passing and Returning Objects for more details).

There's also a method `System.arraycopy()` that copies one array into another, already allocated, array. This will not perform autoboxing or autounboxing—the two arrays must be of exactly the same type.

Comparing Arrays

`Arrays` provides `equals()` to compare one-dimensional arrays for equality, and `deepEquals()` to compare multi-dimensional arrays. These methods are overloaded for all the primitives and for `Object`. To be equal, the arrays must have the same number of elements, and each element must be equivalent to each corresponding element in the other array, using the `equals()` for each element (For primitives, that primitive's wrapper class `equals()` is used; for example, `Integer.equals()` for `int`).

```
// arrays/ComparingArrays.java
// Using Arrays.equals()
import java.util.*;
import onjava.*;

public class ComparingArrays {
  public static final int SZ = 15;
  static String[][] twoDArray() {
    String[][] md = new String[5][];
    Arrays.setAll(md, n -> new String[n]);
    for(int i = 0; i < md.length; i++)
      Arrays.setAll(md[i], new Rand.String()::get);
```

```
    return md;
  }
  public static void main(String[] args) {
    int[] a1 = new int[SZ], a2 = new int[SZ];
    Arrays.setAll(a1, new Count.Integer()::get);
    Arrays.setAll(a2, new Count.Integer()::get);
    System.out.println(
      "a1 == a2: " + Arrays.equals(a1, a2));
    a2[3] = 11;
    System.out.println(
      "a1 == a2: " + Arrays.equals(a1, a2));

    Integer[] a1w = new Integer[SZ],
      a2w = new Integer[SZ];
    Arrays.setAll(a1w, new Count.Integer()::get);
    Arrays.setAll(a2w, new Count.Integer()::get);
    System.out.println(
      "a1w == a2w: " + Arrays.equals(a1w, a2w));
    a2w[3] = 11;
    System.out.println(
      "a1w == a2w: " + Arrays.equals(a1w, a2w));

    String[][] md1 = twoDArray(), md2 = twoDArray();
    System.out.println(Arrays.deepToString(md1));
    System.out.println("deepEquals(md1, md2): " +
      Arrays.deepEquals(md1, md2));
    System.out.println(
      "md1 == md2: " + Arrays.equals(md1, md2));
    md1[4][1] = "#$#$#$#";
    System.out.println(Arrays.deepToString(md1));
    System.out.println("deepEquals(md1, md2): " +
      Arrays.deepEquals(md1, md2));
  }
}
/* Output:
a1 == a2: true
a1 == a2: false
a1w == a2w: true
a1w == a2w: false
[[], [btpenpc], [btpenpc, cuxszgv], [btpenpc, cuxszgv,
gmeinne], [btpenpc, cuxszgv, gmeinne, eloztdv]]
deepEquals(md1, md2): true
```

```
md1 == md2: false
[[], [btpenpc], [btpenpc, cuxszgv], [btpenpc, cuxszgv,
gmeinne], [btpenpc, #$#$#$#, gmeinne, eloztdv]]
deepEquals(md1, md2): false
*/
```

Originally, `a1` and `a2` are exactly equal, so the output is `true`, but then one of the elements is changed, which makes the result `false`. `a1w` and `a2w` repeat the exercise for a wrapped array.

`md1` and `md2` are multidimensional `String` arrays initialized identically via `twoDArray()`. Notice that `deepEquals()` produces `true` because it performs a proper comparison, while the normal `equals()` incorrectly produces `false`. If we change one of the elements in the array, `deepEquals()` detects it.

`Stream`s and Arrays

The `stream()` method easily produces a `Stream` of elements from some types of arrays:

```
// arrays/StreamFromArray.java
import java.util.*;
import onjava.*;

public class StreamFromArray {
  public static void main(String[] args) {
    String[] s = new Rand.String().array(10);
    Arrays.stream(s)
      .skip(3)
      .limit(5)
      .map(ss -> ss + "!")
      .forEach(System.out::println);

    int[] ia = new Rand.Pint().array(10);
    Arrays.stream(ia)
      .skip(3)
      .limit(5)
      .map(i -> i * 10)
      .forEach(System.out::println);
```

```
    Arrays.stream(new long[10]);
    Arrays.stream(new double[10]);

    // Only int, long and double work:
    //- Arrays.stream(new boolean[10]);
    //- Arrays.stream(new byte[10]);
    //- Arrays.stream(new char[10]);
    //- Arrays.stream(new short[10]);
    //- Arrays.stream(new float[10]);

    // For the other types you must use wrapped arrays:
    float[] fa = new Rand.Pfloat().array(10);
    Arrays.stream(ConvertTo.boxed(fa));
    Arrays.stream(new Rand.Float().array(10));
  }
}
/* Output:
eloztdv!
ewcippc!
ygpoalk!
ljlbynx!
taprwxz!
47200
61770
84790
66560
37680
*/
```

Only the "primary" types `int`, `long` and `double` work with `Arrays.stream()`; for the others you must somehow get a wrapped array.

Often it is easier to turn an array into a `Stream` to produce your desired results, instead of manipulating the array directly. Note that even though the `Stream` is "used up" (you can't repeat it), you still have the array so you can use that in other ways—including generating another `Stream`.

Sorting Arrays

Sorting performs comparisons based on the actual type of the object. One approach is to write a different sorting method for every different type, but such code is not reusable for new types.

A primary goal of programming design is to "separate things that change from things that stay the same," and here, the code that stays the same is the general sort algorithm, but the thing that changes from one use to the next is the way objects are compared. So instead of placing the comparison code into many different sort routines, the *Strategy* design pattern is used.[153] With a Strategy, the part of the code that varies is encapsulated inside a separate class (the Strategy object). You hand a Strategy object to the code that's always the same, which uses the Strategy to fulfill its algorithm. That way, you can make different objects expressing different ways of comparison and feed them to the same sorting code.

Java has two ways to provide comparison functionality. The first is with the "natural" comparison method that is imparted to a class by implementing the `java.lang.Comparable` interface. This is a very simple interface with a single method, `compareTo()`. This method takes another object of the same type as an argument and produces a negative value if the current object is less than the argument, zero if the argument is equal, and a positive value if the current object is greater than the argument.

Here's a class that implements `Comparable` and demonstrates comparability using the Java standard library method `Arrays.sort()`:

```
// arrays/CompType.java
// Implementing Comparable in a class
import java.util.*;
import java.util.function.*;
import onjava.*;
import static onjava.ArrayShow.*;

public class CompType implements Comparable<CompType> {
  int i;
  int j;
  private static int count = 1;
```

[153] See the Patterns chapter at the end of this book.

```
  public CompType(int n1, int n2) {
    i = n1;
    j = n2;
  }
  @Override public String toString() {
    String result = "[i = " + i + ", j = " + j + "]";
    if(count++ % 3 == 0)
      result += "\n";
    return result;
  }
  @Override public int compareTo(CompType rv) {
    return (i < rv.i ? -1 : (i == rv.i ? 0 : 1));
  }
  private static SplittableRandom r =
    new SplittableRandom(47);
  public static CompType get() {
    return new CompType(r.nextInt(100), r.nextInt(100));
  }
  public static void main(String[] args) {
    CompType[] a = new CompType[12];
    Arrays.setAll(a, n -> get());
    show("Before sorting", a);
    Arrays.sort(a);
    show("After sorting", a);
  }
}
/* Output:
Before sorting: [[i = 35, j = 37], [i = 41, j = 20], [i
= 77, j = 79]
, [i = 56, j = 68], [i = 48, j = 93], [i = 70, j = 7]
, [i = 0, j = 25], [i = 62, j = 34], [i = 50, j = 82]
, [i = 31, j = 67], [i = 66, j = 54], [i = 21, j = 6]
]
After sorting: [[i = 0, j = 25], [i = 21, j = 6], [i =
31, j = 67]
, [i = 35, j = 37], [i = 41, j = 20], [i = 48, j = 93]
, [i = 50, j = 82], [i = 56, j = 68], [i = 62, j = 34]
, [i = 66, j = 54], [i = 70, j = 7], [i = 77, j = 79]
]
*/
```

When you define the comparison method, you are responsible for deciding what it

means to compare one of your objects to another. Here, only the `i` values are used in the comparison, and the `j` values are ignored.

The `get()` method builds `CompType` objects by initializing them with random values. In `main()`, `get()` is used with `Arrays.setAll()` to fill an array of `CompType`, which is then sorted. If `Comparable` hadn't been implemented, you'd get a `ClassCastException` at run time when you tried to call `sort()`. This is because `sort()` casts its argument to `Comparable`.

Now suppose someone hands you a class that doesn't implement `Comparable`, or hands you this class that *does* implement `Comparable`, but you decide you don't like the way it works and would rather have a different comparison method for the type. To solve the problem, create a separate class that implements the `Comparator` interface (briefly introduced in the Collections chapter). It has two methods, `compare()` and `equals()`. However, you don't implement `equals()` except for special performance needs, because anytime you create a class, it is implicitly inherited from `Object`, which has an `equals()`. You can just use the default `Object equals()` and satisfy the contract imposed by the interface.

The `Collections` class (note the plural; we'll look at it more in the next chapter) contains a method `reverseOrder()` that produces a `Comparator` to reverse the natural sorting order. This can be applied to `CompType`:

```
// arrays/Reverse.java
// The Collections.reverseOrder() Comparator
import java.util.*;
import onjava.*;
import static onjava.ArrayShow.*;

public class Reverse {
  public static void main(String[] args) {
    CompType[] a = new CompType[12];
    Arrays.setAll(a, n -> CompType.get());
    show("Before sorting", a);
    Arrays.sort(a, Collections.reverseOrder());
    show("After sorting", a);
  }
}
/* Output:
Before sorting: [[i = 35, j = 37], [i = 41, j = 20], [i
= 77, j = 79]
```

```
, [i = 56, j = 68], [i = 48, j = 93], [i = 70, j = 7]
, [i = 0, j = 25], [i = 62, j = 34], [i = 50, j = 82]
, [i = 31, j = 67], [i = 66, j = 54], [i = 21, j = 6]
]
After sorting: [[i = 77, j = 79], [i = 70, j = 7], [i =
66, j = 54]
, [i = 62, j = 34], [i = 56, j = 68], [i = 50, j = 82]
, [i = 48, j = 93], [i = 41, j = 20], [i = 35, j = 37]
, [i = 31, j = 67], [i = 21, j = 6], [i = 0, j = 25]
]
*/
```

You can also write your own `Comparator`. This one compares `CompType` objects based on their `j` values rather than their `i` values:

```
// arrays/ComparatorTest.java
// Implementing a Comparator for a class
import java.util.*;
import onjava.*;
import static onjava.ArrayShow.*;

class CompTypeComparator
implements Comparator<CompType> {
  public int compare(CompType o1, CompType o2) {
    return (o1.j < o2.j ? -1 : (o1.j == o2.j ? 0 : 1));
  }
}

public class ComparatorTest {
  public static void main(String[] args) {
    CompType[] a = new CompType[12];
    Arrays.setAll(a, n -> CompType.get());
    show("Before sorting", a);
    Arrays.sort(a, new CompTypeComparator());
    show("After sorting", a);
  }
}
/* Output:
Before sorting: [[i = 35, j = 37], [i = 41, j = 20], [i
= 77, j = 79]
, [i = 56, j = 68], [i = 48, j = 93], [i = 70, j = 7]
, [i = 0, j = 25], [i = 62, j = 34], [i = 50, j = 82]
```

```
, [i = 31, j = 67], [i = 66, j = 54], [i = 21, j = 6]
]
After sorting: [[i = 21, j = 6], [i = 70, j = 7], [i =
41, j = 20]
, [i = 0, j = 25], [i = 62, j = 34], [i = 35, j = 37]
, [i = 66, j = 54], [i = 31, j = 67], [i = 56, j = 68]
, [i = 77, j = 79], [i = 50, j = 82], [i = 48, j = 93]
]
*/
```

Using `Arrays.sort()`

With the built-in sorting methods, you can sort any array of primitives, or any array of objects that either implements `Comparable` or has an associated `Comparator`. Here we generate an array of random `String` objects and sort it:[154]

```
// arrays/StringSorting.java
// Sorting an array of Strings
import java.util.*;
import onjava.*;
import static onjava.ArrayShow.*;

public class StringSorting {
  public static void main(String[] args) {
    String[] sa = new Rand.String().array(20);
    show("Before sort", sa);
    Arrays.sort(sa);
    show("After sort", sa);
    Arrays.sort(sa, Collections.reverseOrder());
    show("Reverse sort", sa);
    Arrays.sort(sa, String.CASE_INSENSITIVE_ORDER);
    show("Case-insensitive sort", sa);
  }
}
/* Output:
Before sort: [btpenpc, cuxszgv, gmeinne, eloztdv,
ewcippc, ygpoalk, ljlbynx, taprwxz, bhmupju, cjwzmmr,
anmkkyh, fcjpthl, skddcat, jbvlgwc, mvducuj, ydpulcq,
zehpfmm, zrxmclh, qgekgly, hyoubzl]
```

[154] Surprisingly, there was no support in Java 1.0 or 1.1 for sorting `Strings`.

```
After sort: [anmkkyh, bhmupju, btpenpc, cjwzmmr,
cuxszgv, eloztdv, ewcippc, fcjpthl, gmeinne, hyoubzl,
jbvlgwc, ljlbynx, mvducuj, qgekgly, skddcat, taprwxz,
ydpulcq, ygpoalk, zehpfmm, zrxmclh]
Reverse sort: [zrxmclh, zehpfmm, ygpoalk, ydpulcq,
taprwxz, skddcat, qgekgly, mvducuj, ljlbynx, jbvlgwc,
hyoubzl, gmeinne, fcjpthl, ewcippc, eloztdv, cuxszgv,
cjwzmmr, btpenpc, bhmupju, anmkkyh]
Case-insensitive sort: [anmkkyh, bhmupju, btpenpc,
cjwzmmr, cuxszgv, eloztdv, ewcippc, fcjpthl, gmeinne,
hyoubzl, jbvlgwc, ljlbynx, mvducuj, qgekgly, skddcat,
taprwxz, ydpulcq, ygpoalk, zehpfmm, zrxmclh]
*/
```

Notice the output in the `String` sorting algorithm. It's *lexicographic*, so it puts all the words starting with uppercase letters first, followed by all the words starting with lowercase letters. (Telephone books are typically sorted this way.) To group the words together regardless of case, use `String.CASE_INSENSITIVE_ORDER` as shown in the last call to `sort()`.

The sorting algorithm that's used in the Java standard library is designed to be optimal for the particular type you're sorting—a Quicksort for primitives, and a stable merge sort for objects.

Sorting in Parallel

If sorting performance is an issue, you can use the Java 8 `parallelSort()`, which has overloaded versions for all contingencies, including sorting regions of an array or using a `Comparator`. To see the benefits of `parallelSort()` vs. the ordinary `sort()`, we use JMH which was introduced in Validating Your Code:

```
// arrays/jmh/ParallelSort.java
package arrays.jmh;
import java.util.*;
import onjava.*;
import org.openjdk.jmh.annotations.*;

@State(Scope.Thread)
public class ParallelSort {
  private long[] la;
  @Setup
  public void setup() {
    la = new Rand.Plong().array(100_000);
  }
  @Benchmark
  public void sort() {
    Arrays.sort(la);
  }
  @Benchmark
  public void parallelSort() {
    Arrays.parallelSort(la);
  }
}
```

The `parallelSort()` algorithm breaks the large array into smaller arrays until the array size reaches a limit, at which point it uses the ordinary `Arrays.sort()` method. Then the results are merged. The algorithm requires additional working space, but this is no larger than the space of the original array.

You might see different results, but on my machine the parallel sort sped things up about 3 times. The parallel version is trivial to use so it's tempting to consider using it everywhere, in preference to `Arrays.sort()`. Of course, it might not be that simple—see Microbenchmarking.

Searching with `Arrays.binarySearch()`

Once an array is sorted, you can perform a fast search for a particular item by using `Arrays.binarySearch()`. However, if you try to use `binarySearch()` on an unsorted array the results are unpredictable. The following example uses the `Rand.Pint` class to create an array filled with random `int` values, then calls `getAsInt()` (because `Rand.Pint` is an `IntSupplier`) to produce search values:

```
// arrays/ArraySearching.java
// Using Arrays.binarySearch()
import java.util.*;
import onjava.*;
import static onjava.ArrayShow.*;

public class ArraySearching {
  public static void main(String[] args) {
    Rand.Pint rand = new Rand.Pint();
    int[] a = new Rand.Pint().array(25);
    Arrays.sort(a);
    show("Sorted array", a);
    while(true) {
      int r = rand.getAsInt();
      int location = Arrays.binarySearch(a, r);
      if(location >= 0) {
        System.out.println(
          "Location of " + r + " is " + location +
          ", a[" + location + "] is " + a[location]);
        break; // Out of while loop
      }
    }
  }
}
/* Output:
Sorted array: [125, 267, 635, 650, 1131, 1506, 1634,
2400, 2766, 3063, 3768, 3941, 4720, 4762, 4948, 5070,
5682, 5807, 6177, 6193, 6656, 7021, 8479, 8737, 9954]
Location of 635 is 2, a[2] is 635
*/
```

In the `while` loop, random values are generated as search items until one of them is found in the array.

`Arrays.binarySearch()` produces a value greater than or equal to zero if the search item is found. Otherwise, it produces a negative value representing the place that the element should be inserted if you are maintaining the sorted array by hand. The value produced is

```
-(insertion point) - 1
```

The insertion point is the index of the first element greater than the key, or `a.size()`, if all elements in the array are less than the specified key.

If an array contains duplicate elements, there is no guarantee which of those duplicates are found. The search algorithm is not designed to support duplicate elements, but rather to tolerate them. If you need a sorted list of non-duplicated elements, use a `TreeSet` (to maintain sorted order) or `LinkedHashSet` (to maintain insertion order). Those classes take care of all the details for you automatically. Only in cases of performance bottlenecks should you replace one of these classes with a hand-maintained array.

If you sort an object array using a `Comparator` (primitive arrays do not allow sorting with a `Comparator`), you must include that same `Comparator` when you perform a `binarySearch()` (using the overloaded version of `binarySearch()`). For example, the `StringSorting.java` program can be modified to perform a search:

```
// arrays/AlphabeticSearch.java
// Searching with a Comparator import
import java.util.*;
import onjava.*;
import static onjava.ArrayShow.*;

public class AlphabeticSearch {
  public static void main(String[] args) {
    String[] sa = new Rand.String().array(30);
    Arrays.sort(sa, String.CASE_INSENSITIVE_ORDER);
    show(sa);
    int index = Arrays.binarySearch(sa,
      sa[10], String.CASE_INSENSITIVE_ORDER);
    System.out.println(
     "Index: "+ index + "\n"+ sa[index]);
  }
}
/* Output:
[anmkkyh, bhmupju, btpenpc, cjwzmmr, cuxszgv, eloztdv,
ewcippc, ezdeklu, fcjpthl, fqmlgsh, gmeinne, hyoubzl,
jbvlgwc, jlxpqds, ljlbynx, mvducuj, qgekgly, skddcat,
taprwxz, uybypgp, vjsszkn, vniyapk, vqqakbm, vwodhcf,
ydpulcq, ygpoalk, yskvett, zehpfmm, zofmmvm, zrxmclh]
Index: 10
gmeinne
*/
```

The `Comparator` must be passed to the overloaded `binarySearch()` as the third

argument. In this example, success is guaranteed because the search item is selected from the array itself.

Accumulating with `parallelPrefix()`

There's no "prefix()" method, only a `parallelPrefix()`. This is like the `reduce()` method from the `Stream` class: it performs an operation on the previous and current elements, and puts the result into the current element location:

```
// arrays/ParallelPrefix1.java
import java.util.*;
import onjava.*;
import static onjava.ArrayShow.*;

public class ParallelPrefix1 {
  public static void main(String[] args) {
    int[] nums = new Count.Pint().array(10);
    show(nums);
    System.out.println(Arrays.stream(nums)
      .reduce(Integer::sum).getAsInt());
    Arrays.parallelPrefix(nums, Integer::sum);
    show(nums);
    System.out.println(Arrays.stream(
      new Count.Pint().array(6))
      .reduce(Integer::sum).getAsInt());
  }
}
/* Output:
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
45
[0, 1, 3, 6, 10, 15, 21, 28, 36, 45]
15
*/
```

Here we apply `Integer::sum` to the array. In location zero, it puts the previously-calculated value (zero, as there is no previous) combined with the value in location zero of the original array. In location one, it takes the previously-calculated value (which it just stored in location zero) and combines it with the value previously in location one. And so on.

`Stream.reduce()` only produces the final result, but with `Arrays.parallelPrefix()` you also get all the intermediate calculations in case those are useful. Notice how the result of the second `Stream.reduce()` calculation is already in the array calculated by `parallelPrefix()`.

It might be clearer using `Strings`:

```
// arrays/ParallelPrefix2.java
import java.util.*;
import onjava.*;
import static onjava.ArrayShow.*;

public class ParallelPrefix2 {
  public static void main(String[] args) {
    String[] strings = new Rand.String(1).array(8);
    show(strings);
    Arrays.parallelPrefix(strings, (a, b) -> a + b);
    show(strings);
  }
}
/* Output:
[b, t, p, e, n, p, c, c]
[b, bt, btp, btpe, btpen, btpenp, btpenpc, btpenpcc]
*/
```

As noted previously, initialization using `Streams` is elegant, but for large arrays that approach can run out of heap space. Performing initialization using `setAll()` is more memory-efficient:

```
// arrays/ParallelPrefix3.java
// {ExcludeFromTravisCI}
import java.util.*;

public class ParallelPrefix3 {
  static final int SIZE = 10_000_000;
  public static void main(String[] args) {
    long[] nums = new long[SIZE];
    Arrays.setAll(nums, n -> n);
    Arrays.parallelPrefix(nums, Long::sum);
    System.out.println("First 20: " + nums[19]);
    System.out.println("First 200: " + nums[199]);
    System.out.println("All: " + nums[nums.length-1]);
```

```
  }
}
/* Output:
First 20: 190
First 200: 19900
All: 49999995000000
*/
```

Because it can be rather complicated to get right, `parallelPrefix()` should generally only be used when you have memory or speed issues (or both). Otherwise, `Stream.reduce()` should be your first choice.

Summary

Java provides reasonable support for fixed-sized, low-level arrays. This kind of array emphasizes performance over flexibility, just like the C and C++ array model. In the initial version of Java, fixed-sized, low-level arrays were absolutely necessary, not only because the Java designers chose to include primitive types (also for performance), but because the support for `Collections` in that version was very minimal. Thus, in early versions of Java, it was always reasonable to choose arrays.

In subsequent versions of Java, `Collection` support improved significantly, and now `Collections` tend to outshine arrays in all ways except for performance, and even then, the performance of `Collections` is significantly improved. As stated in other places in this book, performance problems are usually never where you imagine them to be, anyway.

With autoboxing and generics, holding primitives in `Collections` is effortless, which further encourages you to replace low-level arrays with `Collections`. Because generics produce type-safe `Collections`, arrays no long have an advantage on that front, either.

As noted in this chapter and as you'll see when you try to use them, generics are fairly hostile towards arrays. Often, even when you can get generics and arrays to work together in some form (as you'll see in the next chapter), you'll still end up with "unchecked" warnings during compilation.

On several occasions I heard directly from Java language designers that I should be using `Collections` instead of arrays, when we were discussing particular examples (I was using arrays to demonstrate specific techniques so I did not have that option).

All these issues indicate you should "prefer `Collections` to arrays" when programming in recent versions of Java. Only when you prove that performance is an issue (and that switching to an array will actually make a significant difference) should you refactor to arrays.

This is a rather bold statement, but some languages have no fixed-sized, low-level arrays at all. They only have resizable collections with significantly more functionality than C/C++/Java-style arrays. Python[155], for example, has a `list` type that uses basic array syntax, but with much greater functionality—you can even inherit from it:

```
# arrays/PythonLists.py

aList = [1, 2, 3, 4, 5]
print(type(aList)) # <type 'list'>
print(aList) # [1, 2, 3, 4, 5]
print(aList[4]) # 5  Basic list indexing
aList.append(6) # lists can be resized
aList += [7, 8] # Add a list to a list
print(aList) # [1, 2, 3, 4, 5, 6, 7, 8]
aSlice = aList[2:4]
print(aSlice) # [3, 4]

class MyList(list): # Inherit from list
    # Define a method; 'this' pointer is explicit:
    def getReversed(self):
        reversed = self[:] # Copy list using slices
        reversed.reverse() # Built-in list method
        return reversed

# No 'new' necessary for object creation: {#24-no-new-necessary-for-obj\
ect-creation}
list2 = MyList(aList)
print(type(list2)) # <class '__main__.MyList'>
print(list2.getReversed()) # [8, 7, 6, 5, 4, 3, 2, 1]
```

[155] http://www.Python.org

```
output = """
<class 'list'>
[1, 2, 3, 4, 5]
5
[1, 2, 3, 4, 5, 6, 7, 8]
[3, 4]
<class '__main__.MyList'>
[8, 7, 6, 5, 4, 3, 2, 1]
"""
```

Basic Python syntax was introduced in the previous chapter. Here, a list is created by surrounding a comma-separated sequence of objects with square brackets. The result is an object with a runtime type of `list` (the output of the `print` statements is shown as comments on the same line). The result of printing a `list` is the same as that of using `Arrays.toString()` in Java.

Creating a sub-sequence of a `list` is accomplished with *slicing*, by placing the `:` operator inside the index operation. The `list` type has many more built-in operations, to the point where it's usually all you need for a sequence type.

`MyList` is a `class` definition; the base classes are placed within the parentheses. Inside the class, `def` statements produce methods, and the first argument to the method is automatically the equivalent of `this` in Java, except that in Python it's explicit and the identifier `self` is used by convention (it's not a keyword). Notice how the constructor is automatically inherited.

Although everything in Python really *is* an object (including integral and floating point types), you still have an escape hatch because you can optimize performance-critical portions of your code by writing extensions in C, C++ or using special tools designed to easily speed up your Python code (of which there are many). This way you can have object purity without being prevented from performance improvements.

The PHP language[156] goes even further by having only a single array type, which acts as both an `int`-indexed array and an associative array (a `Map`).

It's interesting to speculate, after this many years of Java evolution, whether the designers would put primitives and low-level arrays in the language if they were to start over again (the Scala language, which also runs on the JVM, does not include

[156] www.php.net

these). If these were left out, it would be possible to make a truly pure object-oriented language (despite claims, Java is not a pure OO language, precisely because of the low-level detritus). The initial argument for efficiency is always compelling, but over time we have seen an evolution away from this idea and towards higher-level components like `Collections`. Add to this the fact that if `Collections` can be built into the core language as they are in some languages, the compiler has a much better opportunity to optimize.

"Green-fields" speculation aside, we are certainly stuck with arrays, and you see them when reading code. `Collections`, however, are almost always a better choice.

Enumerations

> The `enum` keyword creates a new type with a restricted set of named values, and treats those values as regular program components. This turns out to be very useful.[157]

Enumerations were introduced briefly at the end of Housekeeping. However, now that you understand some of the deeper issues in Java, we can take a more detailed look at Java's enumerations. You'll see that `enum`s can be very useful, but this chapter should also give you more insight into other language features, such as generics and reflection. You'll also learn a few more design patterns.

Basic `enum` Features

As shown in Housekeeping, you can step through the list of `enum` constants by calling `values()` on the `enum`. The `values()` method produces an array of the `enum` constants in the order in which they were declared, so you can use the resulting array in (for example) a *for-in* loop.

When you create an `enum`, an associated class is produced for you by the compiler. This class is automatically inherited from `java.lang.Enum`, which provides certain capabilities shown in this example:

[157] Joshua Bloch was extremely helpful in developing this chapter.

```
// enums/EnumClass.java
// Capabilities of the Enum class

enum Shrubbery { GROUND, CRAWLING, HANGING }

public class EnumClass {
  public static void main(String[] args) {
    for(Shrubbery s : Shrubbery.values()) {
      System.out.println(
        s + " ordinal: " + s.ordinal());
      System.out.print(
        s.compareTo(Shrubbery.CRAWLING) + " ");
      System.out.print(
        s.equals(Shrubbery.CRAWLING) + " ");
      System.out.println(s == Shrubbery.CRAWLING);
      System.out.println(s.getDeclaringClass());
      System.out.println(s.name());
      System.out.println("**********************");
    }
    // Produce an enum value from a String name:
    for(String s :
        "HANGING CRAWLING GROUND".split(" ")) {
      Shrubbery shrub =
        Enum.valueOf(Shrubbery.class, s);
      System.out.println(shrub);
    }
  }
}
/* Output:
GROUND ordinal: 0
-1 false false
class Shrubbery
GROUND
**********************
CRAWLING ordinal: 1
0 true true
class Shrubbery
CRAWLING
**********************
HANGING ordinal: 2
1 false false
class Shrubbery
```

```
HANGING
**********
HANGING
CRAWLING
GROUND
*/
```

The `ordinal()` method produces an `int` indicating the declaration order of each `enum` instance, starting from zero. You can always safely compare `enum` instances using `==`, and `equals()` and `hashCode()` are automatically created for you. The `Enum` class is `Comparable`, so there's a `compareTo()` method, and it is also `Serializable`.

If you call `getDeclaringClass()` on an `enum` instance, you'll find out the enclosing `enum` class.

The `name()` method produces the name exactly as it is declared, and this is what you get with `toString()`, as well. `valueOf()` is a `static` member of `Enum`, and produces the `enum` instance that corresponds to the `String` name you pass to it, or throws an exception if there's no match.

Using `static` Imports with `enum`s

Consider a variation of `Burrito.java` from the Housekeeping chapter:

```
// enums/SpicinessEnum.java
package enums;

public enum SpicinessEnum {
  NOT, MILD, MEDIUM, HOT, FLAMING
}
```

```
// enums/Burrito2.java
// {java enums.Burrito2}
package enums;
import static enums.SpicinessEnum.*;

public class Burrito2 {
  SpicinessEnum degree;
  public Burrito2(SpicinessEnum degree) {
    this.degree = degree;
  }
  @Override public String toString() {
    return "Burrito is "+ degree;
  }
  public static void main(String[] args) {
    System.out.println(new Burrito2(NOT));
    System.out.println(new Burrito2(MEDIUM));
    System.out.println(new Burrito2(HOT));
  }
}
/* Output:
Burrito is NOT
Burrito is MEDIUM
Burrito is HOT
*/
```

The `static import` brings all the `enum` instance identifiers into the local namespace, so they don't need qualification. Is this a good idea, or is it better to be explicit and qualify all `enum` instances? It probably depends on the complexity of your code. The compiler certainly won't let you use the wrong type, so your only concern is whether the code is confusing to the reader. In many situations it will probably be fine but evaluate it on an individual basis.

Note it is not possible to use this technique if the `enum` is defined in the same file or the default package (apparently there were some arguments within Sun about whether to allow this).

Adding Methods to an `enum`

Except for the fact you can't inherit from it, an `enum` can be treated much like a regular class. This means you can add methods to an `enum`. It's even possible for an

enum to have a main().

You might produce different descriptions for an enumeration than the default toString(), which simply produces the name of that enum instance, as you've seen. To do this, you can provide a constructor to capture extra information, and additional methods to provide an extended description, like this:

```
// enums/OzWitch.java
// The witches in the land of Oz

public enum OzWitch {
  // Instances must be defined first, before methods:
  WEST("Miss Gulch, aka the Wicked Witch of the West"),
  NORTH("Glinda, the Good Witch of the North"),
  EAST("Wicked Witch of the East, wearer of the Ruby " +
    "Slippers, crushed by Dorothy's house"),
  SOUTH("Good by inference, but missing");
  private String description;
  // Constructor must be package or private access:
  private OzWitch(String description) {
    this.description = description;
  }
  public String getDescription() { return description; }
  public static void main(String[] args) {
    for(OzWitch witch : OzWitch.values())
      System.out.println(
        witch + ": " + witch.getDescription());
  }
}
/* Output:
WEST: Miss Gulch, aka the Wicked Witch of the West
NORTH: Glinda, the Good Witch of the North
EAST: Wicked Witch of the East, wearer of the Ruby
Slippers, crushed by Dorothy's house
SOUTH: Good by inference, but missing
*/
```

If you are going to define methods you must end the sequence of enum instances with a semicolon. Also, Java forces you to define the instances first within the enum. You'll get a compile-time error if you try to define them after any of the methods or fields.

The constructor and methods have the same form as a regular class, because with a few restrictions this *is* a regular class. You can do pretty much anything you want with `enum`s (although you'll usually keep them pretty ordinary).

Although the constructor here is `private` as an example, it doesn't make much difference what access you use—the constructor can only be used to create the `enum` instances you declare inside the `enum` definition; the compiler won't let you use it to create any new instances once the `enum` definition is complete.

Overriding `enum` Methods

Here's another approach to producing different `String` values for enumerations. Here, the instance names are OK but we want to reformat them for display. Overriding the `toString()` method for an `enum` is the same as overriding it for a regular class:

```
// enums/SpaceShip.java
import java.util.stream.*;

public enum SpaceShip {
  SCOUT, CARGO, TRANSPORT,
  CRUISER, BATTLESHIP, MOTHERSHIP;
  @Override public String toString() {
    String id = name();
    String lower = id.substring(1).toLowerCase();
    return id.charAt(0) + lower;
  }
  public static void main(String[] args) {
    Stream.of(values())
      .forEach(System.out::println);
  }
}
/* Output:
Scout
Cargo
Transport
Cruiser
Battleship
Mothership
*/
```

The `toString()` method gets the `SpaceShip` name by calling `name()`, and modifies the result so only the first letter is capitalized.

`enum`s in `switch` Statements

One very convenient capability of `enum`s is the way they can be used in `switch` statements. Ordinarily, a `switch` only works with an integral value, but because `enum`s have an established integral order and the order of an instance can be produced with the `ordinal()` method (apparently the compiler does something like this), `enum`s can be used in `switch` statements.

Although normally you must qualify an `enum` instance with its type, you don't do this in a `case` statement. Here's an example that uses an `enum` to create a little state machine:

```
// enums/TrafficLight.java
// Enums in switch statements

// Define an enum type:
enum Signal { GREEN, YELLOW, RED, }

public class TrafficLight {
  Signal color = Signal.RED;
  public void change() {
    switch(color) {
      // Note you don't have to say Signal.RED
      // in the case statement:
      case RED:    color = Signal.GREEN;
                   break;
      case GREEN:  color = Signal.YELLOW;
                   break;
      case YELLOW: color = Signal.RED;
                   break;
    }
  }
  @Override public String toString() {
    return "The traffic light is " + color;
  }
  public static void main(String[] args) {
```

```
    TrafficLight t = new TrafficLight();
    for(int i = 0; i < 7; i++) {
      System.out.println(t);
      t.change();
    }
  }
}
/* Output:
The traffic light is RED
The traffic light is GREEN
The traffic light is YELLOW
The traffic light is RED
The traffic light is GREEN
The traffic light is YELLOW
The traffic light is RED
*/
```

The compiler does not complain because there is no `default` statement inside the `switch`, but that's not because it notices you have `case` statements for each `Signal` instance. If you comment out one of the `case` statements it still won't complain. This means you must pay attention and ensure you cover all the cases on your own. On the other hand, if you are calling `return` from `case` statements, the compiler *will* complain if you don't have a `default`—even if you've covered all the possible values of the `enum`.

The Mystery of `values()`

As noted earlier, all `enum` classes are created for you by the compiler and extend the `Enum` class. However, if you look at `Enum`, you'll see there is no `values()` method, even though we've been using it. Are there any other "hidden" methods? We can write a small reflection program to find out:

```
// enums/Reflection.java
// Analyzing enums using reflection
import java.lang.reflect.*;
import java.util.*;
import onjava.*;

enum Explore { HERE, THERE }

public class Reflection {
  public static
  Set<String> analyze(Class<?> enumClass) {
    System.out.println(
      "_____ Analyzing " + enumClass + " _____");
    System.out.println("Interfaces:");
    for(Type t : enumClass.getGenericInterfaces())
      System.out.println(t);
    System.out.println(
      "Base: " + enumClass.getSuperclass());
    System.out.println("Methods: ");
    Set<String> methods = new TreeSet<>();
    for(Method m : enumClass.getMethods())
      methods.add(m.getName());
    System.out.println(methods);
    return methods;
  }
  public static void main(String[] args) {
    Set<String> exploreMethods =
      analyze(Explore.class);
    Set<String> enumMethods = analyze(Enum.class);
    System.out.println(
      "Explore.containsAll(Enum)? " +
      exploreMethods.containsAll(enumMethods));
    System.out.print("Explore.removeAll(Enum): ");
    exploreMethods.removeAll(enumMethods);
    System.out.println(exploreMethods);
    // Decompile the code for the enum:
    OSExecute.command(
      "javap -cp build/classes/java/main Explore");
  }
}
/* Output:
_____ Analyzing class Explore _____
```

```
Interfaces:
Base: class java.lang.Enum
Methods:
[compareTo, equals, getClass, getDeclaringClass,
hashCode, name, notify, notifyAll, ordinal, toString,
valueOf, values, wait]
_____ Analyzing class java.lang.Enum _____
Interfaces:
java.lang.Comparable<E>
interface java.io.Serializable
Base: class java.lang.Object
Methods:
[compareTo, equals, getClass, getDeclaringClass,
hashCode, name, notify, notifyAll, ordinal, toString,
valueOf, wait]
Explore.containsAll(Enum)? true
Explore.removeAll(Enum): [values]
Compiled from "Reflection.java"
final class Explore extends java.lang.Enum<Explore> {
  public static final Explore HERE;
  public static final Explore THERE;
  public static Explore[] values();
  public static Explore valueOf(java.lang.String);
  static {};
}
*/
```

So the answer is that `values()` is a `static` method that is added by the compiler. Note that `valueOf()` is also added to `Explore` in the process of creating the `enum`. This is slightly confusing, because there's also a `valueOf()` that is part of the `Enum` class, but that method has two arguments and the added method only has one. However, the `Set` method here is only looking at method names, and not signatures, so after calling `Explore.removeAll(Enum)`, the only thing that remains is `[values]`.

The output shows that `Explore` is made `final` by the compiler, so you cannot inherit from an `enum`. There's also a `static` initialization clause, which as you'll see later can be redefined.

Because of erasure (described in the Generics chapter), the decompiler does not have full information about `Enum`, so it shows the base class of `Explore` as a raw `Enum`

rather than the actual `Enum<Explore>`.

Because `values()` is a `static` method inserted into the `enum` definition by the compiler, if you upcast an `enum` type to `Enum`, the `values()` method is not available. Notice, however, there is a `getEnumConstants()` method in `Class`, so even if `values()` is not part of the interface of `Enum`, you can still get the `enum` instances via the `Class` object:

```
// enums/UpcastEnum.java
// No values() method if you upcast an enum

enum Search { HITHER, YON }

public class UpcastEnum {
  public static void main(String[] args) {
    Search[] vals = Search.values();
    Enum e = Search.HITHER; // Upcast
    // e.values(); // No values() in Enum
    for(Enum en : e.getClass().getEnumConstants())
      System.out.println(en);
  }
}
/* Output:
HITHER
YON
*/
```

Because `getEnumConstants()` is a method of `Class`, you can call it for a class that has no enumerations:

```
// enums/NonEnum.java

public class NonEnum {
  public static void main(String[] args) {
    Class<Integer> intClass = Integer.class;
    try {
      for(Object en : intClass.getEnumConstants())
        System.out.println(en);
    } catch(Exception e) {
      System.out.println("Expected: " + e);
    }
  }
```

```
}
/* Output:
Expected: java.lang.NullPointerException
*/
```

The method returns `null`, so you get an exception if you try to use the result.

Implements, not Inherits

We've established that all `enum`s extend `java.lang.Enum`. Java does not support multiple inheritance so this means you cannot create an `enum` via inheritance:

```
enum NotPossible extends Pet { ... // Won't work
```

However, it *is* possible to create an `enum` that implements one or more interfaces:

```
// enums/cartoons/EnumImplementation.java
// An enum can implement an interface
// {java enums.cartoons.EnumImplementation}
package enums.cartoons;
import java.util.*;
import java.util.function.*;

enum CartoonCharacter
implements Supplier<CartoonCharacter> {
  SLAPPY, SPANKY, PUNCHY,
  SILLY, BOUNCY, NUTTY, BOB;
  private Random rand =
    new Random(47);
  @Override public CartoonCharacter get() {
    return values()[rand.nextInt(values().length)];
  }
}

public class EnumImplementation {
  public static <T> void printNext(Supplier<T> rg) {
    System.out.print(rg.get() + ", ");
  }
  public static void main(String[] args) {
    // Choose any instance:
```

```
    CartoonCharacter cc = CartoonCharacter.BOB;
    for(int i = 0; i < 10; i++)
      printNext(cc);
  }
}
/* Output:
BOB, PUNCHY, BOB, SPANKY, NUTTY, PUNCHY, SLAPPY, NUTTY,
NUTTY, SLAPPY,
*/
```

The result is slightly odd, because to call a method you must have an instance of the `enum` to call it on. However, a `CartoonCharacter` can now be accepted by any method that takes a `Supplier`; for example, `printNext()`.

Random Selection

Many of the examples in this chapter require random selection from among `enum` instances, as you saw in `CartoonCharacter.get()`. It's possible to generalize this task using generics and put the result in the common library:

```
// onjava/Enums.java
package onjava;
import java.util.*;

public class Enums {
  private static Random rand = new Random(47);
  public static
  <T extends Enum<T>> T random(Class<T> ec) {
    return random(ec.getEnumConstants());
  }
  public static <T> T random(T[] values) {
    return values[rand.nextInt(values.length)];
  }
}
```

The rather odd syntax `<T extends Enum<T>>` describes `T` as an `enum` instance. By passing in `Class<T>`, we make the class object available, and the array of `enum` instances can thus be produced. The overloaded `random()` method must only know

it gets a `T[]` because it doesn't perform `Enum` operations; it only selects an array element at random. The return type is the exact type of the `enum`.

Here's a simple test of the `random()` method:

```
// enums/RandomTest.java
import onjava.*;

enum Activity { SITTING, LYING, STANDING, HOPPING,
  RUNNING, DODGING, JUMPING, FALLING, FLYING }

public class RandomTest {
  public static void main(String[] args) {
    for(int i = 0; i < 20; i++)
      System.out.print(
        Enums.random(Activity.class) + " ");
  }
}
/* Output:
STANDING FLYING RUNNING STANDING RUNNING STANDING LYING
DODGING SITTING RUNNING HOPPING HOPPING HOPPING RUNNING
STANDING LYING FALLING RUNNING FLYING LYING
*/
```

Although `Enums` is a small class, you'll see it prevents a fair amount of duplication in this chapter. Duplication tends to produce mistakes, so eliminating duplication is a useful pursuit.

Using Interfaces for Organization

The inability to inherit from an `enum` can be a bit frustrating at times. The motivation for inheriting from an `enum` comes partly from wanting to extend the number of elements in the original `enum`, and partly from wanting to create subcategories by using subtypes.

You can categorize elements by grouping then together inside an interface and creating an enumeration based on that interface. For example, suppose you have different classes of food that you'd like to create as `enum`s, but you'd still like each one to be a type of `Food`. Here's what it looks like:

```
// enums/menu/Food.java
// Subcategorization of enums within interfaces
package enums.menu;

public interface Food {
  enum Appetizer implements Food {
    SALAD, SOUP, SPRING_ROLLS;
  }
  enum MainCourse implements Food {
    LASAGNE, BURRITO, PAD_THAI,
    LENTILS, HUMMUS, VINDALOO;
  }
  enum Dessert implements Food {
    TIRAMISU, GELATO, BLACK_FOREST_CAKE,
    FRUIT, CREME_CARAMEL;
  }
  enum Coffee implements Food {
    BLACK_COFFEE, DECAF_COFFEE, ESPRESSO,
    LATTE, CAPPUCCINO, TEA, HERB_TEA;
  }
}
```

The only subtyping available for an `enum` is that of interface implementation so each nested `enum` `implements` the surrounding interface `Food`. Now it's possible to say that "everything is a type of `Food`" as shown here:

```
// enums/menu/TypeOfFood.java
// {java enums.menu.TypeOfFood}
package enums.menu;
import static enums.menu.Food.*;

public class TypeOfFood {
  public static void main(String[] args) {
    Food food = Appetizer.SALAD;
    food = MainCourse.LASAGNE;
    food = Dessert.GELATO;
    food = Coffee.CAPPUCCINO;
  }
}
```

The upcast to `Food` works for each `enum` type that `implements` `Food`, so they are all types of `Food`.

An interface, however, is not as useful as an enum when you deal with a set of types. For an "enum of enums" you can create a surrounding enum with one instance for each enum in Food:

```
// enums/menu/Course.java
package enums.menu;
import onjava.*;

public enum Course {
  APPETIZER(Food.Appetizer.class),
  MAINCOURSE(Food.MainCourse.class),
  DESSERT(Food.Dessert.class),
  COFFEE(Food.Coffee.class);
  private Food[] values;
  private Course(Class<? extends Food> kind) {
    values = kind.getEnumConstants();
  }
  public Food randomSelection() {
    return Enums.random(values);
  }
}
```

Each of the above enums takes the corresponding Class object as a constructor argument, from which it can extract and store all the enum instances using getEnumConstants(). These instances are later used in randomSelection(), so now we can create a randomly generated meal by selecting one Food item from each Course:

```
// enums/menu/Meal.java
// {java enums.menu.Meal}
package enums.menu;

public class Meal {
  public static void main(String[] args) {
    for(int i = 0; i < 5; i++) {
      for(Course course : Course.values()) {
        Food food = course.randomSelection();
        System.out.println(food);
      }
      System.out.println("***");
    }
  }
```

```
}
/* Output:
SPRING_ROLLS
VINDALOO
FRUIT
DECAF_COFFEE
***
SOUP
VINDALOO
FRUIT
TEA
***
SALAD
BURRITO
FRUIT
TEA
***
SALAD
BURRITO
CREME_CARAMEL
LATTE
***
SOUP
BURRITO
TIRAMISU
ESPRESSO
***
*/
```

Here, the value of creating an `enum` of `enum`s is to iterate through each `Course`. Later, in the `VendingMachine.java` example, you'll see another approach to categorization dictated by different constraints.

Another, more compact, approach to the problem of categorization is to nest `enum`s within `enum`s, like this:

```
// enums/SecurityCategory.java
// More succinct subcategorization of enums
import onjava.*;

enum SecurityCategory {
  STOCK(Security.Stock.class),
  BOND(Security.Bond.class);
  Security[] values;
  SecurityCategory(Class<? extends Security> kind) {
    values = kind.getEnumConstants();
  }
  interface Security {
    enum Stock implements Security {
      SHORT, LONG, MARGIN
    }
    enum Bond implements Security {
      MUNICIPAL, JUNK
    }
  }
  public Security randomSelection() {
    return Enums.random(values);
  }
  public static void main(String[] args) {
    for(int i = 0; i < 10; i++) {
      SecurityCategory category =
        Enums.random(SecurityCategory.class);
      System.out.println(category + ": " +
        category.randomSelection());
    }
  }
}
/* Output:
BOND: MUNICIPAL
BOND: MUNICIPAL
STOCK: MARGIN
STOCK: MARGIN
BOND: JUNK
STOCK: SHORT
STOCK: LONG
STOCK: LONG
BOND: MUNICIPAL
BOND: JUNK
```

```
*/
```

The `Security` interface is necessary to collect the contained `enums` together as a common type. These are then categorized into the `enums` within `SecurityCategory`.

If we take this approach with the `Food` example, the result is:

```
// enums/menu/Meal2.java
// {java enums.menu.Meal2}
package enums.menu;
import onjava.*;

public enum Meal2 {
  APPETIZER(Food.Appetizer.class),
  MAINCOURSE(Food.MainCourse.class),
  DESSERT(Food.Dessert.class),
  COFFEE(Food.Coffee.class);
  private Food[] values;
  private Meal2(Class<? extends Food> kind) {
    values = kind.getEnumConstants();
  }
  public interface Food {
    enum Appetizer implements Food {
      SALAD, SOUP, SPRING_ROLLS;
    }
    enum MainCourse implements Food {
      LASAGNE, BURRITO, PAD_THAI,
      LENTILS, HUMMUS, VINDALOO;
    }
    enum Dessert implements Food {
      TIRAMISU, GELATO, BLACK_FOREST_CAKE,
      FRUIT, CREME_CARAMEL;
    }
    enum Coffee implements Food {
      BLACK_COFFEE, DECAF_COFFEE, ESPRESSO,
      LATTE, CAPPUCCINO, TEA, HERB_TEA;
    }
  }
  public Food randomSelection() {
    return Enums.random(values);
  }
  public static void main(String[] args) {
```

```
    for(int i = 0; i < 5; i++) {
      for(Meal2 meal : Meal2.values()) {
        Food food = meal.randomSelection();
        System.out.println(food);
      }
      System.out.println("***");
    }
  }
}
/* Output:
SPRING_ROLLS
VINDALOO
FRUIT
DECAF_COFFEE
***
SOUP
VINDALOO
FRUIT
TEA
***
SALAD
BURRITO
FRUIT
TEA
***
SALAD
BURRITO
CREME_CARAMEL
LATTE
***
SOUP
BURRITO
TIRAMISU
ESPRESSO
***
*/
```

In the end, it's only a reorganization of the code but it can produce a clearer structure in some cases.

Using `EnumSet` Instead of Flags

A `Set` is a kind of collection that only allows one of each type of object to be added. An `enum` requires that all its members be unique, so seems to have set behavior, but because you can't add or remove elements it's not very useful as a set. The `EnumSet` was added to work in concert with `enum`s to create a replacement for traditional `int`-based "bit flags." Such flags are used to indicate some kind of on-off information, but you end up manipulating bits rather than concepts, so it's easy to write confusing code.

The `EnumSet` is designed for speed, because it must compete effectively with bit flags (operations are typically much faster than a `HashSet`). Internally, it is represented by (if possible) a single `long` that is treated as a bit-vector, so it's extremely fast and efficient. The benefit is that you now have a much more expressive way to indicate the presence or absence of a binary feature, without worrying about performance.

The elements of an `EnumSet` must come from a single `enum`. A possible example uses an `enum` of positions in a building where alarm sensors are present:

```
// enums/AlarmPoints.java
package enums;
public enum AlarmPoints {
  STAIR1, STAIR2, LOBBY, OFFICE1, OFFICE2, OFFICE3,
  OFFICE4, BATHROOM, UTILITY, KITCHEN
}
```

The `EnumSet` keeps track of the alarm status:

```
// enums/EnumSets.java
// Operations on EnumSets
// {java enums.EnumSets}
package enums;
import java.util.*;
import static enums.AlarmPoints.*;

public class EnumSets {
  public static void main(String[] args) {
    EnumSet<AlarmPoints> points =
      EnumSet.noneOf(AlarmPoints.class); // Empty
    points.add(BATHROOM);
```

```
    System.out.println(points);
    points.addAll(
      EnumSet.of(STAIR1, STAIR2, KITCHEN));
    System.out.println(points);
    points = EnumSet.allOf(AlarmPoints.class);
    points.removeAll(
      EnumSet.of(STAIR1, STAIR2, KITCHEN));
    System.out.println(points);
    points.removeAll(
      EnumSet.range(OFFICE1, OFFICE4));
    System.out.println(points);
    points = EnumSet.complementOf(points);
    System.out.println(points);
  }
}
/* Output:
[BATHROOM]
[STAIR1, STAIR2, BATHROOM, KITCHEN]
[LOBBY, OFFICE1, OFFICE2, OFFICE3, OFFICE4, BATHROOM,
UTILITY]
[LOBBY, BATHROOM, UTILITY]
[STAIR1, STAIR2, OFFICE1, OFFICE2, OFFICE3, OFFICE4,
KITCHEN]
*/
```

A `static import` is used to simplify using `enum` constants. The method names are fairly self-explanatory, and you can find the full details in the JDK documentation. When you look at this documentation, you'll see that the `of()` method is overloaded both with varargs and with individual methods taking two through five explicit arguments. This is an indication of the concern for performance with `EnumSet`, because a single `of()` method using varargs could have solved the problem, but it's slightly less efficient than having explicit arguments. Thus, if you call `of()` with two through five arguments you get the explicit (slightly faster) method calls, but if you call it with one argument or more than five, you get the varargs version of `of()`. Notice that if you call it with one argument, the compiler will not construct the varargs array and so there is no extra overhead for calling that version with a single argument.

`EnumSets` are built on top of `longs`, a `long` is 64 bits, and each `enum` instance requires one bit to indicate presence or absence. This means you can have an `EnumSet` for an

enum of up to 64 elements without going beyond a single long. What happens for more than 64 elements in your enum?

```
// enums/BigEnumSet.java
import java.util.*;

public class BigEnumSet {
  enum Big { A0, A1, A2, A3, A4, A5, A6, A7, A8, A9,
    A10, A11, A12, A13, A14, A15, A16, A17, A18, A19,
    A20, A21, A22, A23, A24, A25, A26, A27, A28, A29,
    A30, A31, A32, A33, A34, A35, A36, A37, A38, A39,
    A40, A41, A42, A43, A44, A45, A46, A47, A48, A49,
    A50, A51, A52, A53, A54, A55, A56, A57, A58, A59,
    A60, A61, A62, A63, A64, A65, A66, A67, A68, A69,
    A70, A71, A72, A73, A74, A75 }
  public static void main(String[] args) {
    EnumSet<Big> bigEnumSet = EnumSet.allOf(Big.class);
    System.out.println(bigEnumSet);
  }
}
/* Output:
[A0, A1, A2, A3, A4, A5, A6, A7, A8, A9, A10, A11, A12,
A13, A14, A15, A16, A17, A18, A19, A20, A21, A22, A23,
A24, A25, A26, A27, A28, A29, A30, A31, A32, A33, A34,
A35, A36, A37, A38, A39, A40, A41, A42, A43, A44, A45,
A46, A47, A48, A49, A50, A51, A52, A53, A54, A55, A56,
A57, A58, A59, A60, A61, A62, A63, A64, A65, A66, A67,
A68, A69, A70, A71, A72, A73, A74, A75]
*/
```

The EnumSet clearly has no problem with an enum that has more than 64 elements, so we can presume it adds another long when necessary.

Using EnumMap

An EnumMap is a specialized Map that requires its keys be from a single enum. Because of the constraints on an enum, an EnumMap can be implemented internally as an array. Thus they are extremely fast, so you can freely use EnumMaps for enum-based lookups.

You can only call put() for keys in your enum, but other than that it's like using an ordinary Map.

Here's an example that demonstrates the *Command* design pattern. This pattern starts with an interface containing (typically) a single method, and creates multiple implementations with different behavior for that method. You install Command objects, and your program calls them when necessary:

```
// enums/EnumMaps.java
// Basics of EnumMaps
// {java enums.EnumMaps}
package enums;
import java.util.*;
import static enums.AlarmPoints.*;

interface Command { void action(); }

public class EnumMaps {
  public static void main(String[] args) {
    EnumMap<AlarmPoints,Command> em =
      new EnumMap<>(AlarmPoints.class);
    em.put(KITCHEN,
      () -> System.out.println("Kitchen fire!"));
    em.put(BATHROOM,
      () -> System.out.println("Bathroom alert!"));
    for(Map.Entry<AlarmPoints,Command> e:
        em.entrySet()) {
      System.out.print(e.getKey() + ": ");
      e.getValue().action();
    }
    try { // If there's no value for a particular key:
      em.get(UTILITY).action();
    } catch(Exception e) {
      System.out.println("Expected: " + e);
    }
  }
}
/* Output:
BATHROOM: Bathroom alert!
KITCHEN: Kitchen fire!
Expected: java.lang.NullPointerException
*/
```

Just as with `EnumSet`, the order of elements in the `EnumMap` is determined by their order of definition in the `enum`.

The last part of `main()` shows there is always a key entry for each of the `enum`s, but the value is `null` unless you have called `put()` for that key.

One advantage of `EnumMap` over *constant-specific methods* (described next) is that with an `EnumMap` you can change the value objects, whereas you'll see that constant-specific methods are fixed at compile time.

As you'll see later in the chapter, `EnumMaps` can perform *multiple dispatching* for situations where you have multiple types of `enum`s interacting with each other.

Constant-Specific Methods

Java `enum`s can give each `enum` instance different behavior by creating methods for each one. To do this, you define one or more `abstract` methods as part of the `enum`, then define the methods for each `enum` instance. For example:

```
// enums/ConstantSpecificMethod.java
import java.util.*;
import java.text.*;

public enum ConstantSpecificMethod {
  DATE_TIME {
    @Override String getInfo() {
      return
        DateFormat.getDateInstance()
          .format(new Date());
    }
  },
  CLASSPATH {
    @Override String getInfo() {
      return System.getenv("CLASSPATH");
    }
  },
  VERSION {
    @Override String getInfo() {
      return System.getProperty("java.version");
    }
  };
  abstract String getInfo();
  public static void main(String[] args) {
```

```
    for(ConstantSpecificMethod csm : values())
      System.out.println(csm.getInfo());
  }
}
/* Output:
Jan 24, 2021
C:\Git\OnJava8\ExtractedExamples\\gradle\wrapper\gradle
-wrapper.jar
1.8.0_41
*/
```

You can look up and call methods via their associated `enum` instance. This is often called *table-driven code* (and note the similarity to the aforementioned Command pattern).

In object-oriented programming, different behavior is associated with different classes. Because each instance of an `enum` can have its own behavior via constant-specific methods, this suggests that each instance is a distinct type. In the above example, each `enum` instance is treated as the "base type" `ConstantSpecificMethod` but you get polymorphic behavior with the method call `getInfo()`.

However, you can only take the similarity so far. You cannot treat `enum` instances as class types:

```
// enums/NotClasses.java
// {ExcludeFromGradle}
// javap -c LikeClasses

enum LikeClasses {
  WINKEN {
    @Override void behavior() {
      System.out.println("Behavior1");
    }
  },
  BLINKEN {
    @Override void behavior() {
      System.out.println("Behavior2");
    }
  },
  NOD {
    @Override void behavior() {
      System.out.println("Behavior3");
```

```
    }
  };
  abstract void behavior();
}

public class NotClasses {
  // void f1(LikeClasses.WINKEN instance) {} // Nope
}
/* Output: (First 12 Lines)
Compiled from "NotClasses.java"
abstract class LikeClasses extends
java.lang.Enum<LikeClasses> {
  public static final LikeClasses WINKEN;

  public static final LikeClasses BLINKEN;

  public static final LikeClasses NOD;

  public static LikeClasses[] values();
    Code:
       0: getstatic     #2                  // Field
$VALUES:[LLikeClasses;
       3: invokevirtual #3                  // Method
"[LLikeClasses;".clone:()Ljava/lang/Object;
                  ...
*/
```

In `f1()`, the compiler doesn't allow you to use an `enum` instance as a class type, which makes sense if you consider the code generated by the compiler—each `enum` element is a `static final` instance of `LikeClasses`.

Also, because they are `static`, `enum` instances of inner `enum`s do not behave like ordinary inner classes; you cannot access non-`static` fields or methods in the outer class.

Now consider a car wash. Each customer is given a menu of choices for their wash, and each option performs a different action. A constant-specific method can be associated with each option, and an `EnumSet` holds the customer's selections:

```
// enums/CarWash.java
import java.util.*;

public class CarWash {
  public enum Cycle {
    UNDERBODY {
      @Override void action() {
        System.out.println("Spraying the underbody");
      }
    },
    WHEELWASH {
      @Override void action() {
        System.out.println("Washing the wheels");
      }
    },
    PREWASH {
      @Override void action() {
        System.out.println("Loosening the dirt");
      }
    },
    BASIC {
      @Override void action() {
        System.out.println("The basic wash");
      }
    },
    HOTWAX {
      @Override void action() {
        System.out.println("Applying hot wax");
      }
    },
    RINSE {
      @Override void action() {
        System.out.println("Rinsing");
      }
    },
    BLOWDRY {
      @Override void action() {
        System.out.println("Blowing dry");
      }
    };
    abstract void action();
  }
```

```
  EnumSet<Cycle> cycles =
    EnumSet.of(Cycle.BASIC, Cycle.RINSE);
  public void add(Cycle cycle) {
    cycles.add(cycle);
  }
  public void washCar() {
    for(Cycle c : cycles)
      c.action();
  }
  @Override public String toString() {
    return cycles.toString();
  }
  public static void main(String[] args) {
    CarWash wash = new CarWash();
    System.out.println(wash);
    wash.washCar();
    // Order of addition is unimportant:
    wash.add(Cycle.BLOWDRY);
    wash.add(Cycle.BLOWDRY); // Duplicates ignored
    wash.add(Cycle.RINSE);
    wash.add(Cycle.HOTWAX);
    System.out.println(wash);
    wash.washCar();
  }
}
/* Output:
[BASIC, RINSE]
The basic wash
Rinsing
[BASIC, HOTWAX, RINSE, BLOWDRY]
The basic wash
Applying hot wax
Rinsing
Blowing dry
*/
```

The syntax for defining a constant-specific method is effectively that of an anonymous inner class, but more succinct.

This example also shows more characteristics of `EnumSet`s. It's a set so it will only hold one of each item. Duplicate calls to `add()` with the same argument are ignored (this makes sense because you can only flip a bit "on" once). Also, the order you add

enum instances is unimportant—the output order is determined by the declaration order of the enum.

Is it possible to override constant-specific methods, instead of implementing an abstract method? Yes, as seen here:

```
// enums/OverrideConstantSpecific.java

public enum OverrideConstantSpecific {
  NUT, BOLT,
  WASHER {
    @Override void f() {
      System.out.println("Overridden method");
    }
  };
  void f() {
    System.out.println("default behavior");
  }
  public static void main(String[] args) {
    for(OverrideConstantSpecific ocs : values()) {
      System.out.print(ocs + ": ");
      ocs.f();
    }
  }
}
/* Output:
NUT: default behavior
BOLT: default behavior
WASHER: Overridden method
*/
```

Although enums do prevent certain types of code, in general, experiment with them as if they were classes.

Chain of Responsibility with enums

In the *Chain of Responsibility* design pattern, you create a number of different ways to solve a problem and chain them together. When a request occurs, it is passed along the chain until one of the solutions can handle the request.

You can easily implement a simple *Chain of Responsibility* with constant-specific methods. Consider a model of a post office, which tries to deal with each piece of mail in the most general way possible, but must keep trying until it ends up treating the mail as a dead letter. Each attempt can be thought of as a *Strategy* (another design pattern), and the entire list together is a Chain of Responsibility.

We start by describing a piece of mail. All the different characteristics of interest can be expressed using enums. Because `Mail` objects are randomly generated, the easiest way to reduce the probability of (for example) a piece of mail being given a `YES` for `GeneralDelivery` is to create more non-`YES` instances, so the `enum` definitions look a little funny at first.

Within `Mail`, you'll see `randomMail()`, which creates random pieces of test mail. The `generator()` method produces an `Iterable` object that uses `randomMail()` to produce a number of mail objects, one each time you call `next()` via the iterator. This construct allows the simple creation of a *for-in* loop by calling `Mail.generator()`:

```
// enums/PostOffice.java
// Modeling a post office
import java.util.*;
import onjava.*;

class Mail {
  // The NO's reduce probability of random selection:
  enum GeneralDelivery {YES,NO1,NO2,NO3,NO4,NO5}
  enum Scannability {UNSCANNABLE,YES1,YES2,YES3,YES4}
  enum Readability {ILLEGIBLE,YES1,YES2,YES3,YES4}
  enum Address {INCORRECT,OK1,OK2,OK3,OK4,OK5,OK6}
  enum ReturnAddress {MISSING,OK1,OK2,OK3,OK4,OK5}
  GeneralDelivery generalDelivery;
  Scannability scannability;
  Readability readability;
  Address address;
  ReturnAddress returnAddress;
  static long counter = 0;
  long id = counter++;
  @Override public String toString() {
    return "Mail " + id;
  }
  public String details() {
    return toString() +
```

```
      ", General Delivery: " + generalDelivery +
      ", Address Scannability: " + scannability +
      ", Address Readability: " + readability +
      ", Address Address: " + address +
      ", Return address: " + returnAddress;
  }
  // Generate test Mail:
  public static Mail randomMail() {
    Mail m = new Mail();
    m.generalDelivery =
      Enums.random(GeneralDelivery.class);
    m.scannability =
      Enums.random(Scannability.class);
    m.readability =
      Enums.random(Readability.class);
    m.address = Enums.random(Address.class);
    m.returnAddress =
      Enums.random(ReturnAddress.class);
    return m;
  }
  public static
  Iterable<Mail> generator(final int count) {
    return new Iterable<Mail>() {
      int n = count;
      @Override public Iterator<Mail> iterator() {
        return new Iterator<Mail>() {
          @Override public boolean hasNext() {
            return n-- > 0;
          }
          @Override public Mail next() {
            return randomMail();
          }
          @Override
          public void remove() { // Not implemented
            throw new UnsupportedOperationException();
          }
        };
      }
    };
  }
}
```

```
public class PostOffice {
  enum MailHandler {
    GENERAL_DELIVERY {
      @Override boolean handle(Mail m) {
        switch(m.generalDelivery) {
          case YES:
            System.out.println(
              "Using general delivery for " + m);
            return true;
          default: return false;
        }
      }
    },
    MACHINE_SCAN {
      @Override boolean handle(Mail m) {
        switch(m.scannability) {
          case UNSCANNABLE: return false;
          default:
            switch(m.address) {
              case INCORRECT: return false;
              default:
                System.out.println(
                  "Delivering "+ m + " automatically");
                return true;
            }
        }
      }
    },
    VISUAL_INSPECTION {
      @Override boolean handle(Mail m) {
        switch(m.readability) {
          case ILLEGIBLE: return false;
          default:
            switch(m.address) {
              case INCORRECT: return false;
              default:
                System.out.println(
                  "Delivering " + m + " normally");
                return true;
            }
        }
      }
```

```
    },
    RETURN_TO_SENDER {
      @Override boolean handle(Mail m) {
        switch(m.returnAddress) {
          case MISSING: return false;
          default:
            System.out.println(
              "Returning " + m + " to sender");
            return true;
        }
      }
    };
    abstract boolean handle(Mail m);
  }
  static void handle(Mail m) {
    for(MailHandler handler : MailHandler.values())
      if(handler.handle(m))
        return;
    System.out.println(m + " is a dead letter");
  }
  public static void main(String[] args) {
    for(Mail mail : Mail.generator(10)) {
      System.out.println(mail.details());
      handle(mail);
      System.out.println("*****");
    }
  }
}
/* Output:
Mail 0, General Delivery: NO2, Address Scannability:
UNSCANNABLE, Address Readability: YES3, Address
Address: OK1, Return address: OK1
Delivering Mail 0 normally
*****
Mail 1, General Delivery: NO5, Address Scannability:
YES3, Address Readability: ILLEGIBLE, Address Address:
OK5, Return address: OK1
Delivering Mail 1 automatically
*****
Mail 2, General Delivery: YES, Address Scannability:
YES3, Address Readability: YES1, Address Address: OK1,
Return address: OK5
```

```
Using general delivery for Mail 2
*****
Mail 3, General Delivery: NO4, Address Scannability:
YES3, Address Readability: YES1, Address Address:
INCORRECT, Return address: OK4
Returning Mail 3 to sender
*****
Mail 4, General Delivery: NO4, Address Scannability:
UNSCANNABLE, Address Readability: YES1, Address
Address: INCORRECT, Return address: OK2
Returning Mail 4 to sender
*****
Mail 5, General Delivery: NO3, Address Scannability:
YES1, Address Readability: ILLEGIBLE, Address Address:
OK4, Return address: OK2
Delivering Mail 5 automatically
*****
Mail 6, General Delivery: YES, Address Scannability:
YES4, Address Readability: ILLEGIBLE, Address Address:
OK4, Return address: OK4
Using general delivery for Mail 6
*****
Mail 7, General Delivery: YES, Address Scannability:
YES3, Address Readability: YES4, Address Address: OK2,
Return address: MISSING
Using general delivery for Mail 7
*****
Mail 8, General Delivery: NO3, Address Scannability:
YES1, Address Readability: YES3, Address Address:
INCORRECT, Return address: MISSING
Mail 8 is a dead letter
*****
Mail 9, General Delivery: NO1, Address Scannability:
UNSCANNABLE, Address Readability: YES2, Address
Address: OK1, Return address: OK4
Delivering Mail 9 normally
*****
*/
```

The *Chain of Responsibility* is expressed in `enum MailHandler`, and the order of the `enum` definitions determines the order in which the strategies are attempted on each piece of mail. Each strategy is tried in turn until one succeeds or they all fail, in

which case you have a dead letter.

State Machines with `enum`s

Enumerated types can be ideal for creating *state machines*. A state machine can be in a finite number of specific states. The machine normally moves from one state to the next based on an input, but there are also *transient states*; the machine moves out of these as soon as their task is performed.

There are certain allowable inputs for each state, and different inputs change the state of the machine to different new states. Because `enum`s restrict the set of possible cases, they are useful for enumerating the different states and inputs.

Each state also typically has some kind of associated output.

A vending machine is a good example of a state machine. First, we define the various inputs in an `enum`:

```
// enums/Input.java
import java.util.*;

public enum Input {
  NICKEL(5), DIME(10), QUARTER(25), DOLLAR(100),
  TOOTHPASTE(200), CHIPS(75), SODA(100), SOAP(50),
  ABORT_TRANSACTION {
    @Override public int amount() { // Disallow
      throw new RuntimeException("ABORT.amount()");
    }
  },
  STOP { // This must be the last instance.
    @Override public int amount() { // Disallow
      throw new
        RuntimeException("SHUT_DOWN.amount()");
    }
  };
  int value; // In cents
  Input(int value) { this.value = value; }
  Input() {}
  int amount() { return value; }; // In cents
  static Random rand = new Random(47);
  public static Input randomSelection() {
```

```
    // Don't include STOP:
    return
      values()[rand.nextInt(values().length - 1)];
  }
}
```

Note that two of the `Inputs` have an associated amount, so `amount()` is defined in the interface. However, it is inappropriate to call `amount()` for the other two `Input` types, so they throw an exception if you call `amount()`. Although this is a bit of an odd setup (define a method in an interface, then throw an exception if you call it for certain implementations), it is imposed upon us because of the constraints of `enums`.

The `VendingMachine` reacts to these inputs by first categorizing them via the `Category` `enum`, so it can `switch` on the categories. This example shows how `enums` make code clearer and easier to manage:

```
// enums/VendingMachine.java
// {java VendingMachine VendingMachineInput.txt}
import java.util.*;
import java.io.IOException;
import java.util.function.*;
import java.nio.file.*;
import java.util.stream.*;

enum Category {
  MONEY(Input.NICKEL, Input.DIME,
        Input.QUARTER, Input.DOLLAR),
  ITEM_SELECTION(Input.TOOTHPASTE, Input.CHIPS,
                 Input.SODA, Input.SOAP),
  QUIT_TRANSACTION(Input.ABORT_TRANSACTION),
  SHUT_DOWN(Input.STOP);
  private Input[] values;
  Category(Input... types) { values = types; }
  private static EnumMap<Input,Category> categories =
    new EnumMap<>(Input.class);
  static {
    for(Category c : Category.class.getEnumConstants())
      for(Input type : c.values)
        categories.put(type, c);
  }
  public static Category categorize(Input input) {
```

```
    return categories.get(input);
  }
}

public class VendingMachine {
  private static State state = State.RESTING;
  private static int amount = 0;
  private static Input selection = null;
  enum StateDuration { TRANSIENT } // Tagging enum
  enum State {
    RESTING {
      @Override void next(Input input) {
        switch(Category.categorize(input)) {
          case MONEY:
            amount += input.amount();
            state = ADDING_MONEY;
            break;
          case SHUT_DOWN:
            state = TERMINAL;
          default:
        }
      }
    },
    ADDING_MONEY {
      @Override void next(Input input) {
        switch(Category.categorize(input)) {
          case MONEY:
            amount += input.amount();
            break;
          case ITEM_SELECTION:
            selection = input;
            if(amount < selection.amount())
              System.out.println(
                "Insufficient money for " + selection);
            else state = DISPENSING;
            break;
          case QUIT_TRANSACTION:
            state = GIVING_CHANGE;
            break;
          case SHUT_DOWN:
            state = TERMINAL;
          default:
```

```
        }
      }
    },
    DISPENSING(StateDuration.TRANSIENT) {
      @Override void next() {
        System.out.println("here is your " + selection);
        amount -= selection.amount();
        state = GIVING_CHANGE;
      }
    },
    GIVING_CHANGE(StateDuration.TRANSIENT) {
      @Override void next() {
        if(amount > 0) {
          System.out.println("Your change: " + amount);
          amount = 0;
        }
        state = RESTING;
      }
    },
    TERMINAL {@Override
    void output() { System.out.println("Halted"); } };
    private boolean isTransient = false;
    State() {}
    State(StateDuration trans) { isTransient = true; }
    void next(Input input) {
      throw new RuntimeException("Only call " +
        "next(Input input) for non-transient states");
    }
    void next() {
      throw new RuntimeException(
        "Only call next() for " +
        "StateDuration.TRANSIENT states");
    }
    void output() { System.out.println(amount); }
  }
  static void run(Supplier<Input> gen) {
    while(state != State.TERMINAL) {
      state.next(gen.get());
      while(state.isTransient)
        state.next();
      state.output();
    }
```

```
  }
  public static void main(String[] args) {
    Supplier<Input> gen = new RandomInputSupplier();
    if(args.length == 1)
      gen = new FileInputSupplier(args[0]);
    run(gen);
  }
}

// For a basic sanity check:
class RandomInputSupplier implements Supplier<Input> {
  @Override public Input get() {
    return Input.randomSelection();
  }
}

// Create Inputs from a file of ';'-separated strings:
class FileInputSupplier implements Supplier<Input> {
  private Iterator<String> input;
  FileInputSupplier(String fileName) {
    try {
      input = Files.lines(Paths.get(fileName))
        .skip(1) // Skip the comment line
        .flatMap(s -> Arrays.stream(s.split(";")))
        .map(String::trim)
        .collect(Collectors.toList())
        .iterator();
    } catch(IOException e) {
      throw new RuntimeException(e);
    }
  }
  @Override public Input get() {
    if(!input.hasNext())
      return null;
    return Enum.valueOf(
      Input.class, input.next().trim());
  }
}
/* Output:
25
50
75
```

```
here is your CHIPS
0
100
200
here is your TOOTHPASTE
0
25
35
Your change: 35
0
25
35
Insufficient money for SODA
35
60
70
75
Insufficient money for SODA
75
Your change: 75
0
Halted
*/
```

Because selecting among `enum` instances is most often accomplished with a `switch` statement (notice the extra effort that the language goes to make a `switch` on `enum`s easy), one of the most common questions to ask when you are organizing multiple `enum`s is "What do I want to `switch` on?" Here, it's easiest to work back from the `VendingMachine` by noting that in each `State`, you `switch` on the basic categories of input action: inserting money, item selection, aborting the transaction, and turning off the machine. However, within those categories, you have different types of money that can be inserted and different items that can be selected. The `Category enum` groups the different types of `Input` so the `categorize()` method can produce the appropriate `Category` inside a `switch`. This method uses an `EnumMap` to efficiently and safely perform the lookup.

If you study `class VendingMachine`, you see how each state is different, and responds differently to input. Also note the two transient states; in `run()` the machine waits for an `Input` and doesn't stop moving through states until it is no longer in a transient state.

The VendingMachine can be tested in two ways, by using two different Supplier objects. The RandomInputSupplier just keeps producing inputs, everything except SHUT_DOWN. By running this for a long time you get a kind of sanity check to help ensure that the machine will not wander into a bad state. The FileInputSupplier takes a file describing inputs in text form, turns them into enum instances, and creates Input objects. Here's the text file used to produce the output shown above:

```
// enums/VendingMachineInput.txt
QUARTER; QUARTER; QUARTER; CHIPS;
DOLLAR; DOLLAR; TOOTHPASTE;
QUARTER; DIME; ABORT_TRANSACTION;
QUARTER; DIME; SODA;
QUARTER; DIME; NICKEL; SODA;
ABORT_TRANSACTION;
STOP;
```

The FileInputSupplier constructor turns this file into a Stream of lines, skipping the comment line. Then it uses String.split() to break each line into parts at the semicolons. This produces an array of String, which can be fed into the Stream by first converting it to a Stream, then applying flatMap(). The results have any spaces trimmed off and are turned into a List<String> from which an Iterator<String> is procured.

One limitation to this design is that the fields in VendingMachine accessed by enum State instances *must* be static, which means you can only have a single VendingMachine instance. This might not be that big of an issue if you think about an actual (embedded Java) implementation because you are likely to have only one application per machine.

Multiple Dispatching

When you are dealing with multiple interacting types, a program can get particularly messy. For example, consider a system that parses and executes mathematical expressions. You want to say Number.plus(Number), Number.multiply(Number), etc., where Number is the base class for a family of numerical objects. But when you say a.plus(b), and you don't know the exact type of either a or b, how can you get them to interact properly?

The answer starts with something you probably don't think about: Java only performs *single dispatching*. That is, if you are performing an operation on more than one object whose type is unknown, Java can invoke the dynamic binding mechanism on only one of those types. This doesn't solve the problem described here, so you end up detecting some types manually and effectively producing your own dynamic binding behavior.

The solution is *multiple dispatching* (here called *double dispatching* because there are only two dispatches). Polymorphism can only occur via method calls, so if you want double dispatching, there must be two method calls: the first to determine the first unknown type, and the second to determine the second unknown type. With multiple dispatching, you must have a virtual call for each of the types—if you are working with two different interacting type hierarchies, you'll need a virtual call in each hierarchy. Generally, you'll set up a configuration such that a single method call produces more than one virtual method call and thus services more than one type in the process. To get this effect, you work with more than one method: You'll need a method call for each dispatch. The methods in the following example (which implements the "paper, scissors, rock" game, traditionally called *RoShamBo*) are called `compete()` and `eval()` and are both members of the same type. They produce one of three possible outcomes:

```
// enums/Outcome.java
package enums;
public enum Outcome { WIN, LOSE, DRAW }
```

`Item` is the interface for the multiply-dispatched types:

```
// enums/Item.java
package enums;

public interface Item {
  Outcome compete(Item it);
  Outcome eval(Paper p);
  Outcome eval(Scissors s);
  Outcome eval(Rock r);
}
```

Each specialization of `Item` provides appropriate implementations of these methods:

```
// enums/Paper.java
package enums;
import static enums.Outcome.*;

public class Paper implements Item {
  @Override public Outcome compete(Item it) {
    return it.eval(this);
  }
  @Override
  public Outcome eval(Paper p) { return DRAW; }
  @Override
  public Outcome eval(Scissors s) { return WIN; }
  @Override
  public Outcome eval(Rock r) { return LOSE; }
  @Override public String toString() {
    return "Paper";
  }
}
```

```
// enums/Scissors.java
package enums;
import static enums.Outcome.*;

public class Scissors implements Item {
  @Override public Outcome compete(Item it) {
    return it.eval(this);
  }
  @Override
  public Outcome eval(Paper p) { return LOSE; }
  @Override
  public Outcome eval(Scissors s) { return DRAW; }
  @Override
  public Outcome eval(Rock r) { return WIN; }
  @Override public String toString() {
    return "Scissors";
  }
}
```

```
// enums/Rock.java
package enums;
import static enums.Outcome.*;

public class Rock implements Item {
  @Override public Outcome compete(Item it) {
    return it.eval(this);
  }
  @Override
  public Outcome eval(Paper p) { return WIN; }
  @Override
  public Outcome eval(Scissors s) { return LOSE; }
  @Override
  public Outcome eval(Rock r) { return DRAW; }
  @Override public String toString() {
    return "Rock";
  }
}
```

`RoShamBo1.match()` takes two `Item` objects and starts the double-dispatching process by calling `Item.compete()`:

```
// enums/RoShamBo1.java
// Demonstration of multiple dispatching
// {java enums.RoShamBo1}
package enums;
import java.util.*;

public class RoShamBo1 {
  static final int SIZE = 20;
  private static Random rand = new Random(47);
  public static Item newItem() {
    switch(rand.nextInt(3)) {
      default:
      case 0: return new Scissors();
      case 1: return new Paper();
      case 2: return new Rock();
    }
  }
  public static void match(Item a, Item b) {
    System.out.println(
      a + " vs. " + b + ": " +  a.compete(b));
```

```
  }
  public static void main(String[] args) {
    for(int i = 0; i < SIZE; i++)
      match(newItem(), newItem());
  }
}
/* Output:
Rock vs. Rock: DRAW
Paper vs. Rock: WIN
Paper vs. Rock: WIN
Paper vs. Rock: WIN
Scissors vs. Paper: WIN
Scissors vs. Scissors: DRAW
Scissors vs. Paper: WIN
Rock vs. Paper: LOSE
Paper vs. Paper: DRAW
Rock vs. Paper: LOSE
Paper vs. Scissors: LOSE
Paper vs. Scissors: LOSE
Rock vs. Scissors: WIN
Rock vs. Paper: LOSE
Paper vs. Rock: WIN
Scissors vs. Paper: WIN
Paper vs. Scissors: LOSE
Paper vs. Scissors: LOSE
Paper vs. Scissors: LOSE
Paper vs. Scissors: LOSE
*/
```

The virtual mechanism determines the type of `a`, so it wakes up inside the `compete()` function of `a`'s concrete type. The `compete()` function performs the second dispatch by calling `eval()` on the remaining type. Passing itself (`this`) as an argument to `eval()` produces a call to the overloaded `eval()` function, thus preserving the type information of the first dispatch. When the second dispatch is completed, you know the exact types of both `Item` objects.

It requires a lot of ceremony to set up multiple dispatching, but keep in mind that the benefit is the syntactic elegance achieved when making the call—instead of writing awkward code to determine the type of one or more objects during a call, you simply say, "You two! I don't care what types you are, interact properly with each other!" Make sure this kind of elegance is important to you before embarking on multiple

dispatching, however.

Dispatching with `enum`s

Performing a straight translation of `RoShamBo1.java` into an `enum`-based solution is problematic because `enum` instances are not types, so the overloaded `eval()` methods won't work—you can't use `enum` instances as argument types. However, there are different approaches to implementing multiple dispatching which benefit from `enum`s.

One approach uses a constructor to initialize each `enum` instance with a "row" of outcomes; taken together this produces a kind of lookup table:

```
// enums/RoShamBo2.java
// Switching one enum on another
// {java enums.RoShamBo2}
package enums;
import static enums.Outcome.*;

public enum RoShamBo2 implements Competitor<RoShamBo2> {
  PAPER(DRAW, LOSE, WIN),
  SCISSORS(WIN, DRAW, LOSE),
  ROCK(LOSE, WIN, DRAW);
  private Outcome vPAPER, vSCISSORS, vROCK;
  RoShamBo2(Outcome paper,
    Outcome scissors, Outcome rock) {
    this.vPAPER = paper;
    this.vSCISSORS = scissors;
    this.vROCK = rock;
  }
  @Override public Outcome compete(RoShamBo2 it) {
    switch(it) {
      default:
      case PAPER: return vPAPER;
      case SCISSORS: return vSCISSORS;
      case ROCK: return vROCK;
    }
  }
  public static void main(String[] args) {
    RoShamBo.play(RoShamBo2.class, 20);
  }
```

```
}
/* Output:
ROCK vs. ROCK: DRAW
SCISSORS vs. ROCK: LOSE
SCISSORS vs. ROCK: LOSE
SCISSORS vs. ROCK: LOSE
PAPER vs. SCISSORS: LOSE
PAPER vs. PAPER: DRAW
PAPER vs. SCISSORS: LOSE
ROCK vs. SCISSORS: WIN
SCISSORS vs. SCISSORS: DRAW
ROCK vs. SCISSORS: WIN
SCISSORS vs. PAPER: WIN
SCISSORS vs. PAPER: WIN
ROCK vs. PAPER: LOSE
ROCK vs. SCISSORS: WIN
SCISSORS vs. ROCK: LOSE
PAPER vs. SCISSORS: LOSE
SCISSORS vs. PAPER: WIN
SCISSORS vs. PAPER: WIN
SCISSORS vs. PAPER: WIN
SCISSORS vs. PAPER: WIN
*/
```

Once both types are determined in `compete()`, the only action is the return of the resulting `Outcome`. However, you can also call another method, even (for example) via a *Command* object that was assigned in the constructor.

`RoShamBo2.java` is much smaller and more straightforward than the original example, and thus easier to keep track of. Notice that you're still using two dispatches to determine the type of both objects. In `RoShamBo1.java`, both dispatches were performed using virtual method calls, but here, only the first dispatch uses a virtual method call. The second dispatch uses a `switch`, but is safe because the `enum` limits the choices in the `switch` statement.

The code that drives the `enum` is separate so it can be used in the other examples. First, the `Competitor` interface defines a type that competes with another `Competitor`:

```
// enums/Competitor.java
// Switching one enum on another
package enums;

public interface Competitor<T extends Competitor<T>> {
  Outcome compete(T competitor);
}
```

Then we define two `static` methods (`static` to avoid specifying the parameter type explicitly). First, `match()` calls `compete()` for one `Competitor` vs. another, and you see that here the type parameter need only be a `Competitor<T>`. But in `play()`, the type parameter must be both an `Enum<T>` because it is used in `Enums.random()`, and a `Competitor<T>` because it is passed to `match()`:

```
// enums/RoShamBo.java
// Common tools for RoShamBo examples
package enums;
import onjava.*;

public class RoShamBo {
  public static <T extends Competitor<T>>
  void match(T a, T b) {
    System.out.println(
      a + " vs. " + b + ": " +  a.compete(b));
  }
  public static <T extends Enum<T> & Competitor<T>>
  void play(Class<T> rsbClass, int size) {
    for(int i = 0; i < size; i++)
      match(
        Enums.random(rsbClass),Enums.random(rsbClass));
  }
}
```

The `play()` method does not have a return value that involves the type parameter `T`, so it seems like you might use wildcards inside the `Class<T>` type instead of using the leading parameter description. However, wildcards cannot extend more than one base type, so we must use the above expression.

Using Constant-Specific Methods

Because constant-specific methods allow you to provide different method implementations for each enum instance, they might seem like a perfect solution for setting up multiple dispatching. But even though they can be given different behavior in this way, enum instances are not types, so you cannot use them as argument types in method signatures. The best you can do for this example is to set up a `switch` statement:

```
// enums/RoShamBo3.java
// Using constant-specific methods
// {java enums.RoShamBo3}
package enums;
import static enums.Outcome.*;

public enum RoShamBo3 implements Competitor<RoShamBo3> {
  PAPER {
    @Override public Outcome compete(RoShamBo3 it) {
      switch(it) {
        default: // To placate the compiler
        case PAPER: return DRAW;
        case SCISSORS: return LOSE;
        case ROCK: return WIN;
      }
    }
  },
  SCISSORS {
    @Override public Outcome compete(RoShamBo3 it) {
      switch(it) {
        default:
        case PAPER: return WIN;
        case SCISSORS: return DRAW;
        case ROCK: return LOSE;
      }
    }
  },
  ROCK {
    @Override public Outcome compete(RoShamBo3 it) {
      switch(it) {
        default:
        case PAPER: return LOSE;
```

```
        case SCISSORS: return WIN;
        case ROCK: return DRAW;
      }
    }
  };
  @Override
  public abstract Outcome compete(RoShamBo3 it);
  public static void main(String[] args) {
    RoShamBo.play(RoShamBo3.class, 20);
  }
}
/* Output:
ROCK vs. ROCK: DRAW
SCISSORS vs. ROCK: LOSE
SCISSORS vs. ROCK: LOSE
SCISSORS vs. ROCK: LOSE
PAPER vs. SCISSORS: LOSE
PAPER vs. PAPER: DRAW
PAPER vs. SCISSORS: LOSE
ROCK vs. SCISSORS: WIN
SCISSORS vs. SCISSORS: DRAW
ROCK vs. SCISSORS: WIN
SCISSORS vs. PAPER: WIN
SCISSORS vs. PAPER: WIN
ROCK vs. PAPER: LOSE
ROCK vs. SCISSORS: WIN
SCISSORS vs. ROCK: LOSE
PAPER vs. SCISSORS: LOSE
SCISSORS vs. PAPER: WIN
SCISSORS vs. PAPER: WIN
SCISSORS vs. PAPER: WIN
SCISSORS vs. PAPER: WIN
*/
```

Although this is functional and not unreasonable, the solution of `RoShamBo2.java` seems to require less code when adding a new type, and thus seems more straightforward.

However, `RoShamBo3.java` can be simplified and compressed:

```
// enums/RoShamBo4.java
// {java enums.RoShamBo4}
package enums;

public enum RoShamBo4 implements Competitor<RoShamBo4> {
  ROCK {
    @Override
    public Outcome compete(RoShamBo4 opponent) {
      return compete(SCISSORS, opponent);
    }
  },
  SCISSORS {
    @Override
    public Outcome compete(RoShamBo4 opponent) {
      return compete(PAPER, opponent);
    }
  },
  PAPER {
    @Override
    public Outcome compete(RoShamBo4 opponent) {
      return compete(ROCK, opponent);
    }
  };
  Outcome compete(RoShamBo4 loser, RoShamBo4 opponent) {
    return ((opponent == this) ? Outcome.DRAW
        : ((opponent == loser) ? Outcome.WIN
                               : Outcome.LOSE));
  }
  public static void main(String[] args) {
    RoShamBo.play(RoShamBo4.class, 20);
  }
}
/* Output:
PAPER vs. PAPER: DRAW
SCISSORS vs. PAPER: WIN
SCISSORS vs. PAPER: WIN
SCISSORS vs. PAPER: WIN
ROCK vs. SCISSORS: WIN
ROCK vs. ROCK: DRAW
ROCK vs. SCISSORS: WIN
PAPER vs. SCISSORS: LOSE
SCISSORS vs. SCISSORS: DRAW
```

```
PAPER vs. SCISSORS: LOSE
SCISSORS vs. ROCK: LOSE
SCISSORS vs. ROCK: LOSE
PAPER vs. ROCK: WIN
PAPER vs. SCISSORS: LOSE
SCISSORS vs. PAPER: WIN
ROCK vs. SCISSORS: WIN
SCISSORS vs. ROCK: LOSE
SCISSORS vs. ROCK: LOSE
SCISSORS vs. ROCK: LOSE
SCISSORS vs. ROCK: LOSE
*/
```

Here, the second dispatch is performed by the two-argument version of `compete()`, which performs a sequence of comparisons and is thus similar to the action of a `switch`. It's smaller, but a bit confusing. For a large system this confusion can become debilitating.

Dispatching with `EnumMap`s

It's possible to perform a "true" double dispatch using the `EnumMap` class, which is specifically designed to work very efficiently with `enum`s. The goal is to switch on two unknown types so an `EnumMap` of `EnumMap`s produces the double dispatch:

```
// enums/RoShamBo5.java
// Multiple dispatching using an EnumMap of EnumMaps
// {java enums.RoShamBo5}
package enums;
import java.util.*;
import static enums.Outcome.*;

enum RoShamBo5 implements Competitor<RoShamBo5> {
  PAPER, SCISSORS, ROCK;
  static EnumMap<RoShamBo5,EnumMap<RoShamBo5,Outcome>>
    table = new EnumMap<>(RoShamBo5.class);
  static {
    for(RoShamBo5 it : RoShamBo5.values())
      table.put(it, new EnumMap<>(RoShamBo5.class));
    initRow(PAPER, DRAW, LOSE, WIN);
    initRow(SCISSORS, WIN, DRAW, LOSE);
```

```
    initRow(ROCK, LOSE, WIN, DRAW);
  }
  static void initRow(RoShamBo5 it,
    Outcome vPAPER, Outcome vSCISSORS, Outcome vROCK) {
    EnumMap<RoShamBo5,Outcome> row =
      RoShamBo5.table.get(it);
    row.put(RoShamBo5.PAPER, vPAPER);
    row.put(RoShamBo5.SCISSORS, vSCISSORS);
    row.put(RoShamBo5.ROCK, vROCK);
  }
  @Override public Outcome compete(RoShamBo5 it) {
    return table.get(this).get(it);
  }
  public static void main(String[] args) {
    RoShamBo.play(RoShamBo5.class, 20);
  }
}
/* Output:
ROCK vs. ROCK: DRAW
SCISSORS vs. ROCK: LOSE
SCISSORS vs. ROCK: LOSE
SCISSORS vs. ROCK: LOSE
PAPER vs. SCISSORS: LOSE
PAPER vs. PAPER: DRAW
PAPER vs. SCISSORS: LOSE
ROCK vs. SCISSORS: WIN
SCISSORS vs. SCISSORS: DRAW
ROCK vs. SCISSORS: WIN
SCISSORS vs. PAPER: WIN
SCISSORS vs. PAPER: WIN
ROCK vs. PAPER: LOSE
ROCK vs. SCISSORS: WIN
SCISSORS vs. ROCK: LOSE
PAPER vs. SCISSORS: LOSE
SCISSORS vs. PAPER: WIN
SCISSORS vs. PAPER: WIN
SCISSORS vs. PAPER: WIN
SCISSORS vs. PAPER: WIN
*/
```

The `EnumMap` is initialized using a `static` clause; you see the table-like structure of the calls to `initRow()`. Notice the `compete()` method, where both dispatches

happen in a single statement.

Using a 2-D Array

We can simplify the solution even more by noting that each `enum` instance has a fixed value (based on its declaration order) and that `ordinal()` produces this value. A two-dimensional array mapping the competitors onto the outcomes produces the smallest and most straightforward solution (and possibly the fastest, although remember that `EnumMap` uses an internal array):

```
// enums/RoShamBo6.java
// Enums using "tables" instead of multiple dispatch
// {java enums.RoShamBo6}
package enums;
import static enums.Outcome.*;

enum RoShamBo6 implements Competitor<RoShamBo6> {
  PAPER, SCISSORS, ROCK;
  private static Outcome[][] table = {
    { DRAW, LOSE, WIN }, // PAPER
    { WIN, DRAW, LOSE }, // SCISSORS
    { LOSE, WIN, DRAW }, // ROCK
  };
  @Override public Outcome compete(RoShamBo6 other) {
    return table[this.ordinal()][other.ordinal()];
  }
  public static void main(String[] args) {
    RoShamBo.play(RoShamBo6.class, 20);
  }
}
/* Output:
ROCK vs. ROCK: DRAW
SCISSORS vs. ROCK: LOSE
SCISSORS vs. ROCK: LOSE
SCISSORS vs. ROCK: LOSE
PAPER vs. SCISSORS: LOSE
PAPER vs. PAPER: DRAW
PAPER vs. SCISSORS: LOSE
ROCK vs. SCISSORS: WIN
SCISSORS vs. SCISSORS: DRAW
```

```
ROCK vs. SCISSORS: WIN
SCISSORS vs. PAPER: WIN
SCISSORS vs. PAPER: WIN
ROCK vs. PAPER: LOSE
ROCK vs. SCISSORS: WIN
SCISSORS vs. ROCK: LOSE
PAPER vs. SCISSORS: LOSE
SCISSORS vs. PAPER: WIN
SCISSORS vs. PAPER: WIN
SCISSORS vs. PAPER: WIN
SCISSORS vs. PAPER: WIN
*/
```

The `table` has exactly the same order as the calls to `initRow()` in the previous example.

The small size of this code holds great appeal over the previous examples, partly because it seems much easier to understand and modify but also because it just seems more straightforward. However, it's not as "safe" as the previous examples because it uses an array. With a larger array, you might get the size wrong, and if your tests do not cover all possibilities something could slip through the cracks.

All these solutions are different types of tables, but it's worth exploring the expression of the tables to find the one that fits best. Note that even though the above solution is the most compact, it is also fairly rigid because it can only produce a constant output given constant inputs. However, there's nothing that prevents you from having `table` produce a function object. For certain types of problems, the concept of "table-driven code" can be very powerful.

Summary

Even though enumerated types are not terribly complex in themselves, this chapter was postponed until later in the book because of what you can do with `enums` in combination with features like polymorphism, generics, and reflection.

Although they are significantly more sophisticated than `enums` in C or C++, `enums` are still a "small" feature, something the language has survived (a bit awkwardly) without for many years. And yet this chapter shows the valuable impact that a "small"

feature can have—sometimes it gives you just the right leverage to solve a problem elegantly and clearly, and as I say throughout this book, elegance is important, and clarity can make the difference between a successful solution and one that fails because others cannot understand it.

On the subject of clarity, an unfortunate source of confusion comes from the poor choice in Java 1.0 of the term "enumeration" instead of the common and well-accepted term "iterator" to indicate an object that selects each element of a sequence (as shown in Collections). Some languages even refer to enumerated data types as "enumerators!" This mistake has since been rectified in Java, but the `Enumeration` interface could not, of course, simply be removed and so is still hanging around in old (and sometimes new!) code, the library, and documentation.

Annotations

> Annotations (also known as *metadata*) provide a formalized way to add information to your code so you can easily use that data at some later point.[158]

Annotations are partly motivated by a general trend toward combining metadata with source-code files, instead of keeping it in external documents. They are also a response to feature pressure from other languages like C#.

Annotations are one of the fundamental language changes introduced in Java 5. They provide information that cannot be expressed in Java, but that you need to fully describe your program. Thus, annotations allow you to store extra information about your program in a format validated by the compiler. Annotations can generate descriptor files or even new class definitions and help ease the burden of writing "boilerplate" code. Using annotations, you can keep this metadata in the Java source code, and have the advantage of cleaner looking code, compile-time type checking and the annotation API for building processing tools for your annotations. Although a few types of metadata come predefined in Java, in general the kind of annotations you add and what you do with them are entirely up to you.

The syntax of annotations is reasonably simple and consists mainly of the addition of the `@` symbol to the language. Java 5 introduced the first three general-purpose built-in annotations, defined in `java.lang`:

`@Override`, to indicate that a method definition is intended to override a method in the base class. This generates a compiler error if you accidentally misspell the method name or give an improper signature.[159]

`@Deprecated`, to produce a compiler warning if this element is used.

`@SuppressWarnings`, to turn off inappropriate compiler warnings.

[158] Jeremy Meyer came to Crested Butte and spent two weeks with me working on this chapter. His help was invaluable.

[159] This was no doubt inspired by a similar feature in C#. The C# feature is a keyword and not an annotation, and is enforced by the compiler. That is, when you override a method in C#, you must use the `override` keyword, whereas in Java the `@Override` annotation is optional.

`@SafeVarargs`, added in Java 7 to suppress warnings for callers of a method or constructor with a generics varargs parameter.

`@FunctionalInterface`, added in Java 8 to specify that the type declaration is a functional interface.

Five additional annotation types support the creation of new annotations; you learn about these in this chapter.

Whenever you create classes or interfaces that involve repetitive work, you can usually use annotations to automate and simplify the process. Much of the extra work in *Enterprise JavaBeans* (EJBs), for example, is eliminated through annotations in EJB3.

Annotations can replace existing systems like XDoclet, an independent doclet tool that creates annotation-style doclets. In contrast, annotations are true language constructs and hence are structured and also type-checked at compile time. Keeping all the information in the actual source code and not in comments makes the code neater and easier to maintain. By using and extending the annotation API and tools, or with external bytecode manipulation libraries as you will see in this chapter, you can perform powerful inspection and manipulation of your source code as well as the bytecode.

Basic Syntax

In the example below, the method `testExecute()` is annotated with `@Test`. This doesn't do anything by itself, but the compiler ensures you have a definition for the `@Test` annotation in your CLASSPATH. Later in the chapter, we create a tool to run this method via reflection.

```
// annotations/Testable.java
package annotations;
import onjava.atunit.*;

public class Testable {
  public void execute() {
    System.out.println("Executing..");
  }
  @Test
  void testExecute() { execute(); }
}
```

Annotated methods are no different from other methods. The `@Test` annotation in this example can be used in combination with any of the modifiers like `public` or `static` or `void`. Syntactically, annotations are used in much the same way as modifiers.

Defining Annotations

Here is the definition of the annotation above. Annotation definitions look a lot like interface definitions. In fact, they compile to class files like any other Java interface:

```
// onjava/atunit/Test.java
// The @Test tag
package onjava.atunit;
import java.lang.annotation.*;

@Target(ElementType.METHOD)
@Retention(RetentionPolicy.RUNTIME)
public @interface Test {}
```

Apart from the `@` symbol, the definition of `@Test` is much like that of an empty interface. An annotation definition also requires the *meta-annotations* `@Target` and `@Retention`. `@Target` defines where you can apply this annotation (a method or a field, for example). `@Retention` defines whether the annotations are available in the source code (`SOURCE`), in the class files (`CLASS`), or at runtime (`RUNTIME`).

Annotations usually contain *elements* that specify values. A program or tool can use these parameters when processing your annotations. Elements look like interface methods, except you can declare default values.

An annotation without any elements, such as `@Test` above, is called a *marker annotation*.

Here is a simple annotation that tracks use cases in a project. Programmers annotate each method or set of methods that fulfill the requirements of a particular use case. A project manager can get an idea of project progress by counting the implemented use cases, and developers maintaining the project can easily find use cases to update, or they can debug business rules within the system.

```
// annotations/UseCase.java
import java.lang.annotation.*;

@Target(ElementType.METHOD)
@Retention(RetentionPolicy.RUNTIME)
public @interface UseCase {
  int id();
  String description() default "no description";
}
```

Notice that `id` and `description` resemble method declarations. Because `id` is type-checked by the compiler, it is a reliable way of linking a tracking database to the use case document and the source code. The element `description` has a `default` value picked up by the annotation processor if no value is specified when a method is annotated.

Here is a class with three methods annotated as use cases:

```
// annotations/PasswordUtils.java
import java.util.*;

public class PasswordUtils {
  @UseCase(id = 47, description =
  "Passwords must contain at least one numeric")
  public boolean validatePassword(String passwd) {
    return (passwd.matches("\\w*\\d\\w*"));
  }
  @UseCase(id = 48)
  public String encryptPassword(String passwd) {
   return new StringBuilder(passwd)
    .reverse().toString();
  }
```

```
  @UseCase(id = 49, description =
  "New passwords can't equal previously used ones")
  public boolean checkForNewPassword(
    List<String> prevPasswords, String passwd) {
    return !prevPasswords.contains(passwd);
  }
}
```

The values of the annotation elements are expressed as name-value pairs in parentheses after the `@UseCase` declaration. The annotation for `encryptPassword()` is not passed a value for the `description` element here, so the default value defined in the `@interface UseCase` will appear when the class runs through an annotation processor.

Imagine using an approach like this to "sketch" out your system, then filling in the functionality as you build it.

Meta-Annotations

There are currently only five standard annotations (described earlier) and five meta-annotations defined in the Java language. The meta-annotations are for annotating annotations:

Annotation	Effect
`@Target`	Where this annotation can be applied. The possible `ElementType` arguments are: `CONSTRUCTOR`: Constructor declaration `FIELD`: Field declaration (includes `enum` constants) `LOCAL_VARIABLE`: Local variable declaration `METHOD`: Method declaration `PACKAGE`: Package declaration `PARAMETER`: Parameter declaration `TYPE`: Class, interface (including annotation type), or `enum` declaration
`@Retention`	How long the annotation information is kept. The possible `RetentionPolicy` arguments are: `SOURCE`: Annotations are discarded by the compiler. `CLASS`: Annotations are available in the class file by the compiler but can be discarded by the VM. `RUNTIME`: Annotations are retained by the VM at run time, so they can be read reflectively.
`@Documented`	Include this annotation in the Javadocs.

Annotation	Effect
`@Inherited`	Allow subclasses to inherit parent annotations.
`@Repeatable`	Can apply more than once to the same declaration (Java 8).

Most of the time, you define your own annotations and write your own processors to deal with them.

Writing Annotation Processors

Without tools to read them, annotations are hardly more useful than comments. An important part of the process of using annotations is to create and use *annotation processors*. Java provides extensions to the reflection API to help you create these tools. It also provides a `javac` compiler hook to use annotations at compile time.

Here is a very simple annotation processor that reads the annotated `PasswordUtils` class and uses reflection to look for `@UseCase` tags. Given a list of `id` values, it lists the use cases it finds and reports any that are missing:

```
// annotations/UseCaseTracker.java
import java.util.*;
import java.util.stream.*;
import java.lang.reflect.*;

public class UseCaseTracker {
  public static void
  trackUseCases(List<Integer> useCases, Class<?> cl) {
    for(Method m : cl.getDeclaredMethods()) {
      UseCase uc = m.getAnnotation(UseCase.class);
      if(uc != null) {
        System.out.println("Found Use Case " +
          uc.id() + "\n  " + uc.description());
        useCases.remove(Integer.valueOf(uc.id()));
      }
    }
    useCases.forEach(i ->
      System.out.println("Missing use case " + i));
  }
```

```
  public static void main(String[] args) {
    List<Integer> useCases = IntStream.range(47, 51)
      .boxed().collect(Collectors.toList());
    trackUseCases(useCases, PasswordUtils.class);
  }
}
/* Output:
Found Use Case 49
  New passwords can't equal previously used ones
Found Use Case 48
  no description
Found Use Case 47
  Passwords must contain at least one numeric
Missing use case 50
*/
```

This uses both the reflection method `getDeclaredMethods()` and the method `getAnnotation()`, which comes from the `AnnotatedElement` interface (classes like `Class`, `Method` and `Field` all implement this interface). This method returns the annotation object of the specified type, in this case "`UseCase`." If there are no annotations of that particular type on the annotated method, a `null` value is returned. The element values are extracted by calling `id()` and `description()`. Remember that no description was specified in the annotation for the `encryptPassword()` method, so the processor above finds the default value "`no description`" when it calls the `description()` method on that particular annotation.

Annotation Elements

The `@UseCase` tag defined in `UseCase.java` contains the `int` element `id` and `String` element `description`. Here is a list of the allowed types for annotation elements:

- All primitives (`int`, `float`, `boolean` etc.)
- `String`
- `Class`
- `enum`s
- `Annotation`s
- Arrays of any of the above

The compiler will report an error if you try to use any other types. Note you are not allowed to use any of the wrapper classes, but because of autoboxing this isn't really a limitation. You can also have elements that are themselves annotations. As you will see a bit later, nested annotations can be very helpful.

Default Value Constraints

The compiler is picky about default element values. No element can have an unspecified value. This means elements must either have default values or values provided by the class that uses the annotation.

There is another restriction: none of the non-primitive type elements are allowed to take `null` as a value, either when declared in the source code or when defined as a default value in the annotation interface. This makes it hard to write a processor that acts on the presence or absence of an element, because every element is effectively present in every annotation declaration. You can get around this by checking for specific values, like empty `Strings` or negative values:

```
// annotations/SimulatingNull.java
import java.lang.annotation.*;

@Target(ElementType.METHOD)
@Retention(RetentionPolicy.RUNTIME)
public @interface SimulatingNull {
  int id() default -1;
  String description() default "";
}
```

This is a typical idiom in annotation definitions.

Generating External Files

Annotations are especially useful when working with frameworks that require some sort of additional information to accompany your source code. Technologies like Enterprise JavaBeans (prior to EJB3) require numerous interfaces and deployment descriptors which are "boilerplate" code, defined in the same way for every bean. Web services, custom tag libraries and object/relational mapping tools like Toplink and Hibernate often require XML descriptors external to the code. After defining a

Java class, the programmer must undergo the tedium of respecifying information like the name, package and so on—information that already exists in the original class. Whenever you use an external descriptor file, you end up with two separate sources of information about a class, which usually leads to code synchronization problems. This also requires that programmers working on the project must know about editing the descriptor as well as how to write Java programs.

Suppose you want basic object/relational mapping functionality to automate the creation of a database table. You can use an XML descriptor file to specify the name of the class, each member, and information about its database mapping. Using annotations, however, you can keep all information in a single source-code file. To do this, you need annotations to define the name of the database table, the columns, and the SQL types to map to properties.

Here is an annotation that tells the annotation processor it should create a database table:

```
// annotations/database/DBTable.java
package annotations.database;
import java.lang.annotation.*;

@Target(ElementType.TYPE) // Applies to classes only
@Retention(RetentionPolicy.RUNTIME)
public @interface DBTable {
  String name() default "";
}
```

Each `ElementType` you specify in the `@Target` annotation is a restriction that tells the compiler that your annotation can only be applied to that particular type. You can specify a single value of the `enum ElementType`, or you can specify a comma-separated list of any combination of values. To apply the annotation to any `ElementType`, you can leave out the `@Target` annotation altogether, although this is uncommon.

Note that `@DBTable` has a `name()` element so the annotation can supply a name for the database table that the processor will create.

Here are the annotations for the fields:

```
// annotations/database/Constraints.java
package annotations.database;
import java.lang.annotation.*;

@Target(ElementType.FIELD)
@Retention(RetentionPolicy.RUNTIME)
public @interface Constraints {
  boolean primaryKey() default false;
  boolean allowNull() default true;
  boolean unique() default false;
}
```

```
// annotations/database/SQLString.java
package annotations.database;
import java.lang.annotation.*;

@Target(ElementType.FIELD)
@Retention(RetentionPolicy.RUNTIME)
public @interface SQLString {
  int value() default 0;
  String name() default "";
  Constraints constraints() default @Constraints;
}
```

```
// annotations/database/SQLInteger.java
package annotations.database;
import java.lang.annotation.*;

@Target(ElementType.FIELD)
@Retention(RetentionPolicy.RUNTIME)
public @interface SQLInteger {
  String name() default "";
  Constraints constraints() default @Constraints;
}
```

The `@Constraints` annotation allows the processor to extract the metadata about the database table. This represents a small subset of the constraints generally offered by databases, but it gives you the general idea. The elements `primaryKey()`, `allowNull()` and `unique()` are given sensible default values so in most cases a user of the annotation won't have to type too much.

The other two `@interfaces` define SQL types. Again, for this framework to be more useful, you define an annotation for each additional SQL type. Here, two types are enough.

These types each have a `name()` element and a `constraints()` element. The latter makes use of the nested annotation feature to embed the information about the column type's database constraints. Note that the default value for the `constraints()` element is `@Constraints`. There are no element values specified in parentheses after this annotation type so the default value of `constraints()` is actually a `@Constraints` annotation with its own default values set. To make a nested `@Constraints` annotation with uniqueness set to `true` by default, you can define its element like this:

```
// annotations/database/Uniqueness.java
// Sample of nested annotations
package annotations.database;

public @interface Uniqueness {
  Constraints constraints()
  default @Constraints(unique = true);
}
```

Here is a simple class that uses these annotations:

```
// annotations/database/Member.java
package annotations.database;

@DBTable(name = "MEMBER")
public class Member {
  @SQLString(30) String firstName;
  @SQLString(50) String lastName;
  @SQLInteger Integer age;
  @SQLString(value = 30,
  constraints = @Constraints(primaryKey = true))
  String reference;
  static int memberCount;
  public String getReference() { return reference; }
  public String getFirstName() { return firstName; }
  public String getLastName() { return lastName; }
  @Override public String toString() {
    return reference;
```

```
  }
  public Integer getAge() { return age; }
}
```

The `@DBTable` class annotation is given the value `MEMBER`, which is used as the table name. The properties `firstName` and `lastName` are both annotated with `@SQLStrings` and have element values of 30 and 50, respectively. These annotations are interesting for two reasons: First, they use the default value on the nested `@Constraints` annotation, and second, they use a shortcut feature. If you define an element on an annotation with the name `value`, then as long as it is the only element type specified, you don't need the name-value pair syntax; you can just specify the value within parentheses. This can be applied to any of the legal element types. This limits you to naming your element "value" but in the case above, it does allow for the semantically meaningful and easy-to-read annotation specification:

```
@SQLString(30)
```

The processor will use this value to set the size of the SQL column it will create.

As neat as the default-value syntax is, it quickly becomes complex. Look at the annotation on the field `reference`. This has an `@SQLString` annotation, but it must also be a primary key on the database, so the element type `primaryKey` must be set on the nested `@Constraint` annotation. This is where it gets messy. You are now forced to use the rather long-winded name-value pair form for this nested annotation, respecifying the element name *and* the `@interface` name. But because the specially named element `value` is no longer the only element value specified, you can't use the shortcut form. As you see, the result is not pretty.

Alternative Solutions

There are other ways of creating annotations for this task. You can, for example, have a single annotation class called `@TableColumn` with an `enum` element which defines values like `STRING`, `INTEGER`, `FLOAT`, etc. This eliminates the need for an `@interface` for each SQL type, but makes it impossible to qualify your types with additional elements like *size*, or *precision*, which is probably more useful.

You can also use a `String` element to describe the actual SQL type, e.g., "VARCHAR(30)" or "INTEGER". This does allow you to qualify the types, but it ties up

the mapping from Java type to SQL type in your code, which is not good design. You don't want to recompile classes if you change databases; it would be more elegant just to tell your annotation processor you use a different "flavor" of SQL, and it let it take that into account when processing the annotations.

A third workable solution is to use two annotation types together, `@Constraints` and the relevant SQL type (for example, `@SQLInteger`), to annotate the desired field. This is slightly messy but the compiler allows as many different annotations as you like on an annotation target. In Java 8, when using multiple annotations, you are allowed to use the same annotation more than once.

Annotations Don't Support Inheritance

You cannot use the `extends` keyword with `@interfaces`. This is a pity, because an elegant solution would have been to define an annotation `@TableColumn`, as suggested above, with a nested annotation of type `@SQLType`. That way, you can inherit all your SQL types, like `@SQLInteger` and `@SQLString`, from `@SQLType`. This would reduce typing and neaten the syntax. There doesn't seem to be any suggestion of annotations supporting inheritance in future releases, so the examples above seem the best you can do under the circumstances.

Implementing the Processor

Here is an example of an annotation processor which reads in a class file, checks for its database annotations and generates the SQL command for making the database:

```
// annotations/database/TableCreator.java
// Reflection-based annotation processor
// {java annotations.database.TableCreator
// annotations.database.Member}
package annotations.database;
import java.lang.annotation.*;
import java.lang.reflect.*;
import java.util.*;

public class TableCreator {
  public static void
  main(String[] args) throws Exception {
```

```
    if(args.length < 1) {
      System.out.println(
        "arguments: annotated classes");
      System.exit(0);
    }
    for(String className : args) {
      Class<?> cl = Class.forName(className);
      DBTable dbTable = cl.getAnnotation(DBTable.class);
      if(dbTable == null) {
        System.out.println(
          "No DBTable annotations in class " +
          className);
        continue;
      }
      String tableName = dbTable.name();
      // If the name is empty, use the Class name:
      if(tableName.length() < 1)
        tableName = cl.getName().toUpperCase();
      List<String> columnDefs = new ArrayList<>();
      for(Field field : cl.getDeclaredFields()) {
        String columnName = null;
        Annotation[] anns =
          field.getDeclaredAnnotations();
        if(anns.length < 1)
          continue; // Not a db table column
        if(anns[0] instanceof SQLInteger) {
          SQLInteger sInt = (SQLInteger) anns[0];
          // Use field name if name not specified
          if(sInt.name().length() < 1)
            columnName = field.getName().toUpperCase();
          else
            columnName = sInt.name();
          columnDefs.add(columnName + " INT" +
            getConstraints(sInt.constraints()));
        }
        if(anns[0] instanceof SQLString) {
          SQLString sString = (SQLString) anns[0];
          // Use field name if name not specified.
          if(sString.name().length() < 1)
            columnName = field.getName().toUpperCase();
          else
            columnName = sString.name();
```

```
          columnDefs.add(columnName + " VARCHAR(" +
            sString.value() + ")" +
            getConstraints(sString.constraints()));
        }
        StringBuilder createCommand = new StringBuilder(
          "CREATE TABLE " + tableName + "(");
        for(String columnDef : columnDefs)
          createCommand.append(
            "\n    " + columnDef + ",");
        // Remove trailing comma
        String tableCreate = createCommand.substring(
          0, createCommand.length() - 1) + ");";
        System.out.println("Table Creation SQL for " +
          className + " is:\n" + tableCreate);
      }
    }
  }
  private static
  String getConstraints(Constraints con) {
    String constraints = "";
    if(!con.allowNull())
      constraints += " NOT NULL";
    if(con.primaryKey())
      constraints += " PRIMARY KEY";
    if(con.unique())
      constraints += " UNIQUE";
    return constraints;
  }
}
/* Output:
Table Creation SQL for annotations.database.Member is:
CREATE TABLE MEMBER(
    FIRSTNAME VARCHAR(30));
Table Creation SQL for annotations.database.Member is:
CREATE TABLE MEMBER(
    FIRSTNAME VARCHAR(30),
    LASTNAME VARCHAR(50));
Table Creation SQL for annotations.database.Member is:
CREATE TABLE MEMBER(
    FIRSTNAME VARCHAR(30),
    LASTNAME VARCHAR(50),
    AGE INT);
```

```
Table Creation SQL for annotations.database.Member is:
CREATE TABLE MEMBER(
    FIRSTNAME VARCHAR(30),
    LASTNAME VARCHAR(50),
    AGE INT,
    REFERENCE VARCHAR(30) PRIMARY KEY);
*/
```

The `main()` method cycles through each of the class names on the command line. Each class is loaded using `forName()` and checked to see if it has the `@DBTable` annotation on it with `getAnnotation(DBTable.class)`. If it does, then the table name is found and stored. All fields in the class are then loaded and checked using `getDeclaredAnnotations()`. This method returns an array of all defined annotations for a particular method. The `instanceof` operator is used to determine if these annotations are of type `@SQLInteger` and `@SQLString`, and in each case the relevant `String` fragment is then created with the name of the table column. Note that because there is no inheritance of annotation interfaces, using `getDeclaredAnnotations()` is the only way you can approximate polymorphic behavior.

The nested `@Constraint` annotation is passed to the `getConstraints()` which builds up a `String` containing the SQL constraints.

It is worth mentioning that the technique shown above is a somewhat naïve way of defining an object/relational mapping. Having an annotation of type `@DBTable` which takes the table name as a parameter forces you to recompile your Java code to change the table name. This might not be desirable. There are many available frameworks for mapping objects to relational databases, and more and more of them are making use of annotations.

Using `javac` to Process Annotations

Through `javac`, you can work with annotations on Java source files rather than compiled classes, by creating compile-time annotation processors. There's an important limitation, however: you cannot change the source code via the annotation processor. The only way to influence the outcome is by creating new files.

If your annotation processor creates a new source file, that file is itself checked for annotations in a new round of processing. The tool will continue round after round

of processing until no more source files are created. It then compiles all the source files.

Each annotation you write will need its own processor, but `javac` can easily group several annotation processors together. You can specify multiple classes to be processed, and you can add listeners to receive notifications when an annotation processing round is complete.

The examples in this section will get you started but if you must go deeper, be ready to thrash around a bit, with plenty of visits to Google and StackOverflow.

The Simplest Processor

Let's start by defining the simplest processor we can imagine, just for something to compile and test. Here's the annotation definition:

```
// annotations/simplest/Simple.java
// A bare-bones annotation
package annotations.simplest;
import java.lang.annotation.Retention;
import java.lang.annotation.RetentionPolicy;
import java.lang.annotation.Target;
import java.lang.annotation.ElementType;

@Retention(RetentionPolicy.SOURCE)
@Target({ElementType.TYPE, ElementType.METHOD,
         ElementType.CONSTRUCTOR,
         ElementType.ANNOTATION_TYPE,
         ElementType.PACKAGE, ElementType.FIELD,
         ElementType.LOCAL_VARIABLE})
public @interface Simple {
    String value() default "-default-";
}
```

The `@Retention` is now `SOURCE`, which means the annotations do not survive into the compiled code. This is not necessary to manipulate annotations at compile time—it just makes the point that, here, `javac` is the only agent with the opportunity to process annotations.

The `@Target` declaration shows almost all the possible target types (except for `PACKAGE`), again just for demonstration.

Here's an example to test it:

```
// annotations/simplest/SimpleTest.java
// Test the "Simple" annotation
// {java annotations.simplest.SimpleTest}
package annotations.simplest;

@Simple
public class SimpleTest {
  @Simple
  int i;
  @Simple
  public SimpleTest() {}
  @Simple
  public void foo() {
    System.out.println("SimpleTest.foo()");
  }
  @Simple
  public void bar(String s, int i, float f) {
    System.out.println("SimpleTest.bar()");
  }
  @Simple
  public static void main(String[] args) {
    @Simple
    SimpleTest st = new SimpleTest();
    st.foo();
  }
}
/* Output:
SimpleTest.foo()
*/
```

Here, we annotate everything that `@Simple` is allowed by its `@Target` declaration.

`SimpleTest.java` only requires `Simple.java` to compile successfully. Nothing happens when we compile it, though. `javac` allows the `@Simple` annotation (as long as it exists) but it doesn't do anything with it until we create an annotation processor and hook it into the compiler.

Here's a very simple processor. All it does is print information about the annotations:

```
// annotations/simplest/SimpleProcessor.java
// A bare-bones annotation processor
package annotations.simplest;
import javax.annotation.processing.*;
import javax.lang.model.SourceVersion;
import javax.lang.model.element.*;
import java.util.*;

@SupportedAnnotationTypes(
  "annotations.simplest.Simple")
@SupportedSourceVersion(SourceVersion.RELEASE_8)
public class SimpleProcessor
extends AbstractProcessor {
  @Override public boolean process(
    Set<? extends TypeElement> annotations,
    RoundEnvironment env) {
    for(TypeElement t : annotations)
      System.out.println(t);
    for(Element el :
      env.getElementsAnnotatedWith(Simple.class))
      display(el);
    return false;
  }
  private void display(Element el) {
    System.out.println("==== " + el + " ====");
    System.out.println(el.getKind() +
      " : " + el.getModifiers() +
      " : " + el.getSimpleName() +
      " : " + el.asType());
    if(el.getKind().equals(ElementKind.CLASS)) {
      TypeElement te = (TypeElement)el;
      System.out.println(te.getQualifiedName());
      System.out.println(te.getSuperclass());
      System.out.println(te.getEnclosedElements());
    }
    if(el.getKind().equals(ElementKind.METHOD)) {
      ExecutableElement ex = (ExecutableElement)el;
      System.out.print(ex.getReturnType() + " ");
      System.out.print(ex.getSimpleName() + "(");
      System.out.println(ex.getParameters() + ")");
    }
  }
```

```
}
```

The (old, defunct) `apt` version of annotation processors required extra methods to establish which annotations were supported, and which Java version is supported. Now, however, you can simply use the annotations `@SupportedAnnotationTypes` and `@SupportedSourceVersion`. (This is a good example of how annotations simplify your code).

The only method you must implement is `process()`, where all the action happens. The first argument tells you which annotations are present, and the second argument contains all the rest of the information. All we do here is print the annotations (there's only one) but see the `TypeElement` documentation for other actions.

Using the second `process()` argument, we loop through all the elements annotated with `@Simple`, and call our `display()` method on each one. Every `Element` can produce basic information about itself; for example, `getModifiers()` tells you if it's `public` and `static`.

`Element` can only do things common to all basic objects parsed by the compiler, whereas things like classes and methods have additional information to extract. So (and perhaps this was obvious if you read the right document, but it wasn't in any documentation I found—I had to find clues via StackOverflow) you check to see what `ElementKind` it is, then downcast it to the more specific type of element—here, `TypeElement` for `CLASS` and `ExecutableElement` for `METHOD`. At that point, you can call the additional methods for those `Element` types.

A dynamic downcast (which is not checked at compile-time) is a very un-Java-like way of doing things, thus un-intuitive and probably why I never thought of doing it. Instead, I spent several days cycling around trying to discover how you were supposed to access the information that had seemed at least somewhat straightforward with the defunct `apt` approach. I still haven't come across anything that says the above is the canonical form, but it seems to me it is.

If you just compile `SimpleTest.java` the normal way, you won't get any results. To get the annotation output, you have to add the `–processor` flag and the annotation processor class:

```
javac -processor annotations.simplest.SimpleProcessor SimpleTest.java
```

Now the compiler produces:

```
annotations.simplest.Simple
==== annotations.simplest.SimpleTest ====
CLASS : [public] : SimpleTest : annotations.simplest.SimpleTest
annotations.simplest.SimpleTest
java.lang.Object
i,SimpleTest(),foo(),bar(java.lang.String,int,float),main(java.lang.Str\
ing[])
==== i ====
FIELD : [] : i : int
==== SimpleTest() ====
CONSTRUCTOR : [public] : <init> : ()void
==== foo() ====
METHOD : [public] : foo : ()void
void foo()
==== bar(java.lang.String,int,float) ====
METHOD : [public] : bar : (java.lang.String,int,float)void
void bar(s,i,f)
==== main(java.lang.String[]) ====
METHOD : [public, static] : main : (java.lang.String[])void
void main(args)
```

This gives you a flavor of the kinds of things you can discover, including argument names and types, return value, etc.

A More Complex Processor

When you create an annotation processor for use with `javac`, you can't use the reflection features in Java because you are working with source code, not compiled classes. The various `mirrors`[160] solve this problem by allowing you to view methods, fields and types in uncompiled source code.

Here is an annotation for extracting the public methods from a class, so they can be turned into an interface:

[160]The Java designers coyly suggest that a mirror is where you find a reflection.

```
// annotations/ifx/ExtractInterface.java
// javac-based annotation processing
package annotations.ifx;
import java.lang.annotation.*;

@Target(ElementType.TYPE)
@Retention(RetentionPolicy.SOURCE)
public @interface ExtractInterface {
  String interfaceName() default "-!!-";
}
```

The `RetentionPolicy` is `SOURCE` because there is no point in keeping this annotation in the class file after we have extracted the interface from the class. The following test class provides some public methods which can become part of an interface:

```
// annotations/ifx/Multiplier.java
// javac-based annotation processing
// {java annotations.ifx.Multiplier}
package annotations.ifx;

@ExtractInterface(interfaceName="IMultiplier")
public class Multiplier {
  public boolean flag = false;
  private int n = 0;
  public int multiply(int x, int y) {
    int total = 0;
    for(int i = 0; i < x; i++)
      total = add(total, y);
    return total;
  }
  public int fortySeven() { return 47; }
  private int add(int x, int y) {
    return x + y;
  }
  public double timesTen(double arg) {
    return arg * 10;
  }
  public static void main(String[] args) {
    Multiplier m = new Multiplier();
    System.out.println(
      "11 * 16 = " + m.multiply(11, 16));
  }
```

```
}
/* Output:
11 * 16 = 176
*/
```

The `Multiplier` class (which only works with positive integers) has a `multiply()` method which calls the private `add()` method numerous times to perform multiplication. The `add()` method is not public, so is not part of the interface. The other methods provide some syntax variety. The annotation is given the `interfaceName` of `IMultiplier` as the name of the interface to create.

Here's a compile-time processor that extracts the methods of interest and creates the new interface source-code file (which will in turn be automatically compiled as part of the "rounds"):

```
// annotations/ifx/IfaceExtractorProcessor.java
// javac-based annotation processing
package annotations.ifx;
import javax.annotation.processing.*;
import javax.lang.model.SourceVersion;
import javax.lang.model.element.*;
import javax.lang.model.util.*;
import java.util.*;
import java.util.stream.*;
import java.io.*;

@SupportedAnnotationTypes(
  "annotations.ifx.ExtractInterface")
@SupportedSourceVersion(SourceVersion.RELEASE_8)
public class IfaceExtractorProcessor
extends AbstractProcessor {
  private ArrayList<Element>
    interfaceMethods = new ArrayList<>();
  Elements elementUtils;
  private ProcessingEnvironment processingEnv;
  @Override public void init(
    ProcessingEnvironment processingEnv) {
    this.processingEnv = processingEnv;
    elementUtils = processingEnv.getElementUtils();
  }
  @Override public boolean process(
```

```
    Set<? extends TypeElement> annotations,
    RoundEnvironment env) {
    for(Element elem:env.getElementsAnnotatedWith(
        ExtractInterface.class)) {
      String interfaceName = elem.getAnnotation(
        ExtractInterface.class).interfaceName();
      for(Element enclosed :
          elem.getEnclosedElements()) {
        if(enclosed.getKind()
           .equals(ElementKind.METHOD) &&
           enclosed.getModifiers()
           .contains(Modifier.PUBLIC) &&
           !enclosed.getModifiers()
           .contains(Modifier.STATIC)) {
          interfaceMethods.add(enclosed);
        }
      }
      if(interfaceMethods.size() > 0)
        writeInterfaceFile(interfaceName);
    }
    return false;
  }
  private void
  writeInterfaceFile(String interfaceName) {
    try(
      Writer writer = processingEnv.getFiler()
        .createSourceFile(interfaceName)
        .openWriter()
    ) {
      String packageName = elementUtils
        .getPackageOf(interfaceMethods
                      .get(0)).toString();
      writer.write(
        "package " + packageName + ";\n");
      writer.write("public interface " +
        interfaceName + " {\n");
      for(Element elem : interfaceMethods) {
        ExecutableElement method =
          (ExecutableElement)elem;
        String signature = "  public ";
        signature += method.getReturnType() + " ";
        signature += method.getSimpleName();
```

```
        signature += createArgList(
          method.getParameters());
        System.out.println(signature);
        writer.write(signature + ";\n");
      }
      writer.write("}");
    } catch(Exception e) {
      throw new RuntimeException(e);
    }
  }
  private String createArgList(
    List<? extends VariableElement> parameters) {
    String args = parameters.stream()
      .map(p -> p.asType() + " " + p.getSimpleName())
      .collect(Collectors.joining(", "));
    return "(" + args + ")";
  }
}
```

The `Elements` object `elementUtils` is a collection of `static` tools; we use it to find the package name within `writeInterfaceFile()`.

The `getEnclosedElements()` method produces all the elements "enclosed" by a particular element. Here, the class encloses all its components. Using `getKind()` we find all `public` and `static` methods, and add those to the `interfaceMethods` list. Then `writeInterfaceFile()` uses that list to produce the new interface definition. Note the downcast to `ExecutableElement` in `writeInterfaceFile()`, which enables us to extract all the method information. `createArgList()` is a helper method to generate the argument list.

The `Filer` (produced by `getFiler()`) is a kind of `PrintWriter` which creates new files. The reason you use a `Filer` object, rather than a plain `PrintWriter`, is that it allows `javac` to keep track of any new files you create, so it can check them for annotations and compile them in an additional "round."

Here's the command line to compile using the processor:

```
javac -processor annotations.ifx.IfaceExtractorProcessor Multiplier.java
```

The generated `IMultiplier.java` file, as you might guess by looking at the `println()` statements in the processor above, looks like this:

```
package annotations.ifx;
public interface IMultiplier {
  public int multiply(int x, int y);
  public int fortySeven();
  public double timesTen(double arg);
}
```

This file is also compiled by `javac` (as part of the "rounds"), so you see the file `IMultiplier.class` in the same directory.

Annotation-Based Unit Testing

Unit testing is the practice of creating one or more tests for each method in a class, to regularly test the portions of a class for correct behavior. The most popular tool used for unit testing in Java is called *JUnit* (see Validating Your Code). JUnit version 4 incorporates annotations.[161] One of the main problems with pre-annotation versions of JUnit is the amount of "ceremony" necessary to set up and run JUnit tests. This has reduced over time, but annotations move testing closer to "the simplest unit testing system that can possibly work."

With pre-annotation versions of JUnit, you must create a separate class to hold your unit tests. With annotations we can include the unit tests inside the class to be tested, and thus reduce the time and trouble of unit testing to a minimum. This approach has the additional benefit of testing `private` methods as easily as `public` ones.

This example test framework is annotation-based so it's called `@Unit`. The most basic form of testing, and one which you will probably use much of the time, only needs the `@Test` annotation to indicate which methods should be tested. One option is for the test methods to take no arguments and return a `boolean` to indicate success or failure. You can use any name you like for test methods. Also, `@Unit` test methods can have any access that you'd like, including `private`.

To use `@Unit`, you import `onjava.atunit`, mark the appropriate methods and fields with `@Unit` test tags (which you'll learn about in the following examples), then have your build system run `@Unit` on the resulting class. Here's a simple example:

[161] I originally had thoughts of making a "better JUnit" based on the design shown here. However, it appears that JUnit4 also includes many of the ideas presented here, so it remains easier to go along with that.

```
// annotations/AtUnitExample1.java
// {java onjava.atunit.AtUnit
// build/classes/java/main/annotations/AtUnitExample1.class}
package annotations;
import onjava.atunit.*;
import onjava.*;

public class AtUnitExample1 {
  public String methodOne() {
    return "This is methodOne";
  }
  public int methodTwo() {
    System.out.println("This is methodTwo");
    return 2;
  }
  @Test
  boolean methodOneTest() {
    return methodOne().equals("This is methodOne");
  }
  @Test
  boolean m2() { return methodTwo() == 2; }
  @Test
  private boolean m3() { return true; }
  // Shows output for failure:
  @Test
  boolean failureTest() { return false; }
  @Test
  boolean anotherDisappointment() {
    return false;
  }
}
/* Output:
annotations.AtUnitExample1
  . anotherDisappointment (failed)
  . methodOneTest
  . failureTest (failed)
  . m2 This is methodTwo

  . m3
(5 tests)

>>> 2 FAILURES <<<
```

```
  annotations.AtUnitExample1: anotherDisappointment
  annotations.AtUnitExample1: failureTest
*/
```

Classes to be `@Unit` tested must be placed in packages.

The `@Test` annotation preceding the methods `methodOneTest()`, `m2()`, `m3()`, `failureTest()` and `anotherDisappointment()` tells `@Unit` to run these methods as unit tests. It will also ensure that those methods take no arguments and return a `boolean` or `void`. Your only responsibility when you write the unit test is to determine whether the test succeeds or fails and returns `true` or `false`, respectively (for methods that return `boolean`).

If you're familiar with JUnit, you'll also note `@Unit`'s more informative output—you see the test that's currently running so the output from that test is more useful, and at the end it tells you the classes and tests that caused failures.

You're not forced to embed test methods inside your classes, if that doesn't work for you. The easiest way to create non-embedded tests is with inheritance:

```
// annotations/AUExternalTest.java
// Creating non-embedded tests
// {java onjava.atunit.AtUnit
// build/classes/java/main/annotations/AUExternalTest.class}
package annotations;
import onjava.atunit.*;
import onjava.*;

public class
AUExternalTest extends AtUnitExample1 {
  @Test
  boolean tMethodOne() {
    return methodOne().equals("This is methodOne");
  }
  @Test
  boolean tMethodTwo() {
    return methodTwo() == 2;
  }
}
/* Output:
annotations.AUExternalTest
  . tMethodOne
```

```
  . tMethodTwo This is methodTwo

OK (2 tests)
*/
```

This example also demonstrates the value of flexible naming. Here, `@Test` methods that directly test another method are given the name of that method starting with an underscore (I'm not suggesting this is an ideal style, just showing a possibility).

You can also use composition to create non-embedded tests:

```
// annotations/AUComposition.java
// Creating non-embedded tests
// {java onjava.atunit.AtUnit
// build/classes/java/main/annotations/AUComposition.class}
package annotations;
import onjava.atunit.*;
import onjava.*;

public class AUComposition {
  AtUnitExample1 testObject = new AtUnitExample1();
  @Test
  boolean tMethodOne() {
    return testObject.methodOne()
      .equals("This is methodOne");
  }
  @Test
  boolean tMethodTwo() {
    return testObject.methodTwo() == 2;
  }
}
/* Output:
annotations.AUComposition
  . tMethodOne
  . tMethodTwo This is methodTwo

OK (2 tests)
*/
```

A new member `testObject` is created for each test because an `AUComposition` object is created for each test.

There are no special "assert" methods as there are in JUnit, but the second form of the `@Test` method returns `void` (or `boolean`, if you still want to return `true` or `false` here). To test for success, you can use Java `assert` statements. Java assertions are normally enabled with the `-ea` flag on the `java` command line, but `@Unit` automatically enables them. To indicate failure, you can even use an exception. One of the `@Unit` design goals is to require as little additional syntax as possible, and Java's `assert` and exceptions are all that is necessary to report errors. A failed `assert` or an exception that emerges from the test method is treated as a failed test, but `@Unit` does not halt here—it continues until all the tests are run. Here's an example:

```
// annotations/AtUnitExample2.java
// Assertions and exceptions can be used in @Tests
// {java onjava.atunit.AtUnit
// build/classes/java/main/annotations/AtUnitExample2.class}
package annotations;
import java.io.*;
import onjava.atunit.*;
import onjava.*;

public class AtUnitExample2 {
  public String methodOne() {
    return "This is methodOne";
  }
  public int methodTwo() {
    System.out.println("This is methodTwo");
    return 2;
  }
  @Test
  void assertExample() {
    assert methodOne().equals("This is methodOne");
  }
  @Test
  void assertFailureExample() {
    assert 1 == 2: "What a surprise!";
  }
  @Test
  void exceptionExample() throws IOException {
    try(FileInputStream fis =
        new FileInputStream("nofile.txt")) {} // Throws
  }
```

```
  @Test
  boolean assertAndReturn() {
    // Assertion with message:
    assert methodTwo() == 2: "methodTwo must equal 2";
    return methodOne().equals("This is methodOne");
  }
}
/* Output:
annotations.AtUnitExample2
  . assertFailureExample java.lang.AssertionError: What
a surprise!
(failed)
  . assertExample
  . exceptionExample java.io.FileNotFoundException:
nofile.txt (The system cannot find the file specified)
(failed)
  . assertAndReturn This is methodTwo

(4 tests)

>>> 2 FAILURES <<<
  annotations.AtUnitExample2: assertFailureExample
  annotations.AtUnitExample2: exceptionExample
*/
```

Here's an example using non-embedded tests with assertions, performing some simple tests of `java.util.HashSet`:

```
// annotations/HashSetTest.java
// {java onjava.atunit.AtUnit
// build/classes/java/main/annotations/HashSetTest.class}
package annotations;
import java.util.*;
import onjava.atunit.*;
import onjava.*;

public class HashSetTest {
  HashSet<String> testObject = new HashSet<>();
  @Test
  void initialization() {
    assert testObject.isEmpty();
  }
```

```
  @Test
  void tContains() {
    testObject.add("one");
    assert testObject.contains("one");
  }
  @Test
  void tRemove() {
    testObject.add("one");
    testObject.remove("one");
    assert testObject.isEmpty();
  }
}
/* Output:
annotations.HashSetTest
  . tContains
  . initialization
  . tRemove
OK (3 tests)
*/
```

The inheritance approach seems simpler, in the absence of other constraints.

For each unit test, `@Unit` creates an object of the class under test using the zero-argument constructor. The test is called for that object, then the object is discarded to prevent side effects from leaking into other unit tests. This relies on the zero-argument constructor to create the objects. If you don't have a zero-argument constructor or you need more sophisticated construction for objects, create a `static` method to build the object and attach the `@TestObjectCreate` annotation, like this:

```
// annotations/AtUnitExample3.java
// {java onjava.atunit.AtUnit
// build/classes/java/main/annotations/AtUnitExample3.class}
package annotations;
import onjava.atunit.*;
import onjava.*;

public class AtUnitExample3 {
  private int n;
  public AtUnitExample3(int n) { this.n = n; }
  public int getN() { return n; }
  public String methodOne() {
    return "This is methodOne";
```

```
  }
  public int methodTwo() {
    System.out.println("This is methodTwo");
    return 2;
  }
  @TestObjectCreate
  static AtUnitExample3 create() {
    return new AtUnitExample3(47);
  }
  @Test
  boolean initialization() { return n == 47; }
  @Test
  boolean methodOneTest() {
    return methodOne().equals("This is methodOne");
  }
  @Test
  boolean m2() { return methodTwo() == 2; }
}
/* Output:
annotations.AtUnitExample3
  . initialization
  . methodOneTest
  . m2 This is methodTwo

OK (3 tests)
*/
```

The `@TestObjectCreate` method must be `static` and must return an object of the type that you're testing—the `@Unit` program will ensure this is true.

Sometimes you need additional fields to support unit testing. The `@TestProperty` annotation can tag fields that are only used for unit testing (so they can be optionally removed before you deliver the product to the client). Here's an example that reads values from a `String` that is broken up using the `String.split()` method. This input is used to produce test objects:

```
// annotations/AtUnitExample4.java
// {java onjava.atunit.AtUnit
// build/classes/java/main/annotations/AtUnitExample4.class}
// {VisuallyInspectOutput}
package annotations;
import java.util.*;
import onjava.atunit.*;
import onjava.*;

public class AtUnitExample4 {
  static String theory = "All brontosauruses " +
    "are thin at one end, much MUCH thicker in the " +
    "middle, and then thin again at the far end.";
  private String word;
  private Random rand = new Random(); // Time-based seed
  public AtUnitExample4(String word) {
    this.word = word;
  }
  public String getWord() { return word; }
  public String scrambleWord() {
    List<Character> chars = Arrays.asList(
      ConvertTo.boxed(word.toCharArray()));
    Collections.shuffle(chars, rand);
    StringBuilder result = new StringBuilder();
    for(char ch : chars)
      result.append(ch);
    return result.toString();
  }
  @TestProperty
  static List<String> input =
    Arrays.asList(theory.split(" "));
  @TestProperty
  static Iterator<String> words = input.iterator();
  @TestObjectCreate
  static AtUnitExample4 create() {
    if(words.hasNext())
      return new AtUnitExample4(words.next());
    else
      return null;
  }
  @Test
  boolean words() {
```

```
    System.out.println("'" + getWord() + "'");
    return getWord().equals("are");
  }
  @Test
  boolean scramble1() {
    // Use specific seed to get verifiable results:
    rand = new Random(47);
    System.out.println("'" + getWord() + "'");
    String scrambled = scrambleWord();
    System.out.println(scrambled);
    return scrambled.equals("lAl");
  }
  @Test
  boolean scramble2() {
    rand = new Random(74);
    System.out.println("'" + getWord() + "'");
    String scrambled = scrambleWord();
    System.out.println(scrambled);
    return scrambled.equals("tsaeborornussu");
  }
}
/* Output:
annotations.AtUnitExample4
  . words 'All'
(failed)
  . scramble1 'brontosauruses'
ntsaueorosurbs
(failed)
  . scramble2 'are'
are
(failed)
(3 tests)

>>> 3 FAILURES <<<
  annotations.AtUnitExample4: words
  annotations.AtUnitExample4: scramble1
  annotations.AtUnitExample4: scramble2
*/
```

`@TestProperty` can also be used to tag methods that can be used during testing, but are not tests themselves.

This program relies on the execution order of the tests, which is in general not a good practice.

If your test object creation performs initialization that requires later cleanup, you can optionally add a `static` `@TestObjectCleanup` method to perform cleanup when you are finished with the test object. In this next example, `@TestObjectCreate` opens a file to create each test object, so the file must be closed before the test object is discarded:

```
// annotations/AtUnitExample5.java
// {java onjava.atunit.AtUnit
// build/classes/java/main/annotations/AtUnitExample5.class}
package annotations;
import java.io.*;
import onjava.atunit.*;
import onjava.*;

public class AtUnitExample5 {
  private String text;
  public AtUnitExample5(String text) {
    this.text = text;
  }
  @Override public String toString() { return text; }
  @TestProperty
  static PrintWriter output;
  @TestProperty
  static int counter;
  @TestObjectCreate
  static AtUnitExample5 create() {
    String id = Integer.toString(counter++);
    try {
      output = new PrintWriter("Test" + id + ".txt");
    } catch(IOException e) {
      throw new RuntimeException(e);
    }
    return new AtUnitExample5(id);
  }
  @TestObjectCleanup
  static void cleanup(AtUnitExample5 tobj) {
    System.out.println("Running cleanup");
    output.close();
  }
```

```
  @Test
  boolean test1() {
    output.print("test1");
    return true;
  }
  @Test
  boolean test2() {
    output.print("test2");
    return true;
  }
  @Test
  boolean test3() {
    output.print("test3");
    return true;
  }
}
/* Output:
annotations.AtUnitExample5
  . test1
Running cleanup
  . test3
Running cleanup
  . test2
Running cleanup
OK (3 tests)
*/
```

The output shows that the cleanup method is automatically run after each test.

Using `@Unit` with Generics

Generics pose a special problem, because you can't "test generically." You must test for a specific type parameter or set of parameters. The solution is simple: Inherit a test class from a specified version of the generic class.

Here's a simple implementation of a stack:

```
// annotations/StackL.java
// A stack built on a LinkedList
package annotations;
import java.util.*;

public class StackL<T> {
  private LinkedList<T> list = new LinkedList<>();
  public void push(T v) { list.addFirst(v); }
  public T top() { return list.getFirst(); }
  public T pop() { return list.removeFirst(); }
}
```

To test a `String` version, inherit a test class from `StackL<String>`:

```
// annotations/StackLStringTst.java
// Applying @Unit to generics
// {java onjava.atunit.AtUnit
// build/classes/java/main/annotations/StackLStringTst.class}
package annotations;
import onjava.atunit.*;
import onjava.*;

public class
StackLStringTst extends StackL<String> {
  @Test
  void tPush() {
    push("one");
    assert top().equals("one");
    push("two");
    assert top().equals("two");
  }
  @Test
  void tPop() {
    push("one");
    push("two");
    assert pop().equals("two");
    assert pop().equals("one");
  }
  @Test
  void tTop() {
    push("A");
    push("B");
```

```
    assert top().equals("B");
    assert top().equals("B");
  }
}
/* Output:
annotations.StackLStringTst
  . tPop
  . tTop
  . tPush
OK (3 tests)
*/
```

The only potential drawback to inheritance is that you lose the ability to access `private` methods in the class under test. If this is a problem, you can either make the method in question `protected`, or add a non-private `@TestProperty` method that calls the `private` method (the `@TestProperty` method will then be stripped out of the production code by the `AtUnitRemover` tool that is shown later in this chapter).

`@Unit` searches for class files containing the appropriate annotations, then executes the `@Test` methods. Much of my goal with the `@Unit` testing system is to make it incredibly transparent, so people can begin using it by adding `@Test` methods, with no other special code or knowledge (modern versions of JUnit follow this practice). It's hard enough to write tests without adding any new hurdles, so `@Unit` tries to make it trivial. This way, you're more likely to actually write the tests.

Implementing `@Unit`

First, we define all the annotation types. These are simple tags, and have no fields. The `@Test` tag was defined at the beginning of the chapter, and here are the rest of the annotations:

```
// onjava/atunit/TestObjectCreate.java
// The @Unit @TestObjectCreate tag
package onjava.atunit;
import java.lang.annotation.*;

@Target(ElementType.METHOD)
@Retention(RetentionPolicy.RUNTIME)
public @interface TestObjectCreate {}
```

```
// onjava/atunit/TestObjectCleanup.java
// The @Unit @TestObjectCleanup tag
package onjava.atunit;
import java.lang.annotation.*;

@Target(ElementType.METHOD)
@Retention(RetentionPolicy.RUNTIME)
public @interface TestObjectCleanup {}
```

```
// onjava/atunit/TestProperty.java
// The @Unit @TestProperty tag
package onjava.atunit;
import java.lang.annotation.*;

// Both fields and methods can be tagged as properties:
@Target({ElementType.FIELD, ElementType.METHOD})
@Retention(RetentionPolicy.RUNTIME)
public @interface TestProperty {}
```

All the tests have `RUNTIME` retention because the `@Unit` system must discover the tests in compiled code.

To implement the system that runs the tests, we use reflection to extract the annotations. The program uses this information to decide how to build the test objects and run tests on them. Annotations make the result surprisingly small and straightforward:

```
// onjava/atunit/AtUnit.java
// An annotation-based unit-test framework
// {java onjava.atunit.AtUnit}
package onjava.atunit;
import java.lang.reflect.*;
import java.io.*;
import java.util.*;
import java.nio.file.*;
import java.util.stream.*;
import onjava.*;

public class AtUnit implements ProcessFiles.Strategy {
  static Class<?> testClass;
  static List<String> failedTests= new ArrayList<>();
  static long testsRun = 0;
  static long failures = 0;
  public static void
  main(String[] args) throws Exception {
    ClassLoader.getSystemClassLoader()
      .setDefaultAssertionStatus(true); // Enable assert
    new ProcessFiles(new AtUnit(), "class").start(args);
    if(failures == 0)
      System.out.println("OK (" + testsRun + " tests)");
    else {
      System.out.println("(" + testsRun + " tests)");
      System.out.println(
        "\n>>> " + failures + " FAILURE" +
        (failures > 1 ? "S" : "") + " <<<");
      for(String failed : failedTests)
        System.out.println("  " + failed);
    }
  }
  @Override public void process(File cFile) {
    try {
      String cName = ClassNameFinder.thisClass(
        Files.readAllBytes(cFile.toPath()));
      if(!cName.startsWith("public:"))
        return;
      cName = cName.split(":")[1];
      if(!cName.contains("."))
        return; // Ignore unpackaged classes
      testClass = Class.forName(cName);
```

```
    } catch(IOException | ClassNotFoundException e) {
      throw new RuntimeException(e);
    }
    TestMethods testMethods = new TestMethods();
    Method creator = null;
    Method cleanup = null;
    for(Method m : testClass.getDeclaredMethods()) {
      testMethods.addIfTestMethod(m);
      if(creator == null)
        creator = checkForCreatorMethod(m);
      if(cleanup == null)
        cleanup = checkForCleanupMethod(m);
    }
    if(testMethods.size() > 0) {
      if(creator == null)
        try {
          if(!Modifier.isPublic(testClass
              .getDeclaredConstructor()
              .getModifiers())) {
            System.out.println("Error: " + testClass +
              " zero-argument constructor must be public");
            System.exit(1);
          }
        } catch(NoSuchMethodException e) {
          // Synthesized zero-argument constructor; OK
        }
      System.out.println(testClass.getName());
    }
    for(Method m : testMethods) {
      System.out.print("  . " + m.getName() + " ");
      try {
        Object testObject = createTestObject(creator);
        boolean success = false;
        try {
          if(m.getReturnType().equals(boolean.class))
            success = (Boolean)m.invoke(testObject);
          else {
            m.invoke(testObject);
            success = true; // If no assert fails
          }
        } catch(InvocationTargetException e) {
          // Actual exception is inside e:
```

```
          System.out.println(e.getCause());
        }
        System.out.println(success ? "" : "(failed)");
        testsRun++;
        if(!success) {
          failures++;
          failedTests.add(testClass.getName() +
            ": " + m.getName());
        }
        if(cleanup != null)
          cleanup.invoke(testObject, testObject);
      } catch(IllegalAccessException |
              IllegalArgumentException |
              InvocationTargetException e) {
        throw new RuntimeException(e);
      }
    }
  }
  public static
  class TestMethods extends ArrayList<Method> {
    void addIfTestMethod(Method m) {
      if(m.getAnnotation(Test.class) == null)
        return;
      if(!(m.getReturnType().equals(boolean.class) ||
           m.getReturnType().equals(void.class)))
        throw new RuntimeException("@Test method" +
          " must return boolean or void");
      m.setAccessible(true); // If it's private, etc.
      add(m);
    }
  }
  private static
  Method checkForCreatorMethod(Method m) {
    if(m.getAnnotation(TestObjectCreate.class) == null)
      return null;
    if(!m.getReturnType().equals(testClass))
      throw new RuntimeException("@TestObjectCreate " +
        "must return instance of Class to be tested");
    if((m.getModifiers() &
        java.lang.reflect.Modifier.STATIC) < 1)
      throw new RuntimeException("@TestObjectCreate " +
        "must be static.");
```

```
    m.setAccessible(true);
    return m;
  }
  private static
  Method checkForCleanupMethod(Method m) {
    if(m.getAnnotation(TestObjectCleanup.class) == null)
      return null;
    if(!m.getReturnType().equals(void.class))
      throw new RuntimeException("@TestObjectCleanup " +
        "must return void");
    if((m.getModifiers() &
        java.lang.reflect.Modifier.STATIC) < 1)
      throw new RuntimeException("@TestObjectCleanup " +
        "must be static.");
    if(m.getParameterTypes().length == 0 ||
       m.getParameterTypes()[0] != testClass)
      throw new RuntimeException("@TestObjectCleanup " +
        "must take an argument of the tested type.");
    m.setAccessible(true);
    return m;
  }
  private static Object
  createTestObject(Method creator) {
    if(creator != null) {
      try {
        return creator.invoke(testClass);
      } catch(IllegalAccessException |
              IllegalArgumentException |
              InvocationTargetException e) {
        throw new RuntimeException("Couldn't run " +
          "@TestObject (creator) method.");
      }
    } else { // Use the zero-argument constructor:
      try {
        return testClass
          .getConstructor().newInstance();
      } catch(InstantiationException |
              NoSuchMethodException |
              InvocationTargetException |
              IllegalAccessException e) {
        throw new RuntimeException(
          "Couldn't create a test object. " +
```

```
          "Try using a @TestObject method.");
      }
    }
  }
}
```

Although it might be "premature refactoring," (because it's only used once in the book) `AtUnit.java` uses another tool called `ProcessFiles` to step through each argument on the command line, decide whether it's a directory or a file, and act accordingly. It can be applied to different solutions because it contains a `Strategy` interface for customization:

```
// onjava/ProcessFiles.java
package onjava;
import java.io.*;
import java.nio.file.*;

public class ProcessFiles {
  public interface Strategy {
    void process(File file);
  }
  private Strategy strategy;
  private String ext;
  public ProcessFiles(Strategy strategy, String ext) {
    this.strategy = strategy;
    this.ext = ext;
  }
  public void start(String[] args) {
    try {
      if(args.length == 0)
        processDirectoryTree(new File("."));
      else
        for(String arg : args) {
          File fileArg = new File(arg);
          if(fileArg.isDirectory())
            processDirectoryTree(fileArg);
          else {
            // Allow user to leave off extension:
            if(!arg.endsWith("." + ext))
              arg += "." + ext;
            strategy.process(
              new File(arg).getCanonicalFile());
```

```
          }
        }
    } catch(IOException e) {
      throw new RuntimeException(e);
    }
  }
  public void
  processDirectoryTree(File root) throws IOException {
    PathMatcher matcher = FileSystems.getDefault()
      .getPathMatcher("glob:**/*.{" + ext + "}");
    Files.walk(root.toPath())
      .filter(matcher::matches)
      .forEach(p -> strategy.process(p.toFile()));
  }
}
```

The `AtUnit` class implements `ProcessFiles.Strategy`, containing the method `process()`. This way, an instance of `AtUnit` can be passed to the `ProcessFiles` constructor. The second constructor argument tells `ProcessFiles` to look for all files that have "`class`" extensions.

Here's a simple usage example:

```
// annotations/DemoProcessFiles.java
import onjava.ProcessFiles;

public class DemoProcessFiles {
  public static void main(String[] args) {
    new ProcessFiles(file -> System.out.println(file),
      "java").start(args);
  }
}
/* Output:
.\AtUnitExample1.java
.\AtUnitExample2.java
.\AtUnitExample3.java
.\AtUnitExample4.java
.\AtUnitExample5.java
.\AUComposition.java
.\AUExternalTest.java
.\database\Constraints.java
.\database\DBTable.java
```

```
.\database\Member.java
.\database\SQLInteger.java
.\database\SQLString.java
.\database\TableCreator.java
.\database\Uniqueness.java
.\DemoProcessFiles.java
.\HashSetTest.java
.\ifx\ExtractInterface.java
.\ifx\IfaceExtractorProcessor.java
.\ifx\Multiplier.java
.\PasswordUtils.java
.\simplest\Simple.java
.\simplest\SimpleProcessor.java
.\simplest\SimpleTest.java
.\SimulatingNull.java
.\StackL.java
.\StackLStringTst.java
.\Testable.java
.\UseCase.java
.\UseCaseTracker.java
*/
```

With no command-line argument, the program traverses the current directory tree. You can also provide multiple arguments which can be either class files (with or without the `.class` extension) or directories.

Returning to our discussion of `AtUnit.java`, `@Unit` automatically finds the testable classes and methods so no "suite" mechanism is necessary.

One of the problems `AtUnit.java` must solve when it discovers class files is that the qualified class name (including package) is not evident from the class file name. To discover this information, the class file must be analyzed—not trivial, but not impossible, either.[162] When a `.class` file is found, it is opened and its binary data is read and handed to `ClassNameFinder.thisClass()`. Here, we are moving into the realm of "bytecode engineering," because we are actually analyzing the contents of a class file:

[162] Jeremy Meyer and I spent most of a day figuring this out.

```
// onjava/atunit/ClassNameFinder.java
// {java onjava.atunit.ClassNameFinder}
package onjava.atunit;
import java.io.*;
import java.nio.file.*;
import java.util.*;
import onjava.*;

public class ClassNameFinder {
  public static String thisClass(byte[] classBytes) {
    Map<Integer,Integer> offsetTable = new HashMap<>();
    Map<Integer,String> classNameTable =
      new HashMap<>();
    try {
      DataInputStream data = new DataInputStream(
        new ByteArrayInputStream(classBytes));
      int magic = data.readInt();  // 0xcafebabe
      int minorVersion = data.readShort();
      int majorVersion = data.readShort();
      int constantPoolCount = data.readShort();
      int[] constantPool = new int[constantPoolCount];
      for(int i = 1; i < constantPoolCount; i++) {
        int tag = data.read();
        // int tableSize;
        switch(tag) {
          case 1: // UTF
            int length = data.readShort();
            char[] bytes = new char[length];
            for(int k = 0; k < bytes.length; k++)
              bytes[k] = (char)data.read();
            String className = new String(bytes);
            classNameTable.put(i, className);
            break;
          case 5: // LONG
          case 6: // DOUBLE
            data.readLong(); // discard 8 bytes
            i++; // Special skip necessary
            break;
          case 7: // CLASS
            int offset = data.readShort();
            offsetTable.put(i, offset);
            break;
```

```
          case 8: // STRING
            data.readShort(); // discard 2 bytes
            break;
          case 3:  // INTEGER
          case 4:  // FLOAT
          case 9:  // FIELD_REF
          case 10: // METHOD_REF
          case 11: // INTERFACE_METHOD_REF
          case 12: // NAME_AND_TYPE
          case 18: // Invoke Dynamic
            data.readInt(); // discard 4 bytes
            break;
          case 15: // Method Handle
            data.readByte();
            data.readShort();
            break;
          case 16: // Method Type
            data.readShort();
            break;
          default:
            throw
              new RuntimeException("Bad tag " + tag);
        }
      }
      short accessFlags = data.readShort();
      String access = (accessFlags & 0x0001) == 0 ?
        "nonpublic:" : "public:";
      int thisClass = data.readShort();
      int superClass = data.readShort();
      return access + classNameTable.get(
        offsetTable.get(thisClass)).replace('/', '.');
    } catch(IOException | RuntimeException e) {
      throw new RuntimeException(e);
    }
  }
  // Demonstration:
  public static void
  main(String[] args) throws Exception {
    PathMatcher matcher = FileSystems.getDefault()
      .getPathMatcher("glob:**/*.class");
    // Walk the entire tree:
    Files.walk(Paths.get("."))
```

```
      .filter(matcher::matches)
      .map(p -> {
          try {
            return thisClass(Files.readAllBytes(p));
          } catch(Exception e) {
            throw new RuntimeException(e);
          }
        })
      .filter(s -> s.startsWith("public:"))
      // .filter(s -> s.indexOf('$') >= 0)
      .map(s -> s.split(":")[1])
      .filter(s -> !s.startsWith("enums."))
      .filter(s -> s.contains("."))
      .forEach(System.out::println);
  }
}
/* Output:
onjava.ArrayShow
onjava.atunit.AtUnit$TestMethods
onjava.atunit.AtUnit
onjava.atunit.ClassNameFinder
onjava.atunit.Test
onjava.atunit.TestObjectCleanup
onjava.atunit.TestObjectCreate
onjava.atunit.TestProperty
onjava.BasicSupplier
onjava.CollectionMethodDifferences
onjava.ConvertTo
onjava.Count$Boolean
onjava.Count$Byte
onjava.Count$Character
onjava.Count$Double
onjava.Count$Float
onjava.Count$Integer
onjava.Count$Long
onjava.Count$Pboolean
onjava.Count$Pbyte
onjava.Count$Pchar
onjava.Count$Pdouble
onjava.Count$Pfloat
onjava.Count$Pint
onjava.Count$Plong
```

onjava.Count$Pshort
onjava.Count$Short
onjava.Count
onjava.CountingIntegerList
onjava.CountMap
onjava.Countries
onjava.Enums
onjava.FillMap
onjava.HTMLColors
onjava.MouseClick
onjava.Nap
onjava.Null
onjava.Operations
onjava.OSExecute
onjava.OSExecuteException
onjava.Pair
onjava.ProcessFiles$Strategy
onjava.ProcessFiles
onjava.Rand$Boolean
onjava.Rand$Byte
onjava.Rand$Character
onjava.Rand$Double
onjava.Rand$Float
onjava.Rand$Integer
onjava.Rand$Long
onjava.Rand$Pboolean
onjava.Rand$Pbyte
onjava.Rand$Pchar
onjava.Rand$Pdouble
onjava.Rand$Pfloat
onjava.Rand$Pint
onjava.Rand$Plong
onjava.Rand$Pshort
onjava.Rand$Short
onjava.Rand$String
onjava.Rand
onjava.Range
onjava.Repeat
onjava.RmDir
onjava.Sets
onjava.Stack
onjava.Suppliers

```
onjava.TimedAbort
onjava.Timer
onjava.Tuple
onjava.Tuple2
onjava.Tuple3
onjava.Tuple4
onjava.Tuple5
onjava.TypeCounter
*/
```

Although it's not possible to go into full detail here, each class file follows a particular format and I've tried to use meaningful field names for the pieces of data picked out of the `ByteArrayInputStream`; you can also see the size of each piece by the length of the read performed on the input stream. For example, the first 32 bits of any class file is always the "magic number" hex 0xcafebabe,[163] and the next two `shorts` are version information. The constant pool contains the constants for the program and so is of variable size; the next `short` tells how big it is, so an appropriate-sized array can be allocated. Each entry in the constant pool can be a fixed-size or variable-sized value, so we must examine the tag that begins each one to find out what to do with it—that's the `switch` statement. Here, we are not trying to accurately analyze all the data in the class file, but merely to step through and store the pieces of interest, so you'll notice that a fair amount of data is discarded. Information about classes is stored in the `classNameTable` and the `offsetTable`. After the constant pool is read, the `thisClass` information is found. It is an index into the `offsetTable`, which produces an index into the `classNameTable`, which produces the class name.

Back in `AtUnit.java`, `process()` now has the class name and can look to see if it contains a `.`, which means it's in a package. Unpackaged classes are ignored. If a class is in a package, the standard class loader is used to load the class with `Class.forName()`. Now the class can be analyzed for `@Unit` annotations.

We only look for three things: `@Test` methods, which are stored in a `TestMethods` list, and whether there's an `@TestObjectCreate` and `@TestObjectCleanup` method. These are discovered through the associated method calls you see in the code, which look for the annotations.

If any `@Test` methods are found, the name of the class is displayed so the viewer can see what's happening, then each test is executed. This means printing the method

[163] Various legends surround the meaning of this, but because Java was predominantly created by male nerds we can make a reasonable guess it had something to do with fantasizing about a woman in a coffee shop.

name, then calling `createTestObject()`, which will use the `@TestObjectCreate` method if one exists, or will fall back to the zero-argument constructor otherwise. Once the test object is created, the test method is invoked upon that object. If the test returns a `boolean`, the result is captured. If not, we assume success if there is no exception (which would happen in the case of a failed `assert` or any other kind of exception). If an exception is thrown, the exception information is printed to show the cause. If any failure occurs, the failure count is increased and the class name and method are added to `failedTests` so these can be reported at the end of the run.

Summary

Annotations are a welcome addition to Java. They are a structured and type-checked means of adding metadata to your code without rendering it unreadable and messy. They can help remove the tedium of writing deployment descriptors and other generated files. The fact that the `@deprecated` Javadoc tag is superseded by the `@Deprecated` annotation is just one indication of how much better suited annotations are for describing information about code components than are comments.

Only a small handful of annotations come with Java. This means if you can't find a library elsewhere, you create annotations and the associated logic. With annotation processors attached to `javac`, you can compile newly generated files in one step, easing the build process.

Providers of APIs and frameworks will start including annotations as part of their toolkits. As you can imagine by seeing the `@Unit` system, it is very likely that annotations will cause significant changes in our Java programming experience.

Concurrent Programming

> "But I don't want to go among mad people," Alice remarked. "Oh, you can't help that," said the Cat. "We're all mad here. I'm mad. You're mad." "How do you know I'm mad?" said Alice. "You must be," said the Cat, "or you wouldn't have come here." —*Alice's Adventures in Wonderland*, Chapter 6.

Up to this point we've been programming in a fashion much like the *stream-of-consciousness* narrative device in literature: first one thing happens, then the next. We're in complete control of all the steps and the order they occur. It would be very surprising if we were to set a value to 5, then at some point later come back and find it was 47.

We now enter the strange world of concurrency, where this result is not surprising at all. Everything you're comfortable believing is no longer reliable. It might work and it might not. Most likely it will work under some conditions and not in others, and you'll have to know and understand these situations to determine what works.

As an analogy, your normal life takes place in the world of Newtonian Mechanics. Objects have mass: they fall and transfer their momentum. Wires have resistance, and light travels in straight lines. But if you enter the world of the very small, very hot, very cold or very massive (where we can't exist) things change. We can't tell whether something is a particle or a wave, light is affected by gravity, and some things become superconductors.

Rather than a single stream-of-consciousness narrative, we're inside a spy novel that has numerous stories running at the same time, one for each character. One spy leaves microfilm under a special rock, and when the second spy comes to retrieve the package, it might already have been taken by a third spy. But this particular novel doesn't neatly tie things up; you can easily get to the end and never figure out what happens.

Building concurrent applications is much like the game Jenga[164], where every time

[164] https://en.wikipedia.org/wiki/Jenga

you pull out a block and place it on the tower, the whole thing can come crashing down. Every tower, and every application, is unique, with its own requirements. What you learn from building one system might not apply to the next one.

This chapter is a very basic introduction to concurrency. Although I use the most modern Java 8 tools available to demonstrate the principles, the chapter is far from a comprehensive treatment of the topic. My goal is to give you enough of the fundamentals that you can grasp the complexity—and danger—of the issues, to engender a healthy respect for the difficulty of wading into these shark-infested waters.

For more of the messy, low-level details, see the Appendix: Low-Level Concurrency. To venture further into this domain, you must also read *Java Concurrency in Practice* by Brian Goetz et. al. Although at this writing, that book is over ten years old, it still contains essentials you must know and understand. Ideally, this chapter and the appendix is a good preparation for that book. Another valuable resource is Bill Venners' *Inside the Java Virtual Machine*, which describes in detail the innermost workings of the JVM, including threads.

The Terminology Problem

The terms *concurrent*, *parallel*, *multitasking*, *multiprocessing*, *multithreading*, *distributed systems* (and probably others) are used in numerous conflicting ways throughout programming literature, and are often conflated. After pointing this out in his 2016 presentation From Concurrent to Parallel[165], Brian Goetz suggests a reasonable dichotomy:

- Concurrency is about correctly and efficiently controlling access to shared resources.
- Parallelism uses additional resources to produce an answer faster.

These are good definitions, but there are decades of confusion-producing history that fight against fixing the problem. In general, when people use the word "concurrency," they mean "everything, the entire mess," and I'll probably fall into that practice myself in many places—indeed, most books, including Brian Goetz' *Java Concurrency in Practice*, use the word in the title.

[165] https://www.youtube.com/watch?v=NsDE7E8sIdQ

Concurrency often means "more than one task is making progress," while parallelism almost always means "more than one task is executing simultaneously." You can immediately see the problem with these definitions: parallelism also has more than one task "making progress." The distinction is the details, in exactly *how* that "progress" is happening. Also, the overlap: a program written for parallelism can still sometimes run on a single processor, while some concurrent-programming systems can take advantage of more than one processor.

Here's another approach, writing the definitions around where the slowdown occurs:

Concurrency

Accomplishing more than one task at the same time. One task doesn't need to complete before you start working on other tasks. Concurrency solves problems where *blocking* occurs—when a task can't progress further until something outside its control changes. The most common example is I/O, where a task must wait for some input (in which case it is said to be *blocked*). A problem like this is said to be *I/O bound.*

Parallelism

Accomplishing more than one task *in multiple places* at the same time. This solves so-called *compute-bound* problems, where a program can run faster if you split it into multiple parts and run those different parts on different processors.

The reason the terminology is confusing is shown in the definitions above: the core of both is "accomplishing more than one task at the same time." Parallelism adds distribution across multiple processors. More importantly, the two solve different types of problems: taking an I/O-bound problem and parallelizing might not do you any good because the problem is not overall speed, it's blocking. And taking a compute-bound problem and trying to solve it using concurrency on a single processor might be a similar waste of time. Both approaches try to accomplish more in less time, but the way they achieve speedup is different, and depends upon constraints imposed by the problem.

A major reason that the two concepts get mixed together is that many programming languages including Java use the same mechanism—the *thread*—to implement both concurrency and parallelism.

We can even try to add more granularity to the definitions (however, this is not standardized terminology):

- **Purely Concurrent**: Tasks still run on a single CPU. A purely concurrent system produces results faster than a sequential system, but doesn't run any faster if there are more processors.
- **Concurrent-Parallel**: Using concurrency techniques, the resulting program takes advantage of more processors and produces results faster.
- **Parallel-Concurrent**: Written using parallel programming techniques, the resulting program can still run if there is only a single processor (Java 8 `Streams` are a good example).
- **Purely Parallel**: Won't run unless there is more than one processor.

This might be a useful taxonomy in some situations.

Language and library support for concurrency seem like perfect candidates for the term Leaky Abstraction[166]. The goal of an abstraction is to "abstract away" pieces that are not essential to the idea at hand, to shield you from needless detail. If the abstraction is leaky, those pieces and details keep re-asserting themselves as important, regardless of how much you try to hide them.

I've started to wonder whether there's really any abstraction at all. When writing these kinds of programs you are never shielded from any of the underlying systems and tools, even details about how the CPU cache works. Ultimately, if you've been very careful, what you create works in a particular situation, but it won't work in other situations. Sometimes the difference is the way two machines are configured, or the estimated load for the program. This is not specific to Java per se—it is the nature of concurrent and parallel programming.

You might argue that a pure functional[167] language doesn't have these restrictions. Indeed, a pure functional language solves a large number of concurrency problems, so if you are tackling a difficult concurrency problem you might consider writing that portion in a pure functional language. But ultimately, if you write a system that uses a queue, for example, if it isn't tuned properly and the input rate either isn't estimated correctly or throttled (and throttling means different things and has different impacts in different situations), that queue will either fill up and block, or overflow. In the end, you must understand all the details, and any issue can break your system. It's a very different kind of programming.

[166] https://en.wikipedia.org/wiki/Leaky_abstraction
[167] https://en.wikipedia.org/wiki/Purely_functional

A New Definition of Concurrency

For decades, I have periodically grappled with concurrency in various forms, and one of the biggest challenges has always been simply defining it. While writing this chapter, I finally had an insight which I think captures it:

> Concurrency is a collection of performance techniques focused on the reduction of waiting.

This is actually a rather dense statement, so I'll break it down:

- It's a *collection*: there are many different approaches to solving the problem. This is one of the issues that makes defining concurrency so challenging, because the techniques vary widely.
- These are *performance techniques*: That's it. The whole point of concurrency is to get your program to run faster. In Java, concurrency is very tricky and difficult, so absolutely do not use it unless you have a significant performance problem—and even then, use the easiest approach that produces the performance you need, because concurrency rapidly becomes unmanageable.
- The "reduction of waiting" part is important and subtle. Regardless of (for example) how many processors you are running on, you can only produce a benefit when some kind of waiting is taking place. If you ask for I/O and instantly get a result, there's no delay and thus nothing to improve. If you are running multiple tasks on multiple processors and each is running at full capacity and no task is waiting on any other, there's no point in trying to increase your throughput. The only opportunity for concurrency is if some part of your program is forced to wait. That waiting can appear in many forms—which explains why there are so many different approaches to concurrency.

It's worth emphasizing that the effectiveness of this definition hinges on the word *waiting*. If nothing is waiting there's no opportunity for speedups. And if something is waiting, there are numerous approaches to speeding things up and these depend on multiple factors including the configuration of the system where it's running, the type of problem you're solving, and any number of other issues.

Concurrency Superpowers

Imagine you're inside a science-fiction movie. You must search a tall building for a single item that is carefully and cleverly hidden in one of the ten million rooms of the building. You enter the building and move down a corridor. The corridor divides.

By yourself it will take a hundred lifetimes to accomplish this task.

Now suppose you have a strange superpower. You can split yourself in two, and send one of yourself down one corridor while you continue down the other. Every time you encounter a divide in a corridor or a staircase to the next level, you repeat this splitting-in-two trick. Eventually there is one of you for every terminating corridor in the entire building.

Each corridor contains a thousand rooms. Your superpower is getting stretched a little thin, so you only make 50 of yourself to search the rooms in parallel.

Once a clone enters a room, it must search through all the cracks and hidden pockets of the room. It switches to a second superpower. It divides into a million nanobots, each of which flies or crawls to some unseen spot in the room. You don't understand this power—it just works, once you start it. Under their own control, the nanobots go, search the room and come back and reassemble into you, and suddenly, somehow, you just know whether the item is in the room or not.

I'd love to be able to say, "Your superpower in the science-fiction movie? That's what concurrency is." That it's as simple as splitting yourself in two every time you have more tasks to solve. The problem is that any model we use to describe this phenomenon ends up being a leaky abstraction.

Here's one of those leaks: In an ideal world, every time you cloned yourself, you would also duplicate a hardware processor to run that clone. But of course that isn't what happens—you actually might have four or eight processors on your machine (typical when this was written). You might also have more, and there are still lots of situations where you have only one processor. In the abstraction under discussion, the way physical processors are allocated not only leaks through but can even dominate your decisions.

Let's change something in our science-fiction movie. Now when each clone searcher eventually reaches a door they must knock on it and wait until someone answers.

If we have one processor per searcher, this is no problem—the processor just idles until the door is answered. But if we only have eight processors and thousands of searchers, we don't want a processor to be idle just because a searcher happens to be blocked, waiting for a door to be answered. Instead, we want that processor applied to a searcher where it can do some real work, so we need mechanisms to switch processors from one task to another.

Many models are able to effectively hide the number of processors and allow you to pretend you have a very large number. But there are situations where this breaks down, when you must know the number of processors so you can work around that number.

One of the biggest impacts depends on whether you have a single processor or more than one. If you only have one processor, then the cost of task-switching is also borne by that processor, and applying concurrency techniques to your system can make it run *slower*.

This might make you decide that, in the case of a single processor, it never makes sense to write concurrent code. However, there are situations where the *model* of concurrency produces much simpler code and it's actually worth having it run slower to achieve that.

In the case of the clones knocking on doors and waiting, even the single-processor system benefits from concurrency because it can switch from a task that is waiting (*blocked*) to one that is ready to go. But if all the tasks can run all the time, then the cost of switching will slow everything down, and in that case concurrency usually only makes sense if you *do* have multiple processors.

Suppose you are trying to crack some kind of encryption. The more workers trying to crack it at the same time, the better chance you have of finding the answer sooner. Here, each worker can constantly use as much processor time as you can give it, and the best situation is when each worker has their own processor—in this case (a compute-bound problem), you should write the code so you *only* have as many workers as you have processors.

In a customer-service department that takes phone calls, you only have a certain number of people, but you can have lots of phone calls. Those people (the processors) must work on one phone call at a time until it is complete, and extra calls must be queued.

In the fairy tale of "The Shoemaker and the Elves," the shoemaker had too much work to do and when he was asleep, a group of elves came and made shoes for him. Here the work is distributed, but even with a large number of physical processors the limitation comes when building certain parts of the shoe—if, for example, the sole takes the longest to make, that limits the rate of shoe creation and changes the way you design your solution.

Thus, the problem you're trying to solve drives the design of the solution. There's the lovely abstraction of breaking a problem into subtasks that "run independently," then there's the reality of how it's actually going to happen. The physical reality keeps intruding upon, and shaking up, that abstraction.

That's only part of the problem. Consider a factory that makes cakes. We've somehow distributed the cake-making task among workers, but now it's time for a worker to put their cake in a box. There's a box sitting there, ready to receive a cake. But before the worker can put the cake into the box, another worker darts in and puts *their* cake in the box instead! Our worker is already putting the cake in, and bam! The two cakes are smashed together and ruined. This is the common "shared memory" problem that produces what we call a *race condition*, where the result depends on which worker can get their cake in the box first (you typically solve the problem using a locking mechanism so one worker can grab the box first and prevent cake-smashing).

The problem occurs when tasks that execute "at the same time" interfere with each other. This can happen in such a subtle and occasional manner it's probably fair to say that concurrency is "arguably deterministic but effectively nondeterministic." That is, you can hypothetically write concurrent programs that, through care and code inspection, work correctly. In practice, however, it's much more common to write concurrent programs that only appear to work, but given the right conditions, will fail. These conditions might never actually occur, or occur so infrequently you never see them during testing. In fact, it's often impossible to write test code to generate failure conditions for your concurrent program. The resulting failures often only occur occasionally, and as a result they appear in the form of customer complaints. This is one of the strongest arguments for studying concurrency: If you ignore it, you're likely to get bitten.

Concurrency thus seems fraught with peril, and if that makes you a bit fearful, this is probably a good thing. Although Java 8 makes large improvements in concurrency, there are still no safety nets like compile-time verification or checked exceptions to

tell you when you make a mistake. With concurrency, you're on your own, and only by being knowledgeable, suspicious and aggressive can you write reliable concurrent code in Java.

Concurrency is for Speed

After hearing about the pitfalls of concurrent programming, you may rightly be wondering if it's worth the trouble. The answer is "no, unless your program isn't running fast enough." And you'll want to think carefully before deciding it isn't. Do not casually jump into the well of grief that is concurrent programming. If there's a way to run your program on a faster machine or if you can profile it and discover the bottleneck and swap in a faster algorithm in that spot, do that instead. Only if there's clearly no other choice should you begin using concurrency, and then only in isolated places.

The speed issue sounds simple at first: If you want a program to run faster, break it into pieces and run each piece on a separate processor. With our ability to increase clock speeds running out of steam (at least for conventional chips), speed improvements are appearing in the form of multicore processors rather than faster chips. To make your programs run faster, you'll have to learn to take advantage of those extra processors, and that's one thing that concurrency gives you.

With a multiprocessor machine, multiple tasks can be distributed across those processors, which can dramatically improve throughput. This is often the case with powerful multiprocessor Web servers, which can distribute large numbers of user requests across CPUs in a program that allocates one thread per request.

However, concurrency can often improve the performance of programs running on a *single* processor. This can sound a bit counterintuitive. If you think about it, a concurrent program running on a single processor should actually have *more* overhead than if all the parts of the program ran sequentially, because of the added cost of the *context switch* (changing from one task to another). On the surface, it would appear cheaper to run all the parts of the program as a single task and save the cost of context switching.

The issue that can make a difference is *blocking*. If one task in your program is unable to continue because of some condition outside of the control of the program (typically

I/O), we say that the task or the thread *blocks* (in our science-fiction story, the clone has knocked on the door and is waiting for it to open). Without concurrency, the whole program comes to a stop until the external condition changes. If the program is written using concurrency, however, the other tasks in the program can continue to execute when one task is blocked, so the program continues to move forward. In fact, from a performance standpoint, it makes no sense to use concurrency on a single-processor machine unless one of the tasks might block.

A common example of performance improvements in single-processor systems is *event-driven programming*, in particular user-interface programming. Consider a program that performs some long-running operation and thus ends up ignoring user input and being unresponsive. If you have a "quit" button, you don't want to poll it in every piece of code you write. This produces awkward code, without any guarantee that a programmer won't forget to perform the check. Without concurrency, the only way to produce a responsive user interface is for all tasks to periodically check for user input. By creating a separate thread of execution to respond to user input, the program guarantees a certain level of responsiveness.

A straightforward way to implement concurrency is at the operating system level, using *processes*, which are different from threads. A process is a self-contained program running within its own address space. Processes are attractive because the operating system usually isolates one process from another so they cannot interfere with each other. This makes programming with processes relatively easy. In contrast, threads share resources like memory and I/O, so a fundamental difficulty in writing multithreaded programs is coordinating these resources between different thread-driven tasks, so they cannot be accessed by more than one task at a time.

Some people go so far as to advocate processes as the only reasonable approach to concurrency,[168] but unfortunately there are generally quantity and overhead limitations to processes that prevent their applicability across the concurrency spectrum. (Eventually you get used to the standard concurrency refrain, "That approach works in some cases but not in other cases").

Some programming languages are designed to isolate concurrent tasks from each other. These are generally called *functional languages*, where each function call produces no side effects (and so cannot interfere with other functions) and can thus be driven as an independent task. *Erlang* is one such language, and it includes safe

[168] Eric Raymond, for example, makes a strong case in *The Art of UNIX Programming* (Addison-Wesley, 2004).

mechanisms for one task to communicate with another. If you find that a portion of your program must make heavy use of concurrency and you are running into excessive problems trying to build that portion, you might consider creating that part of your program in a dedicated concurrency language.

Java took the more traditional approach of adding support for threading on top of a sequential language.[169] Instead of forking external processes in a multitasking operating system, threading creates tasks *within* the single process represented by the executing program.

Concurrency imposes costs, including complexity costs, but can be outweighed by improvements in program design, resource balancing, and user convenience. In general, concurrency enables you to create a more loosely coupled design; otherwise, parts of your code would be forced to pay explicit attention to operations that would normally be handled by concurrency.

The Four Maxims of Java Concurrency

After grappling with Java concurrency over many years, I developed these four maxims:

1. Don't do it.
2. Nothing is true and everything matters.
3. Just because it works doesn't mean it's not broken.
4. You must still understand it.

These are specifically about problems in the design of Java, although they can be applied to some other languages as well. However, there do exist languages that are designed to prevent these issues.

1. Don't do it

(And don't do it yourself).

[169] It could be argued that trying to bolt concurrency onto a sequential language is a doomed approach, but you'll have to draw your own conclusions.

The easiest way to avoid entangling yourself in the profound problems produced by concurrency is not to do it. Although it can be seductive and seem safe enough to try something simple, the pitfalls are myriad and subtle. If you can avoid it, your life will be much easier.

The *only* thing that justifies concurrency is speed. If your program isn't running fast enough—and be careful here, because just *wanting* it to run faster isn't justification—first apply a profiler (see Profiling and Optimizing) to discover whether there's some other optimization you can perform.

If you're compelled into concurrency, take the simplest, safest approach to the problem. Use well-known libraries and write as little of your own code as possible. With concurrency, there's no such thing as "too simple." Cleverness is your enemy.

2. Nothing is true and everything matters

Programming without concurrency, you've come to expect a certain order and consistency in your world. With something as simple as setting a variable to a value, it's obvious it should always work properly.

In concurrency-land, some things might be true and others are not, to the point where you must assume that nothing is true. You must question everything. Even setting a variable to a value might or might not work the way you expect, and it goes downhill from there. I've become familiar with the feeling of discovering that something I thought should obviously work, actually doesn't.

All kinds of things you can ignore in non-concurrent programming suddenly become important with concurrency. For example, you must now know about the processor cache and the problems of keeping the local cache consistent with main memory. You must understand the deep complexities of object construction so that your constructor doesn't accidentally expose data to change by other threads. The list goes on.

Although these topics are too complex to give you expertise in this chapter (again, see *Java Concurrency in Practice*), you must be aware of them.

3. Just because it works doesn't mean it's not broken

You can easily write a concurrent program that appears to work but is actually broken, and the problem only reveals itself under the rarest of conditions—inevitably as a user problem after you've deployed the program.

- You can't prove a concurrent program is correct, you can only (sometimes) prove it is incorrect.
- Most of the time you can't even do that: If it's broken you probably won't be able to detect it.
- You can't usually write useful tests, so to discover bugs you must rely on code inspection combined with deep knowledge of concurrency.
- Even working programs only work under their design parameters. Most concurrent programs fail in some way when those design parameters are exceeded.

In other Java topics, we develop a sense of determinism. Everything happens as promised (or implied) by the language, which is comforting and expected—after all, the point of a programming language is to get the machine to do what we want. Moving from the world of deterministic programming into the realm of concurrent programming, we encounter a cognitive bias called the Dunning-Kruger Effect[170] which can be summed up as "the less you know, the more you think you know." It means "…relatively unskilled persons suffer illusory superiority, mistakenly assessing their ability to be much higher than it really is."

My own experience is that, no matter how certain you are that your code is thread-safe, it's probably broken. It's all too easy to be very sure you understand all the issues, then months or years later you discover some concept that makes you realize that most everything you've written is actually vulnerable to concurrency bugs. The compiler doesn't tell you when something is incorrect. To get it right you must hold all the issues of concurrency in your forebrain as you study your code.

In all the non-concurrent areas of Java, "no obvious bugs and no compiler complaints" seems to mean that everything is OK. With concurrency, it means nothing. The very worst thing you can be in this situation is "confident."

[170] https://en.wikipedia.org/wiki/Dunning%E2%80%93Kruger_effect

4. You must still understand it.

After maxims 1-3 you might be properly frightened of concurrency, and think, "I've avoided it up until now, perhaps I can just continue avoiding it."

This is a rational reaction. You might know about other programming languages that are better designed to build concurrent programs—even ones that run on the JVM (and thus provide easy communication with Java) such as Clojure or Kotlin. Why not write the concurrent parts in those languages and use Java for everything else?

Alas, you cannot escape so easily:

- Even if you never explicitly create a thread, frameworks you use might—for example, the Swing Graphical User Interface (GUI) library, or something as simple as the `Timer` class.
- Here's the worst thing: when you create components, you must assume those components might be reused in a multithreading environment. Even if your solution is to give up and declare that your components are "not thread-safe," you must still know enough to realize that such a statement is important and what it means.

People sometimes suggest that concurrency is too advanced to include in a book that introduces the language. They argue that concurrency is a discrete topic that can be treated independently, and the few cases where it appears in daily programming (such as graphical user interfaces) can be handled with special idioms. Why introduce such a complex topic if you can avoid it?

Alas, if only it were so. Unfortunately, you don't get to choose when threads appear in your Java programs. Just because you never start a thread yourself doesn't mean you can avoid writing threaded code. For example, Web systems are one of the most common Java applications, and are inherently multithreaded—Web servers typically contain multiple processors, and parallelism is an ideal way to utilize these processors. As simple as such a system might seem, you must understand concurrency to write it properly.

Java is a multithreaded language, and concurrency issues are present whether you are aware of them or not. As a result, there are many Java programs in use that either just work by accident, or work most of the time and mysteriously break every now and again because of undiscovered flaws. Sometimes this breakage is relatively benign,

but sometimes it means the loss of valuable data, and if you aren't at least aware of concurrency issues, you can end up assuming the problem is somewhere else rather than in your code. These kinds of issues can also be exposed or amplified if a program is moved to a multiprocessor system. Basically, knowing about concurrency makes you aware that apparently correct programs can exhibit incorrect behavior.

The Brutal Truth

When humans began cooking their food, they dramatically reduced the amount of energy their bodies required to break down and digest that food. Cooking created an "externalized stomach," thus freeing up that energy for other pursuits. The technology of fire enabled civilization.

We have now begun a second fundamental shift by creating an "externalized brain" through the technology of computers and networks. We've only scratched the surface, but have already triggered other shifts such as the ability to design biological mechanisms, and have seen a dramatic acceleration in cultural evolution (in the past, people had to travel to mix cultures, but now they are beginning to mix on the Internet). The impact and benefits of these shifts have far exceeded the abilities of science-fiction writers to predict them (they have an especially hard time predicting cultural and personal changes, or even secondary effects from technology shifts).

With such a fundamental human change, it is unsurprising to see numerous disruptions and failed experiments. Indeed, evolution relies on myriad experiments, most of which fail. Those experiments are essential to move forward.

Java was created in an atmosphere of confidence, enthusiasm, and urgency. When inventing a programming language, it's all too easy to feel like the initial plasticity of the language will persist, that you can try something out and if it doesn't work out, fix it. Programming languages are unique this way—they go through water-like phase changes: gaseous, liquid and finally solid. During the gaseous phase the flexibility seems infinite, and it's easy to think it will always be that way. Once people start using your language, changes have bigger impacts and the environment becomes more viscous. The *process* of language design is itself an art.

The urgency came from the initial rise of the internet. It seemed like a race, and the first one to get through the starting gate would "win" (indeed, the popularity of

languages like Java, JavaScript and PHP seem to bear this out). Alas, the cognitive load and technical debt produced by designing languages in a hurry eventually catches up with us.

Turing-completeness[171] is not enough; languages need something more: they must enable creative expression, not weigh us down with needless detail. It is pointless to liberate our mental capacity only to turn around and bog it down again. I acknowledge that we have accomplished amazing things despite these issues, but also I see how much more we can do without them.

Enthusiasm caused the original Java designers to throw in features because they seemed necessary. Confidence (and the gaseousness of the original language) let them think that any problems could be fixed later. Somewhere along the timeline, someone decided that anything added to Java is fixed and permanent—this is confidence squared, to believe that the first decision would always be the right one, and so we see the landscape of Java littered with poor decisions. Some of these decisions ultimately do have little consequence; you can tell people not to use `Vector`, for example, but leave it in the language to support old code.

Threads were included in Java 1.0. Certainly, concurrency is a fundamental language design decision that affects far corners of the language, and it's hard to imagine adding it later. To be fair, at the time it wasn't clear just how fundamental concurrency is. Other languages like C were able to treat threads as an add-on feature, so the Java designers followed suit, including a `Thread` class and the necessary JVM support (which is more complex than you might imagine).

The C language is primitive, and this limits your ambitions. These limits make add-on threading libraries reasonable. The much grander ambitions of Java rapidly exposed fundamental problems when taking a primitive model and pasting it into a sophisticated language. This mismatch is made obvious in the deprecation of many of the methods in the `Thread` class, and in subsequent waves of higher-level libraries that attempt to provide better abstractions for concurrency.

Unfortunately, to get concurrency right in a higher-level language, all language features are affected, including the most basic ones like whether an identifier represents a changeable value. Making everything *invariant* and preventing *side-effects* in functions and methods produces a sea-change in the simplification of concurrent programming (these are foundations of pure functional programming

[171] https://en.wikipedia.org/wiki/Turing_completeness

languages), but at the time seemed like strange ideas for creators of a mainstream language. The original Java designers were either unaware of these choices or decided they were too different and would turn away many potential adopters of the language. We can be generous and say that the language-design community simply didn't have enough experience at the time to understand the impact of patching in a threading library.

The Java experiment has shown us that the results are quietly disastrous. Programmers easily fall into the trap of thinking that Java threads aren't that difficult. Programs that seem to work are riddled with subtle concurrency bugs.

To get concurrency right, language features must be designed from the ground up with concurrency in mind. That ship has sailed; Java will never be a language designed for concurrency, but simply a language that allows it.

What's impressive is how far it has come despite these fundamental un-fixable flaws. Subsequent versions of Java have added libraries to raise the level of abstraction when working with concurrency. In fact, I never would have thought it possible to make the improvements in Java 8: parallel streams and `CompletableFutures`—this was a magic trick of epic proportions, the like of which I will be very surprised to see repeated.

These improvements are very helpful, and we will focus on parallel streams and `CompletableFutures` in this chapter. Although they can greatly simplify the way you think about concurrency and the subsequent code, the fundamental issues still exist: all parts of your code are vulnerable because of the original design of the Java language, and you must understand these complicated and subtle issues. Threading in Java can never be simple or safe; that experience must be relegated to another, newer language.

The Rest of the Chapter

Here's what we'll cover in the remainder of this chapter. Remember that the emphasis of this chapter is on using the most recent and high-level Java concurrency constructs. Using these makes your life much easier than the older alternatives. However, there are still some low-level tools that you will encounter in legacy code. On

occasion, you might be forced to use some of these yourself. The Appendix: Low-Level Concurrency contains an introduction to some of the more primitive Java concurrency elements.

- **Parallel Streams**

Up to this point in the book, I've emphasized the improved syntax provided by Java 8 `Streams`. Now that you're comfortable with (and I hope, a fan of) that syntax, you can reap additional benefits: You can parallelize a stream by simply adding `parallel()` into the expression. This is a simple, powerful, and frankly rather amazing way to take advantage of multiple processors.

Adding `parallel()` to increase speed seems trivial, but alas, it can never be that simple, as you just learned in The Brutal Truth. I'll demonstrate and explain some of the pitfalls that come from blindly adding `parallel()` to a `Stream` expression.

- **Creating and Running Tasks**

A task is a piece of code that can be run independently. To explain some of the basics of creating and running tasks, this section introduces a less-sophisticated mechanism than parallel streams or `CompletableFutures`: the `Executor`. `Executors` manage a pool of low-level `Thread` objects (the most primitive form of concurrency in Java). You create a task, then hand it to an `Executor` to be run.

There are multiple types of `Executors` for different purposes. Here, we will show canonical forms representing the simplest and best approaches to creating and running tasks.

- **Terminating Long-Running Tasks**

Tasks run independently and thus need a mechanism to shut them down. The typical approach uses a flag, and this introduces the problem of shared memory, which we'll sidestep using Java's "Atomic" library.

- **Completable Futures**

When you take clothing to a dry cleaner, they give you a receipt. You continue with other tasks, and eventually your clothing is clean and you can pick it up. The receipt is your connection to the task performed in the background by the dry cleaner. This is the approach taken by the `Future` introduced in Java 5.

The `Future` was somewhat more convenient than the previous approach, but you must still show up and fetch your dry-cleaning with the ticket, and wait if the task hasn't completed. For a pipeline of operations, `Futures` don't really help that much.

The Java 8 `CompletableFuture` is a much better solution: it allows you to chain operations together so you don't have to write the code to interface sequenced operations. With `CompletableFutures` it becomes much easier to do something like "procure ingredients, combine ingredients, cook food, serve food, clean up dishes, store dishes" as a sequence of chained operations.

- **Deadlock**

Some tasks must wait—*block*—for results from other tasks. A blocked task has the potential of waiting for another blocked task, which is waiting for another one, etc. If the chain of blocked tasks loops around to the first one, no one can make any progress and you get *deadlock*.

The biggest problems happen if the deadlock doesn't show up right away when you run the program. Your system can be *deadlock-prone*, and will only deadlock under certain conditions. A program might run just fine on a certain platform, for example your development machine, but then start deadlocking when you deploy it to different hardware.

Deadlock typically arises from subtle programming errors; a sequence of innocent decisions that ends up accidentally creating a dependency loop. This section contains a classic example that demonstrates the elusive nature of deadlocking.

- **Effort**, **Complexity**, **Cost**

We'll finish the chapter by simulating the process of creating a pizza, first implementing it using parallel streams, then `CompletableFutures`. This is not just a comparison of the two approaches, but more importantly an exploration of how much work you should invest in trying to speed up a program.

Parallel Streams

One significant benefit of Java 8 streams is that, in some cases, they can be easily parallelized. This comes from careful library design, in particular the way streams

use *internal iteration*—that is, they control their own iterators. In particular, they use a special kind of iterator called a *Spliterator* which is constrained to be easily and automatically dividable. This produces the rather magical result of being able to simply say `.parallel()` and suddenly everything in your stream is running as a set of parallel tasks. If your code is written using `Streams`, parallelizing to increase speed seems trivial.

For example, consider `Prime.java` from Streams. Finding prime numbers can be a time-consuming process, as we can see by timing a rewrite of that program:

```
// concurrent/ParallelPrime.java
import java.util.*;
import java.util.stream.*;
import static java.util.stream.LongStream.*;
import java.io.*;
import java.nio.file.*;
import onjava.Timer;

public class ParallelPrime {
  static final int COUNT = 100_000;
  public static boolean isPrime(long n) {
    return rangeClosed(2, (long)Math.sqrt(n))
      .noneMatch(i -> n % i == 0);
  }
  public static void main(String[] args)
    throws IOException {
    Timer timer = new Timer();
    List<String> primes =
      iterate(2, i -> i + 1)
        .parallel()                        // [1]
        .filter(ParallelPrime::isPrime)
        .limit(COUNT)
        .mapToObj(Long::toString)
        .collect(Collectors.toList());
    System.out.println(timer.duration());
    Files.write(Paths.get("primes.txt"), primes,
      StandardOpenOption.CREATE);
  }
}
/* Output:
1635
*/
```

Notice this is not a microbenchmark because we are timing the whole program. We save the data on disk to guard against aggressive optimization; if we didn't do anything with the result a wily compiler might observe that the program is pointless and eliminate the calculation (this is unlikely but not impossible). Note the simplicity of writing a file using the `nio2` library (described in the Files chapter).

When I comment out the **[1]** `parallel()` line, my results take roughly three times as long as with `parallel()` in place.

Parallel streams seem like a sweet deal. All you need do is cast your programming problems into streams, then insert `parallel()` to speed things up. Indeed, sometimes it's just that easy. But unfortunately there are numerous pitfalls.

`parallel()` is not a Panacea

As an exploration of the uncertainties of streams and parallel streams, let's look at a problem that seems simple: summing an incremental sequence of numbers. There turns out to be a surprising number of ways to do this, and I'll take the risk of comparing them through timing—trying to be careful, but acknowledging I might fall into one of the many fundamental pitfalls when timing code execution. The results may have some flaws (there's no "warming up" of the JVM, for example), but I think it nonetheless gives some useful indications.

I'll start with a timing method `timeTest()` which takes a `LongSupplier`, measures the length of the `getAsLong()` call, compares the result with a `checkValue` and displays the results.

Note that everything must rigorously use `longs`; I spent a bit of time chasing down quiet overflows before realizing I had missed `longs` in important places.

All the numbers and discussions about time and memory refer to "my machine." Your experience will probably be different.

```
// concurrent/Summing.java
import java.util.stream.*;
import java.util.function.*;
import onjava.Timer;

public class Summing {
  static void timeTest(String id, long checkValue,
    LongSupplier operation) {
    System.out.print(id + ": ");
    Timer timer = new Timer();
    long result = operation.getAsLong();
    if(result == checkValue)
      System.out.println(timer.duration() + "ms");
    else
      System.out.format("result: %d%ncheckValue: %d%n",
        result, checkValue);
  }
  public static final int SZ = 100_000_000;
  // This even works:
  // public static final int SZ = 1_000_000_000;
  public static final long CHECK =
    (long)SZ * ((long)SZ + 1)/2; // Gauss's formula
  public static void main(String[] args) {
    System.out.println(CHECK);
    timeTest("Sum Stream", CHECK, () ->
      LongStream.rangeClosed(0, SZ).sum());
    timeTest("Sum Stream Parallel", CHECK, () ->
      LongStream.rangeClosed(0, SZ).parallel().sum());
    timeTest("Sum Iterated", CHECK, () ->
      LongStream.iterate(0, i -> i + 1)
        .limit(SZ+1).sum());
    // Slower & runs out of memory above 1_000_000:
    // timeTest("Sum Iterated Parallel", CHECK, () ->
    //   LongStream.iterate(0, i -> i + 1)
    //     .parallel()
    //     .limit(SZ+1).sum());
  }
}
/* Output:
5000000050000000
Sum Stream: 841ms
Sum Stream Parallel: 179ms
```

```
Sum Iterated: 4051ms
*/
```

The `CHECK` value is calculated using the formula created by Carl Friedrich Gauss while still in primary school in the late 1700's.

This first version of `main()` uses the straightforward approach of generating a `Stream` and calling `sum()`. We see the benefits of streams in that a `SZ` of a billion is handled without overflow (I use a smaller number so the program doesn't take so long to run). Using the basic range operation with `parallel()` is notably faster.

If `iterate()` is used to produce the sequence the slowdown is dramatic, probably because the lambda must be called each time a number is generated. But if we try to parallelize that, the result not only takes longer than the non-parallel version but it also runs out of memory (on some machines) when `SZ` gets above a million. Of course you wouldn't use `iterate()` when you could use `range()`, but if you're generating something other than a simple sequence you must use `iterate()`. Applying `parallel()` is a reasonable thing to attempt, but produces these surprising results. We shall explore the reason for the memory limitation in a later section, but we can make some initial observations regarding the stream parallel algorithms:

- Stream parallelism divides the incoming data into pieces so the algorithm(s) can be applied to those separate pieces.
- Arrays split cheaply, evenly and with perfect knowledge of split sizes.
- Linked Lists have none of these properties; "splitting" a linked list only means dividing it into "first element" and "rest of list," which is relatively useless.
- Stateless generators behave like arrays; the use of `range` above is stateless.
- Iterative generators behave like linked lists; `iterate()` is an iterative generator.

Now let's try solving the problem by filling an array with values, then summing over the array. Because the array is only allocated once, it seems unlikely we'll run into garbage collection timing issues.

First we'll try an array filled with primitive `longs`:

```
// concurrent/Summing2.java
// {ExcludeFromTravisCI}
import java.util.*;

public class Summing2 {
  static long basicSum(long[] ia) {
    long sum = 0;
    int size = ia.length;
    for(int i = 0; i < size; i++)
      sum += ia[i];
    return sum;
  }
  // Approximate largest value of SZ before
  // running out of memory on my machine:
  public static final int SZ = 20_000_000;
  public static final long CHECK =
    (long)SZ * ((long)SZ + 1)/2;
  public static void main(String[] args) {
    System.out.println(CHECK);
    long[] la = new long[SZ+1];
    Arrays.parallelSetAll(la, i -> i);
    Summing.timeTest("Array Stream Sum", CHECK, () ->
      Arrays.stream(la).sum());
    Summing.timeTest("Parallel", CHECK, () ->
      Arrays.stream(la).parallel().sum());
    Summing.timeTest("Basic Sum", CHECK, () ->
      basicSum(la));
    // Destructive summation:
    Summing.timeTest("parallelPrefix", CHECK, () -> {
      Arrays.parallelPrefix(la, Long::sum);
      return la[la.length - 1];
    });
  }
}
/* Output:
200000010000000
Array Stream Sum: 166ms
Parallel: 30ms
Basic Sum: 45ms
parallelPrefix: 53ms
*/
```

The first limitation is memory size; because the array is allocated up front, we can't create anything nearly as large as the previous version. Parallelizing speeds things up, even a bit faster than just looping through using `basicSum()`. Interestingly, `Arrays.parallelPrefix()` seems to actually slow things down. However, any of these techniques might be more useful under other conditions—that's why you can't make any deterministic statements about what to do, other than "you must try it out."

Finally, consider the effect of using boxed `Long`s instead:

```
// concurrent/Summing3.java
// {ExcludeFromTravisCI}
import java.util.*;

public class Summing3 {
  static long basicSum(Long[] ia) {
    long sum = 0;
    int size = ia.length;
    for(int i = 0; i < size; i++)
      sum += ia[i];
    return sum;
  }
  // Approximate largest value of SZ before
  // running out of memory on my machine:
  public static final int SZ = 10_000_000;
  public static final long CHECK =
    (long)SZ * ((long)SZ + 1)/2;
  public static void main(String[] args) {
    System.out.println(CHECK);
    Long[] aL = new Long[SZ+1];
    Arrays.parallelSetAll(aL, i -> (long)i);
    Summing.timeTest("Long Array Stream Reduce",
      CHECK, () ->
      Arrays.stream(aL).reduce(0L, Long::sum));
    Summing.timeTest("Long Basic Sum", CHECK, () ->
      basicSum(aL));
    // Destructive summation:
    Summing.timeTest("Long parallelPrefix",CHECK, ()-> {
      Arrays.parallelPrefix(aL, Long::sum);
      return aL[aL.length - 1];
    });
  }
```

```
}
/* Output:
50000005000000
Long Array Stream Reduce: 1510ms
Long Basic Sum: 35ms
Long parallelPrefix: 4306ms
*/
```

Now the amount of memory available is approximately cut in half, and the amount of time required has exploded in all cases except `basicSum()`, which simply loops through the array. Surprisingly, `Arrays.parallelPrefix()` takes significantly longer than any other approach.

I separated the `parallel()` version because running it inside the above program caused a lengthy garbage collection, distorting the results:

```
// concurrent/Summing4.java
// {ExcludeFromTravisCI}
import java.util.*;

public class Summing4 {
  public static void main(String[] args) {
    System.out.println(Summing3.CHECK);
    Long[] aL = new Long[Summing3.SZ+1];
    Arrays.parallelSetAll(aL, i -> (long)i);
    Summing.timeTest("Long Parallel",
      Summing3.CHECK, () ->
      Arrays.stream(aL)
        .parallel()
        .reduce(0L,Long::sum));
  }
}
/* Output:
50000005000000
Long Parallel: 1147ms
*/
```

It's slightly faster than the non-`parallel()` version, but not significantly.

A big reason for this increase in time is the processor memory cache. With the primitive `longs` in `Summing2.java`, the array `la` is contiguous memory. The processor can more easily anticipate the use of this array and keep the cache filled with the array

elements that are needed next. Accessing the cache is much, much faster than going out to main memory. It appears that the `Long parallelPrefix` calculation suffers because it reads two array elements for each calculation, plus writes the result back into the array, and each of these produces an out-of-cache reference for the `Long`.

With `Summing3.java` and `Summing4.java`, `aL` is an array of `Long`, which is not a contiguous array of data, but a contiguous array of references to `Long` objects. Even though that array will probably be kept in cache, the objects pointed to will almost always be out-of-cache.

These examples used different `SZ` values to show the memory limits. To do a time comparison, here are the results with `SZ` set to the smallest value of 10 million:

```
Sum Stream: 69ms
Sum Stream Parallel: 18ms
Sum Iterated: 277ms
Array Stream Sum: 57ms
Parallel: 14ms
Basic Sum: 16ms
parallelPrefix: 28ms
Long Array Stream Reduce: 1046ms
Long Basic Sum: 21ms
Long parallelPrefix: 3287ms
Long Parallel: 1008ms
```

While Java 8's various built-in "parallel" tools are terrific, I've seen them treated as magical panaceas: "Just add `parallel()` and it will run faster!" I hope I've begun to show that this is not the case at all, and that blindly applying the built-in "parallel" operations can sometimes even make things run noticeably slower.

The `parallel()`/`limit()` Intersection

There's a further complication when using `parallel()`. Streams, as taken from other languages, are designed around an *infinite* stream model. If you have a finite number of elements you use a collection and the associated algorithms designed for limited-sized collections. If you use infinite streams, you use those algorithms, optimized for streams.

Java 8 conflates the two. For example, `Collections` have no built-in `map()` operation. The only stream-like batch operation in `Collection` and `Map` is `forEach()`. If

you want to perform operations like `map()` and `reduce()`, you must first turn the `Collection` into a `Stream` where those operations exist:

```
// concurrent/CollectionIntoStream.java
import onjava.*;
import java.util.*;
import java.util.stream.*;

public class CollectionIntoStream {
  public static void main(String[] args) {
    List<String> strings =
      Stream.generate(new Rand.String(5))
        .limit(10)
        .collect(Collectors.toList());
    strings.forEach(System.out::println);
    // Convert to a Stream for many more options:
    String result = strings.stream()
      .map(String::toUpperCase)
      .map(s -> s.substring(2))
      .reduce(":", (s1, s2) -> s1 + s2);
    System.out.println(result);
  }
}
/* Output:
btpen
pccux
szgvg
meinn
eeloz
tdvew
cippc
ygpoa
lkljl
bynxt
:PENCUXGVGINNLOZVEWPPCPOALJLNXT
*/
```

`Collections` do have some batch operations like `removeAll()`, `removeIf()` and `retainAll()`, but these are destructive actions. `ConcurrentHashMap` has special extensive support for `forEach` and `reduce` operations.

In many cases, there's no problem with just calling `stream()` or `parallelStream()`

on a collection. Sometimes, however, conflating `Stream` with `Collection` can produce surprises. Here's an interesting puzzle:

```
// concurrent/ParallelStreamPuzzle.java
import java.util.*;
import java.util.function.*;
import java.util.stream.*;

public class ParallelStreamPuzzle {
  static class IntGenerator
  implements Supplier<Integer> {
    private int current = 0;
    @Override public Integer get() {
      return current++;
    }
  }
  public static void main(String[] args) {
    List<Integer> x =
      Stream.generate(new IntGenerator())
        .limit(10)
        .parallel()                    // [1]
        .collect(Collectors.toList());
    System.out.println(x);
  }
}
/* Output:
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
*/
```

If you run this with the **[1]** commented, it produces the expected:

```
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
```

Every time. But with the `.parallel()` included, it seems like a random-number generator, with outputs (that differ from one run to the next) like:

```
[0, 3, 6, 8, 11, 14, 17, 20, 23, 26]
```

How can such a simple program seem so broken? Let's consider what we're trying to achieve here: "parallel generation."" What does that even mean? A bunch of threads all pulling at a single generator, then somehow selecting a limited set of the results? The code makes it look simple, but it turns out to be an especially messy problem.

To see it, we'll add some instrumentation. We're dealing with threads so we must capture any trace information into a concurrent data structure. Here I use a `ConcurrentLinkedDeque`:

```
// concurrent/ParallelStreamPuzzle2.java
import java.util.*;
import java.util.function.*;
import java.util.stream.*;
import java.util.concurrent.*;
import java.util.concurrent.atomic.*;
import java.nio.file.*;

public class ParallelStreamPuzzle2 {
  public static final Deque<String> trace =
    new ConcurrentLinkedDeque<>();
  static class
  IntGenerator implements Supplier<Integer> {
    private AtomicInteger current =
      new AtomicInteger();
    @Override public Integer get() {
      trace.add(current.get() + ": " +
        Thread.currentThread().getName());
      return current.getAndIncrement();
    }
  }
  public static void
  main(String[] args) throws Exception {
    List<Integer> x =
      Stream.generate(new IntGenerator())
        .limit(10)
        .parallel()
        .collect(Collectors.toList());
    System.out.println(x);
    Files.write(Paths.get("PSP2.txt"), trace);
  }
}
/* Output:
[1, 5, 7, 8, 9, 11, 13, 15, 18, 21]
*/
```

`current` is defined using the thread-safe `AtomicInteger` class, to prevent race conditions; `parallel()` allows `get()` to be called by multiple threads.

You might be surprised when looking at `PSP2.txt`. `IntGenerator.get()` is called **1024** times!

```
0: main
1: ForkJoinPool.commonPool-worker-1
2: ForkJoinPool.commonPool-worker-2
3: ForkJoinPool.commonPool-worker-2
4: ForkJoinPool.commonPool-worker-1
5: ForkJoinPool.commonPool-worker-1
6: ForkJoinPool.commonPool-worker-1
7: ForkJoinPool.commonPool-worker-1
8: ForkJoinPool.commonPool-worker-4
9: ForkJoinPool.commonPool-worker-4
10: ForkJoinPool.commonPool-worker-4
11: main
12: main
13: main
14: main
15: main
...
1017: ForkJoinPool.commonPool-worker-1
1018: ForkJoinPool.commonPool-worker-6
1019: ForkJoinPool.commonPool-worker-6
1020: ForkJoinPool.commonPool-worker-1
1021: ForkJoinPool.commonPool-worker-1
1022: ForkJoinPool.commonPool-worker-1
1023: ForkJoinPool.commonPool-worker-1
```

This block size appears to be part of the internal implementation (try different arguments to `limit()` to see different block sizes). Combining `parallel()` with `limit()` tells it to pre-fetch a bunch of values, to be fed out as the stream.

Try to imagine what's happening here: a stream abstracts an infinite sequence, produced on-demand. When you ask it to produce the stream in parallel, you're asking all those threads to call `get()` whenever they can. Add in `limit()` and you're saying "just take *some* of these." Basically, you're *asking* for random output when you combine `parallel()` with `limit()`—which might be just fine for the problem you're solving. But you must understand that when you do this. It's an experts-only feature, and not something to be thrown out to argue that "Java gets it wrong."

What's a more reasonable way to approach the problem? Well, if you want to produce a stream of `int`, you can use `IntStream.range()`, like this:

```
// concurrent/ParallelStreamPuzzle3.java
// {VisuallyInspectOutput}
import java.util.*;
import java.util.stream.*;

public class ParallelStreamPuzzle3 {
  public static void main(String[] args) {
    List<Integer> x = IntStream.range(0, 30)
      .peek(e -> System.out.println(e + ": " +
        Thread.currentThread().getName()))
      .limit(10)
      .parallel()
      .boxed()
      .collect(Collectors.toList());
    System.out.println(x);
  }
}
/* Output:
8: main
6: ForkJoinPool.commonPool-worker-5
3: ForkJoinPool.commonPool-worker-7
5: ForkJoinPool.commonPool-worker-5
1: ForkJoinPool.commonPool-worker-3
2: ForkJoinPool.commonPool-worker-6
4: ForkJoinPool.commonPool-worker-1
0: ForkJoinPool.commonPool-worker-4
7: ForkJoinPool.commonPool-worker-1
9: ForkJoinPool.commonPool-worker-2
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
*/
```

To show that `.parallel()` is indeed working, I've added a call to `peek()`, a stream function that is predominantly intended for debugging: it pulls a value out of the stream and does something with it, but doesn't affect the elements that are passed down the stream. Note that this interferes with thread behavior, but I'm just trying to show something here, not actually debug anything.

You can also see the addition of `boxed()`, which takes the stream of `int` and turns it into a stream of `Integer`.

Now we get multiple threads producing the different values, but it's also only producing 10 values as requested, rather than 1024 in order to produce 10.

Is it any faster? A better question is: When does it start to make sense? Certainly not with a set this small; the expense of context switching is likely to far outweigh any speedups from parallelism. It's a little hard to imagine *when* a simple sequence of numbers will make sense to generate in parallel. If you're using something with expensive generation, it might make sense—but that's all speculation. The only way to know is through testing. Remember the maxim: "First make it work, then make it fast—but only if you must." Combining `parallel()` and `limit()` is for experts only (and to be clear, I don't consider myself an expert here).

Parallel Streams Only *Look* Easy

Actually, in many cases parallel streams do in fact effortlessly produce results faster. But as you've seen, just slapping `parallel()` onto your `Stream` operations is not necessarily a safe thing to do. Before you use `parallel()`, you must understand how parallelism might help or hurt your operations. The fundamental error here is thinking that parallelism is always a good idea. It's not. Streams mean you don't need to rewrite all your code to run it in parallel. What streams don't do is replace the need to understand how parallelism works, and whether it will help achieve your goal.

Creating and Running Tasks

If you cannot achieve concurrency with parallel streams then you must create and run your own tasks. Later you'll see that the ideal Java 8 approach for running tasks is the `CompletableFuture`, but we'll introduce the concepts using more basic tools.

The history of Java concurrency starts with very primitive and problematic mechanisms, and is littered with various attempts at improvement. Those are primarily relegated to the Appendix: Low-Level Concurrency. Here, we will show a canonical form representing the simplest and best approaches to creating and running tasks. As with everything in concurrency, there are all kinds of variations, but these are either relegated to that appendix or beyond the scope of this book.

Tasks and Executors

In early versions of Java you used threads by creating your own `Thread` objects directly, even subclassing them to create your own specific "task-thread" objects. You called constructors by hand and started the threads yourself.

The overhead of creating all those threads turned out to be significant, and the hands-on approach is now discouraged. In Java 5, classes were added to handle *thread pooling* for you. Instead of creating a new subtype of `Thread` for each different type of task, you create a task as a separate type, then hand it to an `ExecutorService` to run that task. The `ExecutorService` manages the threads for you, and recycles threads rather than throwing them away after they've run a task.

To start, we'll create a task that does almost nothing. It "sleeps" (suspends execution) for 100 milliseconds, displays its identifier and the name of the `Thread` that's executing the task, then finishes:

```
// concurrent/NapTask.java
import onjava.Nap;

public class NapTask implements Runnable {
  final int id;
  public NapTask(int id) { this.id = id; }
  @Override public void run() {
    new Nap(0.1); // Seconds
    System.out.println(this + " " +
      Thread.currentThread().getName());
  }
  @Override public String toString() {
    return "NapTask[" + id + "]";
  }
}
```

This is simply a `Runnable`: a class containing a `run()` method. It doesn't include a mechanism for actually running the task.

We "sleep" using the `Nap` class:

```
// onjava/Nap.java
package onjava;
import java.util.concurrent.*;

public class Nap {
  public Nap(double t) { // Seconds
    try {
      TimeUnit.MILLISECONDS.sleep((int)(1000 * t));
    } catch(InterruptedException e) {
      throw new RuntimeException(e);
    }
  }
  public Nap(double t, String msg) {
    this(t);
    System.out.println(msg);
  }
}
```

To eliminate the visual noise of the exception handling, this is defined as a utility. The second constructor displays a message when it times out.

The call to `TimeUnit.MILLISECONDS.sleep()` gets the "current thread" and puts it to sleep for the time in the argument, which means that thread is suspended. This does *not* mean that the underlying processor stops. The OS switches it to some other task, for example running another window on your computer. Periodically the OS *task manager* checks to see if the `sleep()` has timed out. When it does, the thread is "woken up" and given more processing time.

You can see that `sleep()` throws a checked `InterruptedException`; this is an artifact from the original Java design that terminated tasks by abruptly breaking out of them. Because it tends to produce unstable states, termination has subsequently been discouraged. However, we must catch the exception for situations when termination is required or still happens.

To execute tasks, we'll start with the simplest approach, the *SingleThreadExecutor*:

```
// concurrent/SingleThreadExecutor.java
import java.util.concurrent.*;
import java.util.stream.*;
import onjava.*;

public class SingleThreadExecutor {
  public static void main(String[] args) {
    ExecutorService exec =
      Executors.newSingleThreadExecutor();
    IntStream.range(0, 10)
      .mapToObj(NapTask::new)
      .forEach(exec::execute);
    System.out.println("All tasks submitted");
    exec.shutdown();
    while(!exec.isTerminated()) {
      System.out.println(
        Thread.currentThread().getName() +
        " awaiting termination");
      new Nap(0.1);
    }
  }
}
/* Output:
All tasks submitted
main awaiting termination
main awaiting termination
NapTask[0] pool-1-thread-1
main awaiting termination
NapTask[1] pool-1-thread-1
NapTask[2] pool-1-thread-1
main awaiting termination
NapTask[3] pool-1-thread-1
main awaiting termination
main awaiting termination
NapTask[4] pool-1-thread-1
NapTask[5] pool-1-thread-1
main awaiting termination
NapTask[6] pool-1-thread-1
main awaiting termination
main awaiting termination
NapTask[7] pool-1-thread-1
main awaiting termination
```

```
NapTask[8] pool-1-thread-1
main awaiting termination
NapTask[9] pool-1-thread-1
*/
```

First note there is no `SingleThreadExecutor` class. `newSingleThreadExecutor()` is a factory in `Executors` that creates that particular kind of `ExecutorService`.[172]

I create ten `NapTasks` and submit them to the `ExecutorService`, which means they start running on their own. In the meantime, however, `main()` continues doing things. When I call `exec.shutdown()`, that tells the `ExecutorService` to finish tasks that have already been submitted, but not to accept any new tasks. At this point those tasks are still running, however, so we must wait until they complete before exiting `main()`. This is achieved by checking `exec.isTerminated()`, which becomes `true` after all the tasks complete.

Notice that the name of the thread in `main()` is `main`, and there is only one other thread, `pool-1-thread-1`. Also, the interleaved output shows that the two threads are indeed running concurrently.

If you simply call `exec.shutdown()`, the program will complete once all the tasks finish. That is, `while(!exec.isTerminated())` is not required:

```
// concurrent/SingleThreadExecutor2.java
import java.util.concurrent.*;
import java.util.stream.*;

public class SingleThreadExecutor2 {
  public static void main(String[] args)
    throws InterruptedException {
    ExecutorService exec =
      Executors.newSingleThreadExecutor();
    IntStream.range(0, 10)
      .mapToObj(NapTask::new)
      .forEach(exec::execute);
    exec.shutdown();
  }
}
/* Output:
```

[172] This is an interesting, albeit inconsistent, approach. Normally we expect different behavior on a common interface to be expressed with an explicit class.

```
NapTask[0] pool-1-thread-1
NapTask[1] pool-1-thread-1
NapTask[2] pool-1-thread-1
NapTask[3] pool-1-thread-1
NapTask[4] pool-1-thread-1
NapTask[5] pool-1-thread-1
NapTask[6] pool-1-thread-1
NapTask[7] pool-1-thread-1
NapTask[8] pool-1-thread-1
NapTask[9] pool-1-thread-1
*/
```

Once you call `exec.shutdown()`, trying to submit new tasks will throw a `RejectedExecutionException`:

```
// concurrent/MoreTasksAfterShutdown.java
import java.util.concurrent.*;

public class MoreTasksAfterShutdown {
  public static void main(String[] args) {
    ExecutorService exec =
      Executors.newSingleThreadExecutor();
    exec.execute(new NapTask(1));
    exec.shutdown();
    try {
      exec.execute(new NapTask(99));
    } catch(RejectedExecutionException e) {
      System.out.println(e);
    }
  }
}
/* Output:
java.util.concurrent.RejectedExecutionException: Task
NapTask[99] rejected from java.util.concurrent.ThreadPo
olExecutor@106d69c[Shutting down, pool size = 1, active
threads = 1, queued tasks = 0, completed tasks = 0]
NapTask[1] pool-1-thread-1
*/
```

The alternative to `exec.shutdown()` is `exec.shutdownNow()`, which, in addition to not accepting new tasks, will also try to stop any currently-running tasks by interrupting them. Again, interruption is messy, error-prone and discouraged.

Using More Threads

The point of using threads is (almost always) to get things done faster, so why should we limit ourselves to the *SingleThreadExecutor*? Look at the Javadoc for `Executors` and you'll see more options. For example *CachedThreadPool*:

```
// concurrent/CachedThreadPool.java
import java.util.concurrent.*;
import java.util.stream.*;

public class CachedThreadPool {
  public static void main(String[] args) {
    ExecutorService exec =
      Executors.newCachedThreadPool();
    IntStream.range(0, 10)
      .mapToObj(NapTask::new)
      .forEach(exec::execute);
    exec.shutdown();
  }
}
/* Output:
NapTask[1] pool-1-thread-2
NapTask[5] pool-1-thread-6
NapTask[4] pool-1-thread-5
NapTask[8] pool-1-thread-9
NapTask[0] pool-1-thread-1
NapTask[9] pool-1-thread-10
NapTask[2] pool-1-thread-3
NapTask[3] pool-1-thread-4
NapTask[6] pool-1-thread-7
NapTask[7] pool-1-thread-8
*/
```

When you run this program, you'll notice it finishes more quickly. This makes sense because, instead of using the same thread to sequentially run each task, every task gets its own thread so they all run in parallel. There seems to be no downside and it's hard to see why anyone would use a *SingleThreadExecutor*.

To understand the problem, we need a more complex task:

```
// concurrent/InterferingTask.java

public class InterferingTask implements Runnable {
  final int id;
  private static Integer val = 0;
  public InterferingTask(int id) { this.id = id; }
  @Override public void run() {
    for(int i = 0; i < 100; i++)
      val++;
    System.out.println(id + " " +
      Thread.currentThread().getName() + " " + val);
  }
}
```

Each task increments `val` one hundred times. This seems simple enough. Let's try it with a *CachedThreadPool*:

```
// concurrent/CachedThreadPool2.java
import java.util.concurrent.*;
import java.util.stream.*;

public class CachedThreadPool2 {
  public static void main(String[] args) {
    ExecutorService exec =
      Executors.newCachedThreadPool();
    IntStream.range(0, 10)
      .mapToObj(InterferingTask::new)
      .forEach(exec::execute);
    exec.shutdown();
  }
}
/* Output:
3 pool-1-thread-4 100
2 pool-1-thread-3 200
6 pool-1-thread-7 300
7 pool-1-thread-8 400
0 pool-1-thread-1 514
1 pool-1-thread-2 527
4 pool-1-thread-5 627
5 pool-1-thread-6 727
9 pool-1-thread-10 827
8 pool-1-thread-9 927
*/
```

The output is not what we expect *and* it varies from one run to the next. The problem is that all the tasks are trying to write to the single instance of `val`, and they are stepping on each other's toes. We say that such a class is *not thread-safe*. Let's see what happens with a *SingleThreadExecutor*:

```
// concurrent/SingleThreadExecutor3.java
import java.util.concurrent.*;
import java.util.stream.*;

public class SingleThreadExecutor3 {
  public static void main(String[] args)
    throws InterruptedException {
    ExecutorService exec =
      Executors.newSingleThreadExecutor();
    IntStream.range(0, 10)
      .mapToObj(InterferingTask::new)
      .forEach(exec::execute);
    exec.shutdown();
  }
}
/* Output:
0 pool-1-thread-1 100
1 pool-1-thread-1 200
2 pool-1-thread-1 300
3 pool-1-thread-1 400
4 pool-1-thread-1 500
5 pool-1-thread-1 600
6 pool-1-thread-1 700
7 pool-1-thread-1 800
8 pool-1-thread-1 900
9 pool-1-thread-1 1000
*/
```

Now we get consistent results, every time, despite the lack of thread-safety in `InterferingTask`. This is the primary benefit of a *SingleThreadExecutor*—because it runs one task at a time, those tasks never interfere with each other so thread safety is imposed. This phenomenon is called *thread confinement* because running tasks on a single thread confines their effects. Thread confinement limits speedup but can save a lot of difficult debugging and rewriting.

Producing Results

Because `InterferingTask` is a `Runnable`, it has no return value and can thus only produce results using *side effects*—manipulating its environment rather than returning a result. Side effects are one of the main problems in concurrent programming, for the reason we saw in `CachedThreadPool2.java`. The `val` in `InterferingTask` is called *mutable shared state*, and that's what gives you trouble: multiple tasks modifying the same variable at the same time produce what is called a *race condition*. The result depends on which task races to the finish line first and modifies the variable (and the myriad variations of other possibilities).

The best way to avoid race conditions is to avoid mutable shared state. We might call this the *selfish child principle*: Share nothing.

With `InterferingTask`, it would be nice to remove the side effects and just return a result from the task. To do this, we create a `Callable` rather than a `Runnable`:

```
// concurrent/CountingTask.java
import java.util.concurrent.*;

public class CountingTask implements Callable<Integer> {
  final int id;
  public CountingTask(int id) { this.id = id; }
  @Override public Integer call() {
    Integer val = 0;
    for(int i = 0; i < 100; i++)
      val++;
    System.out.println(id + " " +
      Thread.currentThread().getName() + " " + val);
    return val;
  }
}
```

`call()` produces its result entirely independently of all other CountingTasks, which means there is no mutable shared state.

An `ExecutorService` allows you to start every `Callable` in a collection using `invokeAll()`:

```
// concurrent/CachedThreadPool3.java
import java.util.*;
import java.util.concurrent.*;
import java.util.stream.*;

public class CachedThreadPool3 {
  public static Integer
  extractResult(Future<Integer> f) {
    try {
      return f.get();
    } catch(Exception e) {
      throw new RuntimeException(e);
    }
  }
  public static void main(String[] args)
    throws InterruptedException {
    ExecutorService exec =
      Executors.newCachedThreadPool();
    List<CountingTask> tasks =
      IntStream.range(0, 10)
        .mapToObj(CountingTask::new)
        .collect(Collectors.toList());
    List<Future<Integer>> futures =
      exec.invokeAll(tasks);
    Integer sum = futures.stream()
      .map(CachedThreadPool3::extractResult)
      .reduce(0, Integer::sum);
    System.out.println("sum = " + sum);
    exec.shutdown();
  }
}
/* Output:
0 pool-1-thread-1 100
9 pool-1-thread-1 100
1 pool-1-thread-2 100
2 pool-1-thread-3 100
3 pool-1-thread-4 100
6 pool-1-thread-7 100
7 pool-1-thread-8 100
4 pool-1-thread-5 100
5 pool-1-thread-6 100
8 pool-1-thread-9 100
```

```
sum = 1000
*/
```

Only after all the tasks are complete does `invokeAll()` return a `List` of `Future`, one `Future` for each task. A `Future` is a mechanism introduced in Java 5 that allows you to submit a task without waiting for it to complete. Here, we use `ExecutorService.submit()`:

```
// concurrent/Futures.java
import java.util.*;
import java.util.concurrent.*;
import java.util.stream.*;

public class Futures {
  public static void main(String[] args)
    throws InterruptedException, ExecutionException {
    ExecutorService exec =
      Executors.newSingleThreadExecutor();
    Future<Integer> f =
      exec.submit(new CountingTask(99));
    System.out.println(f.get());              // [1]
    exec.shutdown();
  }
}
/* Output:
99 pool-1-thread-1 100
100
*/
```

- **[1]** When you call `get()` on a `Future` whose task hasn't completed yet, the call blocks (waits) until the result is available.

But this means, in `CachedThreadPool3.java`, that `Future` seems redundant because `invokeAll()` doesn't even return until all the tasks have completed. However, here the `Future` is not used for the delayed result, but rather to capture any exceptions that might happen.

Notice also the messiness of extracting the results in `CachedThreadPool3.java`. `get()` throws exceptions, so `extractResult()` performs this extraction within a `Stream`.

Because a `Future` blocks when you call `get()`, it only puts off the problem of waiting for tasks to finish. Ultimately, `Futures` were deemed an ineffective solution and are now discouraged, in favor of Java 8's `CompletableFuture`, which we explore later in this chapter. Of course, you'll still encounter `Futures` in legacy libraries.

We can solve this problem in a much simpler and more elegant fashion using parallel `Streams`:

```
// concurrent/CountingStream.java
// {VisuallyInspectOutput}
import java.util.*;
import java.util.concurrent.*;
import java.util.stream.*;

public class CountingStream {
  public static void main(String[] args) {
    System.out.println(
      IntStream.range(0, 10)
        .parallel()
        .mapToObj(CountingTask::new)
        .map(ct -> ct.call())
        .reduce(0, Integer::sum));
  }
}
/* Output:
1 ForkJoinPool.commonPool-worker-3 100
8 ForkJoinPool.commonPool-worker-2 100
0 ForkJoinPool.commonPool-worker-6 100
2 ForkJoinPool.commonPool-worker-1 100
4 ForkJoinPool.commonPool-worker-5 100
9 ForkJoinPool.commonPool-worker-7 100
6 main 100
7 ForkJoinPool.commonPool-worker-4 100
5 ForkJoinPool.commonPool-worker-2 100
3 ForkJoinPool.commonPool-worker-3 100
1000
*/
```

Not only is this much easier to understand, all we needed to do was insert `parallel()` into an otherwise sequential operation and suddenly everything is running concurrently.

Lambdas and Method References as Tasks

With lambdas and method references, you're not constrained to using only `Runnables` and `Callables`. Because Java 8 supports lambdas and method references by matching signatures (that is, it supports *structural conformance*), we can pass arguments that are *not* `Runnables` or `Callables` to an `ExecutorService`:

```
// concurrent/LambdasAndMethodReferences.java
import java.util.concurrent.*;

class NotRunnable {
  public void go() {
    System.out.println("NotRunnable");
  }
}

class NotCallable {
  public Integer get() {
    System.out.println("NotCallable");
    return 1;
  }
}

public class LambdasAndMethodReferences {
  public static void main(String[] args)
    throws InterruptedException {
    ExecutorService exec =
      Executors.newCachedThreadPool();
    exec.submit(() -> System.out.println("Lambda1"));
    exec.submit(new NotRunnable()::go);
    exec.submit(() -> {
      System.out.println("Lambda2");
      return 1;
    });
    exec.submit(new NotCallable()::get);
    exec.shutdown();
  }
}
/* Output:
Lambda1
NotRunnable
```

```
Lambda2
NotCallable
*/
```

Here, the first two `submit()` calls can instead be calls to `execute()`. All `submit()` calls return `Futures`, which you can use to extract the result in the case of the second two calls.

Terminating Long-Running Tasks

Concurrent programs commonly use long-running tasks. A `Callable` task returns a value upon completion; although this gives it a finite lifetime, that can still be long. `Runnable` tasks are sometimes set up as background processes that run forever. You often need a way to stop both `Runnable` and `Callable` tasks before their normal completion, such as when you're shutting down a program.

The original Java design provided mechanisms (which, for backwards compatibility, still exist) to *interrupt* running tasks; the interruption mechanisms include issues around blocking. Interrupting tasks is messy and complicated because you must understand all possible states from which that interruption might occur, along with the possible resulting data loss. Using interruption is considered an antipattern, but we are still forced to catch `InterruptedException` because of the backward-compatibility residue of the design.

The best approach to task termination is to set a flag that the task periodically checks. Then the task can go through its own shutdown process and terminate gracefully. Instead of pulling the plug on a task at some random time, you ask the task to terminate itself when it reaches a good point. This always produces better results than interruption, along with more sensible code that is easier to understand.

Terminating a task this way sounds simple enough: set a `boolean` flag that the task can see. Write the task so it periodically checks the flag and performs graceful termination. That is in fact what you do, but there's a complication: Our old nemesis, shared mutable state. If the flag can be manipulated by another task, then there are collision possibilities.

When studying Java literature you'll find numerous approaches to this problem, very often using the `volatile` keyword. We shall use a simpler technique and avoid

all the vagaries of `volatile`, which are only covered in the Appendix: Low-Level Concurrency.

Java 5 introduced the `Atomic` classes, which provide a set of types you can use without worrying about concurrency problems. We'll add an `AtomicBoolean` flag that tells a task to clean itself up and exit:

```
// concurrent/QuittableTask.java
import java.util.concurrent.atomic.AtomicBoolean;
import onjava.Nap;

public class QuittableTask implements Runnable {
  final int id;
  public QuittableTask(int id) { this.id = id; }
  private AtomicBoolean running =
    new AtomicBoolean(true);
  public void quit() { running.set(false); }
  @Override public void run() {
    while(running.get())                  // [1]
      new Nap(0.1);
    System.out.print(id + " ");           // [2]
  }
}
```

Although multiple tasks can successfully call `quit()` at the same instant, the `AtomicBoolean` prevents more than one of those tasks from actually modifying `running` at the same time, rendering the `quit()` method thread-safe.

- **[1]**: As long as the `running` flag is `true`, this task's `run()` method will continue.
- **[2]**: The display only happens as the task exits.

The need to make `running` an `AtomicBoolean` demonstrates one of the most fundamental difficulties when writing concurrent Java programs. If you make `running` an ordinary `boolean`, you might never see the problem in an executing program. Indeed, in this example the chances are that you'd never have any problems—and yet that code would still be unsafe. It can be difficult or impossible to write a test that demonstrates that problem. Thus you don't have any immediate feedback to tell you that you've done something wrong. Often, the only way for you to write thread-safe code is just by knowing all the subtle places where things can go wrong.

As a test, we'll start a lot of `QuittableTasks` and then shut them down. Try playing with larger values of COUNT:

```
// concurrent/QuittingTasks.java
import java.util.*;
import java.util.stream.*;
import java.util.concurrent.*;
import onjava.Nap;

public class QuittingTasks {
  public static final int COUNT = 150;
  public static void main(String[] args) {
    ExecutorService es =
      Executors.newCachedThreadPool();
    List<QuittableTask> tasks =
      IntStream.range(1, COUNT)
        .mapToObj(QuittableTask::new)
        .peek(qt -> es.execute(qt))
        .collect(Collectors.toList());
    new Nap(1);
    tasks.forEach(QuittableTask::quit);
    es.shutdown();
  }
}
/* Output:
11 23 20 12 24 16 19 15 35 147 32 27 7 4 28 31 8 83 3 1
13 9 5 2 6 18 14 17 21 25 22 26 29 30 33 34 37 41 38 46
45 49 50 53 57 58 54 69 104 112 40 73 74 115 116 70 119
77 81 56 85 78 82 111 86 90 48 89 36 108 107 44 55 52
43 60 63 59 64 71 68 67 75 76 72 79 80 84 87 88 66 91
10 95 65 96 94 92 62 100 61 93 47 39 51 99 103 128 123
127 124 140 120 139 136 135 143 148 144 105 102 131 101
132 98 97 149 137 134 42 106 110 109 114 133 113 117
118 130 129 126 121 125 122 138 141 145 142 146
*/
```

I use `peek()` to pass the `QuittableTasks` to the `ExecutorService` before collecting those tasks into a `List`.

`main()` prevents the program from exiting as long as any tasks remain running. The tasks don't shut down in the same order they were created, even though the `quit()` method *is* called for each task in order. Independently-running tasks don't respond to signals deterministically.

CompletableFutureS

As an introduction, here's a translation of `QuittingTasks.java` using `CompletableFutures`:

```
// concurrent/QuittingCompletable.java
import java.util.*;
import java.util.stream.*;
import java.util.concurrent.*;
import onjava.Nap;

public class QuittingCompletable {
  public static void main(String[] args) {
    List<QuittableTask> tasks =
      IntStream.range(1, QuittingTasks.COUNT)
        .mapToObj(QuittableTask::new)
        .collect(Collectors.toList());
    List<CompletableFuture<Void>> cfutures =
      tasks.stream()
        .map(CompletableFuture::runAsync)
        .collect(Collectors.toList());
    new Nap(1);
    tasks.forEach(QuittableTask::quit);
    cfutures.forEach(CompletableFuture::join);
  }
}
/* Output:
6 7 5 9 11 12 13 14 15 16 17 18 19 20 21 22 23 10 24 26
27 28 29 30 31 32 25 33 8 36 37 38 39 40 41 42 43 44 45
46 47 48 49 50 51 52 53 54 55 56 57 58 59 60 61 62 63
64 65 66 67 68 69 70 71 72 73 74 75 76 77 78 79 80 81
82 83 84 85 86 87 88 89 90 91 92 93 94 95 96 97 98 99
100 101 102 103 104 105 106 107 108 34 110 111 112 113
114 115 116 117 118 119 120 121 109 123 124 125 126 127
128 129 130 131 35 132 122 134 133 136 135 138 140 141
142 143 144 145 146 147 148 149 137 139 3 1 2 4
*/
```

`tasks` is a `List<QuittableTask>` just as in `QuittingTasks.java`, but in this example, there's no `peek()` that submits each `QuittableTask` to an `ExecutorService`.

Instead, during the creation of `cfutures` each task is handed to `CompletableFuture::runAsync`. This executes `QuittableTask.run()` and returns a `CompletableFuture<Void>`. Because `run()` doesn't return anything, I only use the `CompletableFuture` in this case to call `join()` to wait for it to finish.

The important thing to notice in this example is that no `ExecutorService` is required to run the tasks. This is managed by the `CompletableFuture` (although there are options to provide your own `ExecutorService`). You also don't need to call `shutdown()`; in fact, unless you explicitly call `join()` as I do here, the program will exit as soon as it can, without waiting for tasks to complete.

This example is just a starting point. You'll soon see that `CompletableFutures` are capable of much more.

Basic Usage

Here's a class with a `static` method `work()` that performs some kind of work on objects of that class:

```
// concurrent/Machina.java
import onjava.Nap;

public class Machina {
  public enum State {
    START, ONE, TWO, THREE, END;
    State step() {
      if(equals(END)) return END;
      return values()[ordinal() + 1];
    }
  }
  private State state = State.START;
  private final int id;
  public Machina(int id) { this.id = id; }
  public static Machina work(Machina m) {
    if(!m.state.equals(State.END)){
      new Nap(0.1);
      m.state = m.state.step();
    }
    System.out.println(m);
    return m;
```

```
  }
  @Override public String toString() {
    return "Machina" + id + ": " +
      (state.equals(State.END)? "complete" : state);
  }
}
```

This is a *finite state machine*, a trivial one because it has no branches...it just traverses a single path from beginning to end. The `work()` method moves the machine from one state to the next and requires 100 milliseconds to do that "work."

One thing we can do with a `CompletableFuture` is to wrap it around an object of interest using `completedFuture()`:

```
// concurrent/CompletedMachina.java
import java.util.concurrent.*;

public class CompletedMachina {
  public static void main(String[] args) {
    CompletableFuture<Machina> cf =
      CompletableFuture.completedFuture(
        new Machina(0));
    try {
      Machina m = cf.get();  // Doesn't block
    } catch(InterruptedException |
            ExecutionException e) {
      throw new RuntimeException(e);
    }
  }
}
```

`completedFuture()` creates a `CompletableFuture` which is "already complete." The only useful thing to do with such a future is to `get()` the object inside, so this doesn't seem that helpful at first.

Note that the `CompletableFuture` is typed to the object it contains. This is important.

Normally, `get()` blocks the calling thread while it waits on the result. This block can be broken out of via either an `InterruptedException` or an `ExecutionException`. In this case, the blocking never happens because the `CompletableFuture` is already complete, so the answer is instantly available.

Things get much more interesting when, once we've wrapped our `Machina` in a `CompletableFuture`, we discover that we can add operations onto that `CompletableFuture` to work on the contained object:

```
// concurrent/CompletableApply.java
import java.util.concurrent.*;

public class CompletableApply {
  public static void main(String[] args) {
    CompletableFuture<Machina> cf =
      CompletableFuture.completedFuture(
        new Machina(0));
    CompletableFuture<Machina> cf2 =
      cf.thenApply(Machina::work);
    CompletableFuture<Machina> cf3 =
      cf2.thenApply(Machina::work);
    CompletableFuture<Machina> cf4 =
      cf3.thenApply(Machina::work);
    CompletableFuture<Machina> cf5 =
      cf4.thenApply(Machina::work);
  }
}
/* Output:
Machina0: ONE
Machina0: TWO
Machina0: THREE
Machina0: complete
*/
```

`thenApply()` applies a `Function` that takes an input and produces an output. In this case, the `work()` `Function` produces the same type that it takes in, so each resulting `CompletableFuture` is still typed as `Machina`, but (similar to `map()` in `Streams`) the `Function` can also return different types, which would be reflected in the return type.

You see something essential about `CompletableFutures` here: They automatically unwrap and re-wrap the object they are carrying when you perform an operation. This way you don't get caught up in messy details, which makes writing and understanding code much easier.

We can eliminate the intermediate variables and just chain the operations together, the way we do with `Streams`:

```
// concurrent/CompletableApplyChained.java
import java.util.concurrent.*;
import onjava.Timer;

public class CompletableApplyChained {
  public static void main(String[] args) {
    Timer timer = new Timer();
    CompletableFuture<Machina> cf =
      CompletableFuture.completedFuture(
        new Machina(0))
      .thenApply(Machina::work)
      .thenApply(Machina::work)
      .thenApply(Machina::work)
      .thenApply(Machina::work);
    System.out.println(timer.duration());
  }
}
/* Output:
Machina0: ONE
Machina0: TWO
Machina0: THREE
Machina0: complete
521
*/
```

Here we've also added a `Timer` which shows us that each step adds 100 milliseconds, and there's some additional overhead.

One important benefit of `CompletableFutures` is that they encourage the use of the *selfish child principle* (share nothing). By default, using `thenApply()` to apply a function doesn't communicate with anyone—it just takes an argument and returns a result. This is a foundation of functional programming, and one reason it works so well for concurrency.[173] Parallel streams and `CompletableFutures` are designed to support these principles. As long as you don't decide to share data anyway (sharing is very easy to do, even accidentally) you can write relatively safe concurrent programs.

Calling `thenApply()` starts the operation, and in this case the *creation* of the `CompletableFuture` isn't finished until all the tasks are completed. Although this is sometimes useful, it's typically more valuable to start all tasks so you can move on

[173] And no, there can never be a pure functional Java. The best we can hope for is an entirely new language that runs on the JVM.

and do something else while they run. We accomplish this by adding `Async` to the end of the operation:

```
// concurrent/CompletableApplyAsync.java
import java.util.concurrent.*;
import onjava.*;

public class CompletableApplyAsync {
  public static void main(String[] args) {
    Timer timer = new Timer();
    CompletableFuture<Machina> cf =
      CompletableFuture.completedFuture(
        new Machina(0))
       .thenApplyAsync(Machina::work)
       .thenApplyAsync(Machina::work)
       .thenApplyAsync(Machina::work)
       .thenApplyAsync(Machina::work);
    System.out.println(timer.duration());
    System.out.println(cf.join());
    System.out.println(timer.duration());
  }
}
/* Output:
103
Machina0: ONE
Machina0: TWO
Machina0: THREE
Machina0: complete
Machina0: complete
545
*/
```

A *synchronous* call (the kind we normally make) means "return when you are finished working," whereas an *asynchronous* call means "return right away but keep working in the background." As you can see, the creation of `cf` happens much faster now. Each call to `thenApplyAsync()` returns immediately so the next call can be made, and the whole chaining sequence completes much more quickly than before.

So quickly, in fact, that without the call to `cf.join()` the program exits before finishing its work (try taking out that line). The call to `join()` blocks the `main()` thread from progressing until the `cf` operations complete, and we can see that most of the time was indeed spent there.

This *async* ability to "return right away" requires some undercover work by the `CompletableFuture` library. In particular, it must store the chain of operations you require as a set of *callbacks*. When the first background operation completes and returns, the second one must take the resulting `Machina` and begin work, and when that one completes, the next operation takes over, etc. But without our ordinary sequence of function calls, controlled via the program call stack, this order would be lost so it is instead stored using callbacks—a table of function addresses.

Fortunately, that's all you need to know about callbacks. Programmers refer to the mess you get into doing it by hand as "callback hell." With `Async` calls, the `CompletableFuture` manages all the callbacks for you. And unless you know something specific about your system that changes things, you'll probably want to use `Async` calls.

Other Operations

When you look at the Javadocs for `CompletableFuture`, you'll see it has many methods, but the majority of this number come from variations on the different operations. For example, there's `thenApply()`, `thenApplyAsync()`, and a second form of `thenApplyAsync()` that takes an `Executor` on which to run the tasks (in this book we ignore the `Executor` option).

Here's an example that shows all the "basic" operations, ones that don't involve either combining two `CompletableFutures` or exceptions (we shall look at those later). First, here are two utilities we shall reuse to provide brevity and convenience:

```
// concurrent/CompletableUtilities.java
package onjava;
import java.util.concurrent.*;

public class CompletableUtilities {
  // Get and show value stored in a CF:
  public static void showr(CompletableFuture<?> c) {
    try {
      System.out.println(c.get());
    } catch(InterruptedException
            | ExecutionException e) {
      throw new RuntimeException(e);
    }
```

```
  }
  // For CF operations that have no value:
  public static void voidr(CompletableFuture<Void> c) {
    try {
      c.get(); // Returns void
    } catch(InterruptedException
            | ExecutionException e) {
      throw new RuntimeException(e);
    }
  }
}
```

`showr()` calls `get()` on a `CompletableFuture<Integer>` and displays the result, catching the two possible exceptions. `voidr()` is a version of `showr()` for `CompletableFuture<Void>`, that is, `CompletableFutures` that only exist to show when a task completes or fails.

For simplicity, the following `CompletableFutures` just wrap `Integers`. `cfi()` is a convenience method that wraps an `int` inside a completed `CompletableFuture<Integer>`:

```
// concurrent/CompletableOperations.java
import java.util.concurrent.*;
import static onjava.CompletableUtilities.*;

public class CompletableOperations {
  static CompletableFuture<Integer> cfi(int i) {
    return
      CompletableFuture.completedFuture(
        Integer.valueOf(i));
  }
  public static void main(String[] args) {
    showr(cfi(1)); // Basic test
    voidr(cfi(2).runAsync(() ->
      System.out.println("runAsync")));
    voidr(cfi(3).thenRunAsync(() ->
      System.out.println("thenRunAsync")));
    voidr(CompletableFuture.runAsync(() ->
      System.out.println("runAsync is static")));
    showr(CompletableFuture.supplyAsync(() -> 99));
    voidr(cfi(4).thenAcceptAsync(i ->
      System.out.println("thenAcceptAsync: " + i)));
```

```
    showr(cfi(5).thenApplyAsync(i -> i + 42));
    showr(cfi(6).thenComposeAsync(i -> cfi(i + 99)));
    CompletableFuture<Integer> c = cfi(7);
    c.obtrudeValue(111);
    showr(c);
    showr(cfi(8).toCompletableFuture());
    c = new CompletableFuture<>();
    c.complete(9);
    showr(c);
    c = new CompletableFuture<>();
    c.cancel(true);
    System.out.println("cancelled: " +
      c.isCancelled());
    System.out.println("completed exceptionally: " +
      c.isCompletedExceptionally());
    System.out.println("done: " + c.isDone());
    System.out.println(c);
    c = new CompletableFuture<>();
    System.out.println(c.getNow(777));
    c = new CompletableFuture<>();
    c.thenApplyAsync(i -> i + 42)
      .thenApplyAsync(i -> i * 12);
    System.out.println("dependents: " +
      c.getNumberOfDependents());
    c.thenApplyAsync(i -> i / 2);
    System.out.println("dependents: " +
      c.getNumberOfDependents());
  }
}
/* Output:
1
runAsync
thenRunAsync
runAsync is static
99
thenAcceptAsync: 4
47
105
111
8
9
cancelled: true
```

```
completed exceptionally: true
done: true
java.util.concurrent.CompletableFuture@1629346[Complete
d exceptionally]
777
dependents: 1
dependents: 2
*/
```

`main()` contains a sequence of tests that can be referred to by their `int` values. `cfi(1)` demonstrates that `showr()` works properly. `cfi(2)` is an example of calling `runAsync()`. A `Runnable` produces no return value so the result is a `CompletableFuture<Void>` and `voidr()` is used.

Notice with `cfi(3)` that `thenRunAsync()` appears to be the same as `runAsync()`. The difference is shown in the subsequent test: `runAsync()` is a `static` method, so you normally wouldn't call it as in `cfi(2)`. Instead you'd use it as in `QuittingCompletable.java`. The subsequent test shows that `supplyAsync()` is also `static`, but requires a `Supplier` rather than a `Runnable` and produces a `CompletableFuture<Integer>` instead of a `CompletableFuture<Void>`.

"**then**" methods apply operations to an existing `CompletableFuture<Integer>`. Unlike `thenRunAsync()`, the "**then**" methods for `cfi(4)`, `cfi(5)` and `cfi(6)` are handed the unwrapped `Integer` as their argument. As you can see by the use of `voidr()`, `thenAcceptAsync()` takes a `Consumer` and so doesn't produce a result. `thenApplyAsync()` takes a `Function` and so produces a result (which can be of a different type than its argument). `thenComposeAsync()` is quite similar to `thenApplyAsync()` except that its `Function` must produce a result that is *already wrapped* in a `CompletableFuture`.

The `cfi(7)` example demonstrates `obtrudeValue()`, which forces a value in as the result. `cfi(8)` uses `toCompletableFuture()` which produces a `CompletableFuture` from this `CompletionStage`. `c.complete(9)` shows how you can complete a future by giving it a result (versus `obtrudeValue()` which could then force its result in to replace this one).

If you `cancel()` a `CompletableFuture`, it also becomes *done* and is completed exceptionally.

The `getNow()` method returns either the completed value of the `CompletableFuture`, or the substitute argument of `getNow()` if the future hasn't yet completed.

Finally, we look at the concept of *dependents*. If we chain two `thenApplyAsync()` calls onto a `CompletableFuture`, the number of dependents is still one. But if we attach another `thenApplyAsync()` directly to `c`, we now have two dependents: The chain of two and the additional one. This shows that you can have a single `CompletionStage` that, when it completes, can fork multiple new tasks based on its result.

Combining `CompletableFuture`s

The second category of `CompletableFuture` methods takes two `CompletableFutures` and combines them in various ways. One `CompletableFuture` typically finishes before another, as if the two are in a race. These methods allow you to handle the results in different ways.

To test this, we'll create a task that takes as one of its arguments the amount of time to complete, so we can control which `CompletableFuture` finishes first:

```
// concurrent/Workable.java
import java.util.concurrent.*;
import onjava.Nap;

public class Workable {
  String id;
  final double duration;
  public Workable(String id, double duration) {
    this.id = id;
    this.duration = duration;
  }
  @Override public String toString() {
    return "Workable[" + id + "]";
  }
  public static Workable work(Workable tt) {
    new Nap(tt.duration); // Seconds
    tt.id = tt.id + "W";
    System.out.println(tt);
    return tt;
  }
  public static CompletableFuture<Workable>
  make(String id, double duration) {
    return
```

```
      CompletableFuture.completedFuture(
        new Workable(id, duration))
      .thenApplyAsync(Workable::work);
  }
}
```

In `make()`, the `work()` method is applied to the `CompletableFuture`. `work()` takes `duration` to complete, then it attaches the letter `W` to `id` to indicate that work has been completed.

Now we can create multiple competing `CompletableFutures` and connect them using the various methods in the `CompletableFuture` library:

```
// concurrent/DualCompletableOperations.java
import java.util.concurrent.*;
import static onjava.CompletableUtilities.*;

public class DualCompletableOperations {
  static CompletableFuture<Workable> cfA, cfB;
  static void init() {
    cfA = Workable.make("A", 0.15);
    cfB = Workable.make("B", 0.10); // Always wins
  }
  static void join() {
    cfA.join();
    cfB.join();
    System.out.println("*****************");
  }
  public static void main(String[] args) {
    init();
    voidr(cfA.runAfterEitherAsync(cfB, () ->
      System.out.println("runAfterEither")));
    join();

    init();
    voidr(cfA.runAfterBothAsync(cfB, () ->
      System.out.println("runAfterBoth")));
    join();

    init();
    showr(cfA.applyToEitherAsync(cfB, w -> {
      System.out.println("applyToEither: " + w);
```

```
      return w;
    }));
    join();

    init();
    voidr(cfA.acceptEitherAsync(cfB, w -> {
      System.out.println("acceptEither: " + w);
    }));
    join();

    init();
    voidr(cfA.thenAcceptBothAsync(cfB, (w1, w2) -> {
      System.out.println("thenAcceptBoth: "
        + w1 + ", " + w2);
    }));
    join();

    init();
    showr(cfA.thenCombineAsync(cfB, (w1, w2) -> {
      System.out.println("thenCombine: "
        + w1 + ", " + w2);
      return w1;
    }));
    join();

    init();
    CompletableFuture<Workable>
      cfC = Workable.make("C", 0.08),
      cfD = Workable.make("D", 0.09);
    CompletableFuture.anyOf(cfA, cfB, cfC, cfD)
      .thenRunAsync(() ->
        System.out.println("anyOf"));
    join();

    init();
    cfC = Workable.make("C", 0.08);
    cfD = Workable.make("D", 0.09);
    CompletableFuture.allOf(cfA, cfB, cfC, cfD)
      .thenRunAsync(() ->
        System.out.println("allOf"));
    join();
  }
```

```
}
/* Output:
Workable[BW]
runAfterEither
Workable[AW]
*****************
Workable[BW]
Workable[AW]
runAfterBoth
*****************
Workable[BW]
applyToEither: Workable[BW]
Workable[BW]
Workable[AW]
*****************
Workable[BW]
acceptEither: Workable[BW]
Workable[AW]
*****************
Workable[BW]
Workable[AW]
thenAcceptBoth: Workable[AW], Workable[BW]
*****************
Workable[BW]
Workable[AW]
thenCombine: Workable[AW], Workable[BW]
Workable[AW]
*****************
Workable[CW]
anyOf
Workable[DW]
Workable[BW]
Workable[AW]
*****************
Workable[CW]
Workable[DW]
Workable[BW]
Workable[AW]
*****************
allOf
*/
```

For easy access, `cfA` and `cfB` are `static`. `init()` initializes the two with `"B"` always given the shorter delay and thus always "winning." `join()` is another convenience method to call `join()` on both methods and display a border.

All of these "dual" methods take one `CompletableFuture` as the object to call the method upon, and a second `CompletableFuture` as the first argument, followed by the operation to perform.

You can see by the use of `showr()` and `voidr()` that "run" and "accept" are terminal operations, while "apply" and "combine" produce new payload-bearing `CompletableFutures`.

The names of the methods are self-explanatory, and you can verify this by looking at the output. One particularly interesting method is `combineAsync()`, which waits for both `CompletableFutures` to complete and then hands *both* of them to a `BiFunction` which can then join the results into the payload of the resulting `CompletableFuture`.

A Simulation

As an example of how you might wire together a sequence of operations using `CompletableFutures`, let us simulate the process of making a cake. In the first stage, we prepare and combine the ingredients into a batter:

```
// concurrent/Batter.java
import java.util.concurrent.*;
import onjava.Nap;

public class Batter {
  static class Eggs {}
  static class Milk {}
  static class Sugar {}
  static class Flour {}
  static <T> T prepare(T ingredient) {
    new Nap(0.1);
    return ingredient;
  }
  static <T> CompletableFuture<T> prep(T ingredient) {
    return CompletableFuture
      .completedFuture(ingredient)
```

```
      .thenApplyAsync(Batter::prepare);
  }
  public static CompletableFuture<Batter> mix() {
    CompletableFuture<Eggs> eggs = prep(new Eggs());
    CompletableFuture<Milk> milk = prep(new Milk());
    CompletableFuture<Sugar> sugar = prep(new Sugar());
    CompletableFuture<Flour> flour = prep(new Flour());
    CompletableFuture
      .allOf(eggs, milk, sugar, flour)
      .join();
    new Nap(0.1); // Mixing time
    return
      CompletableFuture.completedFuture(new Batter());
  }
}
```

Each ingredient takes some time to prepare. `allOf()` waits for all ingredients to be ready, then some more time is required to mix it into a batter.

Next we put the single batch of batter into four pans and bake it. The product is returned as a `Stream` of `CompletableFutures`:

```
// concurrent/Baked.java
import java.util.concurrent.*;
import java.util.stream.*;
import onjava.Nap;

public class Baked {
  static class Pan {}
  static Pan pan(Batter b) {
    new Nap(0.1);
    return new Pan();
  }
  static Baked heat(Pan p) {
    new Nap(0.1);
    return new Baked();
  }
  static CompletableFuture<Baked>
  bake(CompletableFuture<Batter> cfb) {
    return cfb
      .thenApplyAsync(Baked::pan)
      .thenApplyAsync(Baked::heat);
```

```
  }
  public static
  Stream<CompletableFuture<Baked>> batch() {
    CompletableFuture<Batter> batter = Batter.mix();
    return Stream.of(bake(batter), bake(batter),
                     bake(batter), bake(batter));
  }
}
```

Finally, we create a batch of `Frosting` and frost our cakes with it:

```
// concurrent/FrostedCake.java
import java.util.concurrent.*;
import java.util.stream.*;
import onjava.Nap;

final class Frosting {
  private Frosting() {}
  static CompletableFuture<Frosting> make() {
    new Nap(0.1);
    return CompletableFuture
      .completedFuture(new Frosting());
  }
}

public class FrostedCake {
  public FrostedCake(Baked baked, Frosting frosting) {
    new Nap(0.1);
  }
  @Override public String toString() {
    return "FrostedCake";
  }
  public static void main(String[] args) {
    Baked.batch().forEach(baked -> baked
      .thenCombineAsync(Frosting.make(),
        (cake, frosting) ->
          new FrostedCake(cake, frosting))
      .thenAcceptAsync(System.out::println)
      .join());
  }
}
```

Once you're comfortable with the ideas behind `CompletableFutures` they are relatively easy to use.

Exceptions

The same way a `CompletableFuture` wraps the objects within the processing chain, it also buffers you from exceptions. These don't appear to the caller while processing, but only when you try to extract the result. To show how they work, we'll start by creating a class that throws an exception under certain conditions:

```
// concurrent/Breakable.java
import java.util.concurrent.*;

public class Breakable {
  String id;
  private int failcount;
  public Breakable(String id, int failcount) {
    this.id = id;
    this.failcount = failcount;
  }
  @Override public String toString() {
    return "Breakable_" + id +
      " [" + failcount + "]";
  }
  public static Breakable work(Breakable b) {
    if(--b.failcount == 0) {
      System.out.println(
        "Throwing Exception for " + b.id + "");
      throw new RuntimeException(
        "Breakable_" + b.id + " failed");
    }
    System.out.println(b);
    return b;
  }
}
```

With a positive `failcount`, every time you pass the object to the `work()` method it decrements that `failcount`. When it goes to zero, `work()` throws an exception. If you give it a `failcount` of zero, it never throws an exception.

Notice that it reports throwing an exception as that exception is thrown.

In the following `test()` method, `work()` is applied to a `Breakable` multiple times, so if `failcount` is within range the exception is thrown. However, in tests `A` through `E`, you can see from the output that the exceptions are thrown, but they never emerge:

```
// concurrent/CompletableExceptions.java
import java.util.concurrent.*;

public class CompletableExceptions {
  static CompletableFuture<Breakable>
  test(String id, int failcount) {
    return
      CompletableFuture.completedFuture(
        new Breakable(id, failcount))
          .thenApply(Breakable::work)
          .thenApply(Breakable::work)
          .thenApply(Breakable::work)
          .thenApply(Breakable::work);
  }
  public static void main(String[] args) {
    // Exceptions don't appear ...
    test("A", 1);
    test("B", 2);
    test("C", 3);
    test("D", 4);
    test("E", 5);
    // ... until you try to fetch the value:
    try {
      test("F", 2).get(); // or join()
    } catch(Exception e) {
      System.out.println(e.getMessage());
    }
    // Test for exceptions:
    System.out.println(
      test("G", 2).isCompletedExceptionally());
    // Counts as "done":
    System.out.println(test("H", 2).isDone());
    // Force an exception:
    CompletableFuture<Integer> cfi =
      new CompletableFuture<>();
    System.out.println("done? " + cfi.isDone());
    cfi.completeExceptionally(
      new RuntimeException("forced"));
```

```
    try {
      cfi.get();
    } catch(Exception e) {
      System.out.println(e.getMessage());
    }
  }
}
/* Output:
Throwing Exception for A
Breakable_B [1]
Throwing Exception for B
Breakable_C [2]
Breakable_C [1]
Throwing Exception for C
Breakable_D [3]
Breakable_D [2]
Breakable_D [1]
Throwing Exception for D
Breakable_E [4]
Breakable_E [3]
Breakable_E [2]
Breakable_E [1]
Breakable_F [1]
Throwing Exception for F
java.lang.RuntimeException: Breakable_F failed
Breakable_G [1]
Throwing Exception for G
true
Breakable_H [1]
Throwing Exception for H
true
done? false
java.lang.RuntimeException: forced
*/
```

Tests `A` through `E` run up to the point they throw their exception, and then … nothing. Only when calling `get()` in test `F` do we see the thrown exception.

Test `G` shows that you can check first to see whether an exception was thrown during processing, without throwing that exception. However, test `H` tells us that an exception still qualifies as being "done," regardless of whether it was actually successful.

The last section of code shows how you can insert an exception into a `CompletableFuture`, regardless of whether there's any failure.

Rather than using a crude try-catch when joining or getting the result, we use the more sophisticated mechanisms provided by `CompletableFuture` to automatically respond to exceptions. You do this using the same form we've seen for all `CompletableFutures`: Insert a `CompletableFuture` call in the chain. There are three options: `exceptionally()`, `handle()`, and `whenComplete()`:

```
// concurrent/CatchCompletableExceptions.java
import java.util.concurrent.*;

public class CatchCompletableExceptions {
  static void handleException(int failcount) {
    // Call the Function only if there's an
    // exception, must produce same type as came in:
    CompletableExceptions
      .test("exceptionally", failcount)
      .exceptionally((ex) -> { // Function
        if(ex == null)
          System.out.println("I don't get it yet");
        return new Breakable(ex.getMessage(), 0);
      })
      .thenAccept(str ->
        System.out.println("result: " + str));

    // Create a new result (recover):
    CompletableExceptions
      .test("handle", failcount)
      .handle((result, fail) -> { // BiFunction
        if(fail != null)
          return "Failure recovery object";
        else
          return result + " is good";
      })
      .thenAccept(str ->
        System.out.println("result: " + str));

    // Do something but pass the same result through:
    CompletableExceptions
      .test("whenComplete", failcount)
      .whenComplete((result, fail) -> { // BiConsumer
```

```
        if(fail != null)
          System.out.println("It failed");
        else
          System.out.println(result + " OK");
      })
      .thenAccept(r ->
        System.out.println("result: " + r));
  }
  public static void main(String[] args) {
    System.out.println("**** Failure Mode ****");
    handleException(2);
    System.out.println("**** Success Mode ****");
    handleException(0);
  }
}
/* Output:
**** Failure Mode ****
Breakable_exceptionally [1]
Throwing Exception for exceptionally
result: Breakable_java.lang.RuntimeException:
Breakable_exceptionally failed [0]
Breakable_handle [1]
Throwing Exception for handle
result: Failure recovery object
Breakable_whenComplete [1]
Throwing Exception for whenComplete
It failed
**** Success Mode ****
Breakable_exceptionally [-1]
Breakable_exceptionally [-2]
Breakable_exceptionally [-3]
Breakable_exceptionally [-4]
result: Breakable_exceptionally [-4]
Breakable_handle [-1]
Breakable_handle [-2]
Breakable_handle [-3]
Breakable_handle [-4]
result: Breakable_handle [-4] is good
Breakable_whenComplete [-1]
Breakable_whenComplete [-2]
Breakable_whenComplete [-3]
Breakable_whenComplete [-4]
```

```
Breakable_whenComplete [-4] OK
result: Breakable_whenComplete [-4]
*/
```

The `exceptionally()` argument only runs if there's been an exception up to that point. `exceptionally()` is restrictive in that the `Function` can only return a value of the same type that came in. `exceptionally()` recovers to a workable state by inserting a good object back into the stream.

`handle()` is always called, and you must check if `fail` is true to see whether an exception occurred. But `handle()` can produce any new type, so it allows you to perform processing and not just recover as with `exceptionally()`.

`whenComplete()` is like `handle()` in that you must test for failure, but the argument is a consumer and doesn't modify the `result` object that's being passed through.

Stream Exceptions

Let's see how `CompletableFuture` exceptions differ from those for `Streams` by modifying `CompletableExceptions.java`:

```
// concurrent/StreamExceptions.java
import java.util.concurrent.*;
import java.util.stream.*;

public class StreamExceptions {
  static Stream<Breakable>
  test(String id, int failcount) {
    return
      Stream.of(new Breakable(id, failcount))
        .map(Breakable::work)
        .map(Breakable::work)
        .map(Breakable::work)
        .map(Breakable::work);
  }
  public static void main(String[] args) {
    // No operations are even applied ...
    test("A", 1);
    test("B", 2);
    Stream<Breakable> c = test("C", 3);
    test("D", 4);
```

```
    test("E", 5);
    // ... until there's a terminal operation:
    System.out.println("Entering try");
    try {
      c.forEach(System.out::println);   // [1]
    } catch(Exception e) {
      System.out.println(e.getMessage());
    }
  }
}
/* Output:
Entering try
Breakable_C [2]
Breakable_C [1]
Throwing Exception for C
Breakable_C failed
*/
```

With `CompletableFutures` we saw progress in tests `A` through `E`, but with `Streams`, nothing even begins until you apply a terminal operation, such as the `forEach()` at **[1]**. A `CompletableFuture` performs work and captures any exceptions for later retrieval. It's not quite straightforward to compare the two, because of the way a `Stream` doesn't do anything at all without a terminal operation—but a `Stream` definitely doesn't store its exceptions.

Checked Exceptions

Neither `CompletableFutures` nor parallel `Streams` support operations containing checked exceptions. Instead, you must handle the checked exception at the point you invoke the operation, which produces much less elegant code:

```
// concurrent/ThrowsChecked.java
import java.util.stream.*;
import java.util.concurrent.*;

public class ThrowsChecked {
  class Checked extends Exception {}
  static ThrowsChecked nochecked(ThrowsChecked tc) {
    return tc;
  }
  static ThrowsChecked
  withchecked(ThrowsChecked tc) throws Checked {
    return tc;
  }
  static void testStream() {
    Stream.of(new ThrowsChecked())
      .map(ThrowsChecked::nochecked)
      // .map(ThrowsChecked::withchecked);        // [1]
      .map(tc -> {
        try {
          return withchecked(tc);
        } catch(Checked e) {
          throw new RuntimeException(e);
        }
      });
  }
  static void testCompletableFuture() {
    CompletableFuture
      .completedFuture(new ThrowsChecked())
      .thenApply(ThrowsChecked::nochecked)
      // .thenApply(ThrowsChecked::withchecked);  // [2]
      .thenApply(tc -> {
        try {
          return withchecked(tc);
        } catch(Checked e) {
          throw new RuntimeException(e);
        }
      });
  }
}
```

The compiler complains at **[1]** and **[2]** if you try to use method references for `withchecked()` as you can with `nochecked()`. Instead, you must write out the

lambda expression (or write a wrapper method that doesn't throw the exception).

Deadlock

Because tasks can become blocked, it's possible for one task to get stuck waiting for another task, which in turn waits for another task, and so on, until the chain leads back to a task waiting on the first one. You get a continuous loop of tasks waiting on each other, and no one can move. This is called *deadlock*.[174]

If you try running a program and it deadlocks right away, you can immediately track down the bug. The real problem is when your program seems to be working fine but has the hidden potential to deadlock. Here, you might not get any indication that deadlocking is possible, so the flaw is latent in your program until it unexpectedly happens—typically to a customer (in a way almost certainly difficult to reproduce). Thus, preventing deadlock through careful program design is a critical part of developing concurrent systems.

The *Dining Philosophers* problem, invented by Edsger Dijkstra, is the classic demonstration of deadlock. The basic description specifies five philosophers (the example shown here allows any number). These philosophers spend part of their time thinking and part of their time eating. While they are thinking, they don't need any shared resources, but they eat using a limited number of utensils. In the original problem description, the utensils are forks, and two forks are required to get spaghetti from a bowl in the middle of the table. A more convincing version uses chopsticks; clearly, each philosopher requires two chopsticks to eat.

A difficulty is introduced: As philosophers, they have very little money, so they can only afford five chopsticks (more generally, the same number of chopsticks as philosophers). These are spaced around the table between them. When a philosopher wants to eat, that philosopher must pick up the chopstick to the left and the one to the right. If the philosopher on either side is using a desired chopstick, our philosopher must wait until the necessary chopsticks become available.

The `StickHolder` class manages a single `Chopstick` by keeping it in a `BlockingQueue` of size one. A `BlockingQueue` is a collection, designed to be safely used in

[174] You can also have *livelock* when two tasks are able to change their state so that they don't block, but they never make any useful progress.

concurrent programs, that blocks (waits) if you call `take()` and the queue is empty. Once a new element is placed in the queue, the block is released and that value is returned:

```
// concurrent/StickHolder.java
import java.util.concurrent.*;

public class StickHolder {
  private static class Chopstick {}
  private Chopstick stick = new Chopstick();
  private BlockingQueue<Chopstick> holder =
    new ArrayBlockingQueue<>(1);
  public StickHolder() { putDown(); }
  public void pickUp() {
    try {
      holder.take(); // Blocks if unavailable
    } catch(InterruptedException e) {
      throw new RuntimeException(e);
    }
  }
  public void putDown() {
    try {
      holder.put(stick);
    } catch(InterruptedException e) {
      throw new RuntimeException(e);
    }
  }
}
```

For simplicity, the `Chopstick` is never actually produced by the `StickHolder`, but kept `private` within the class. If you call `pickUp()` and the stick is unavailable, `pickUp()` blocks until the stick is returned by another `Philosopher` calling `putDown()`. Note that all thread safety in this class is achieved through the `BlockingQueue`.

Each `Philosopher` is a task that attempts to `pickUp()` the chopstick to both its right and left so it can eat, then releases those chopsticks with `putDown()`:

```
// concurrent/Philosopher.java

public class Philosopher implements Runnable {
  private final int seat;
  private final StickHolder left, right;
  public Philosopher(int seat,
    StickHolder left, StickHolder right) {
    this.seat = seat;
    this.left = left;
    this.right = right;
  }
  @Override public String toString() {
    return "P" + seat;
  }
  @Override public void run() {
    while(true) {
      // System.out.println("Thinking");   // [1]
      right.pickUp();
      left.pickUp();
      System.out.println(this + " eating");
      right.putDown();
      left.putDown();
    }
  }
}
```

No two `Philosophers` can successfully `take()` the same chopstick at the same time. In addition, if a chopstick has already been taken by one `Philosopher`, the next `Philosopher` trying to take that same chopstick will block, waiting for it to be released.

The result is a seemingly-innocent program that deadlocks. I've used arrays here instead of collections only because the resulting syntax is cleaner:

```
// concurrent/DiningPhilosophers.java
// Hidden deadlock
// {ExcludeFromGradle} Gradle has trouble
import java.util.*;
import java.util.concurrent.*;
import onjava.Nap;

public class DiningPhilosophers {
  private StickHolder[] sticks;
  private Philosopher[] philosophers;
  public DiningPhilosophers(int n) {
    sticks = new StickHolder[n];
    Arrays.setAll(sticks, i -> new StickHolder());
    philosophers = new Philosopher[n];
    Arrays.setAll(philosophers, i ->
      new Philosopher(i,
        sticks[i], sticks[(i + 1) % n]));    // [1]
    // Fix by reversing stick order for this one:
    // philosophers[1] =                     // [2]
    //   new Philosopher(0, sticks[0], sticks[1]);
    Arrays.stream(philosophers)
      .forEach(CompletableFuture::runAsync); // [3]
  }
  public static void main(String[] args) {
    // Returns right away:
    new DiningPhilosophers(5);              // [4]
    // Keeps main() from exiting:
    new Nap(3, "Shutdown");
  }
}
```

When you stop seeing output, the program is deadlocked. Depending on your machine configuration, however, you might not see deadlocking. It appears this depends on the number of cores[175] on your machine. Two cores don't seem to produce deadlocking, but more than two appear to readily produce deadlock. This behavior makes the example an even better demonstration of deadlock, because you might be writing your program on a machine that has two cores (if that is indeed what causes the issue) and become convinced that it is working correctly, only to have it start

[175]Not hyperthreads; there are often two hyperthreads per core and when asked for the number of cores, the version of Java used for this book would report the number of hyperthreads instead. Hyperthreads produce much faster context switching, but only actual cores do the work.

deadlocking when you install it on a different machine. And note that just because you can't easily see the deadlock doesn't mean the program can't deadlock on a two-core machine. The program is still deadlock-prone, it just happens rarely—arguably the worst situation because the problem doesn't present itself easily.

In the `DiningPhilosophers` constructor, each `Philosopher` is given a reference to a left and right `StickHolder`. Every `Philosopher` except the last one is initialized by situating that `Philosopher` between the next pair of chopsticks. The last `Philosopher` is given the zeroth chopstick for its right chopstick, so the round table is completed. That's because the last `Philosopher` is sitting right next to the first one, and they both share that zeroth chopstick. **[1]** shows the right-hand stick selected with a modulus of `n`, wrapping the last `Philosopher` around to be next to the first one.

Now all `Philosophers` can try to eat, each waiting on the `Philosopher` next to them to put down its chopstick.

To start each `Philosopher` running at **[3]**, I call `runAsync()` which means that the `DiningPhilosophers` constructor returns right away at **[4]**. Without anything to keep `main()` from completing, the program simply exits and doesn't do much. The `Nap` object blocks `main()` from exiting, then after three seconds forces an exit from the (presumably) deadlocked program.

In the configuration as given, the `Philosophers` spend virtually no time thinking. Thus they all compete for chopsticks while trying to eat, and deadlock tends to happen quickly. You can change this:

1. Add more `Philosophers` by increasing the value at **[4]**.
2. Uncomment line **[1]** in `Philosopher.java`.

Either one will make deadlock less likely, which shows the danger of writing a concurrent program and believing it's safe because it seems to "run OK on my machine." You can easily convince yourself the program is deadlock free, even though it isn't. This example is interesting precisely because it demonstrates that a program can appear to run correctly while still prone to deadlock.

To repair the problem, we observe that deadlock occurs when four conditions are simultaneously met:

1. Mutual exclusion. At least one resource used by the tasks must not be shareable. Here, a chopstick can be used by only one `Philosopher` at a time.

2. At least one task must hold a resource and wait to acquire a resource currently held by another task. That is, for deadlock to occur, a `Philosopher` must hold one chopstick and be waiting for a second one.
3. A resource cannot be preemptively taken away from a task. Tasks only release resources as a normal event. Our `Philosophers` are polite and they don't grab chopsticks from other `Philosophers`.
4. A circular wait can happen, whereby a task waits on a resource held by another task, which in turn is waiting on a resource held by another task, and so on, until one of the tasks is waiting on a resource held by the first task, thus gridlocking everything. In `DiningPhilosophers.java`, the circular wait happens because each `Philosopher` tries to get the right chopstick first, then the left.

Because *all* these conditions must be met to cause deadlock, you must only prevent one of them to prohibit deadlock. In this program, an easy way to prevent deadlock is to break the fourth condition. This condition happens because each `Philosopher` tries to pick up its chopsticks in a particular sequence: first right, then left. Because of that, it's possible for each `Philosopher` to hold its right chopstick while waiting for the left, causing the circular wait condition. However, if one of the `Philosophers` tries instead to get the left chopstick first, that `Philosopher` never prevents the `Philosopher` on the immediate right from picking up a chopstick, precluding the circular wait.

In `DiningPhilosophers.java`, uncomment the line at **[1]** and the one following it. This replaces the original `philosophers[1]` with a `Philosopher` that has its chopsticks reversed. By ensuring that the second `Philosopher` picks up and puts down the left chopstick before the right, we remove the potential for deadlock.

This is only one solution to the problem. You can also solve it by preventing one of the other conditions.

There is no language support to help prevent deadlock; it's up to you to avoid it by careful design. These are not comforting words to the person who's trying to debug a deadlocking program. And of course the easiest and best way to avoid concurrency problems is *never share resources*—unfortunately that's not always possible.

Constructors are not Thread-Safe

When you imagine the construction process, it can be easy to think that it's thread-safe. After all, no one can even see the new object before it finishes initialization, so how could there be contention over that object? Indeed, the Java Language specification[176] (JLS) confidently states:

"There is no practical need for a constructor to be synchronized, because it would lock the object under construction, which is normally not made available to other threads until all constructors for the object have completed their work."

Unfortunately, object construction is as vulnerable to shared-memory concurrency problems as anything else. The mechanisms can be more subtle, however.

Consider the automatic creation of a unique identifier for each object using a `static` field. To test different implementations, we'll start with an interface:

```
// concurrent/HasID.java

public interface HasID {
  int getID();
}
```

Then implement that interface in an obvious way:

```
// concurrent/StaticIDField.java

public class StaticIDField implements HasID {
  private static int counter = 0;
  private int id = counter++;
  @Override public int getID() { return id; }
}
```

This is about as simple and innocuous a class as you can imagine. It doesn't even have an explicit constructor to cause problems. To see what happens when we make multiple concurrent tasks that create these objects, here's a test harness:

[176] http://docs.oracle.com/javase/specs/jls/se8/html/jls-8.html#jls-8.8.3

```
// concurrent/IDChecker.java
import java.util.*;
import java.util.function.*;
import java.util.stream.*;
import java.util.concurrent.*;
import com.google.common.collect.Sets;

public class IDChecker {
  public static final int SIZE = 100_000;
  static class MakeObjects
  implements Supplier<List<Integer>> {
    private Supplier<HasID> gen;
    MakeObjects(Supplier<HasID> gen) {
      this.gen = gen;
    }
    @Override public List<Integer> get() {
      return
        Stream.generate(gen)
          .limit(SIZE)
          .map(HasID::getID)
          .collect(Collectors.toList());
    }
  }
  public static void test(Supplier<HasID> gen) {
    CompletableFuture<List<Integer>>
      groupA = CompletableFuture
        .supplyAsync(new MakeObjects(gen)),
      groupB = CompletableFuture
        .supplyAsync(new MakeObjects(gen));
    groupA.thenAcceptBoth(groupB, (a, b) -> {
      System.out.println(
        Sets.intersection(
          Sets.newHashSet(a),
          Sets.newHashSet(b)).size());
    }).join();
  }
}
```

The `MakeObjects` class is a `Supplier` with a `get()` that produces a `List<Integer>`. This `List` is generated by extracting the `id` from each `HasID` object. The `test()` method creates two parallel `CompletableFutures` that run `MakeObjects` suppliers, then takes the results of each and uses the Guava library `Sets.intersection()` to

find out how many `ids` are common between the two `List<Integer>` (Guava is much faster than using `retainAll()`).

Now we can test the `StaticIDField`:

```
// concurrent/TestStaticIDField.java

public class TestStaticIDField {
  public static void main(String[] args) {
    IDChecker.test(StaticIDField::new);
  }
}
/* Output:
21397
*/
```

That's a rather large number of duplicates. Clearly, a plain `static int` is not safe to use for construction. Let's make it thread-safe using an `AtomicInteger`:

```
// concurrent/GuardedIDField.java
import java.util.concurrent.atomic.*;

public class GuardedIDField implements HasID {
  private static AtomicInteger counter =
    new AtomicInteger();
  private int id = counter.getAndIncrement();
  @Override public int getID() { return id; }
  public static void main(String[] args) {
    IDChecker.test(GuardedIDField::new);
  }
}
/* Output:
0
*/
```

Constructors have an even more subtle way to share state: through constructor arguments:

```
// concurrent/SharedConstructorArgument.java
import java.util.concurrent.atomic.*;

interface SharedArg {
  int get();
}

class Unsafe implements SharedArg {
  private int i = 0;
  @Override public int get() { return i++; }
}

class Safe implements SharedArg {
  private static AtomicInteger counter =
    new AtomicInteger();
  @Override public int get() {
    return counter.getAndIncrement();
  }
}

class SharedUser implements HasID {
  private final int id;
  SharedUser(SharedArg sa) {
    id = sa.get();
  }
  @Override public int getID() { return id; }
}

public class SharedConstructorArgument {
  public static void main(String[] args) {
    Unsafe unsafe = new Unsafe();
    IDChecker.test(() -> new SharedUser(unsafe));
    Safe safe = new Safe();
    IDChecker.test(() -> new SharedUser(safe));
  }
}
/* Output:
16537
0
*/
```

Here, the `SharedUser` constructors share the same argument. Even though `Share-`

`dUser` is using its argument in a completely innocent and reasonable fashion, *the way the constructor is called* causes collisions. `SharedUser` cannot even know it is being used this way, much less control it!

`synchronized` constructors are not supported by the language, but it's possible to create your own using a `synchronized` block (see the Appendix: Low-Level Concurrency to learn about the `synchronized` keyword). Although the JLS states that "*... it would lock the object under construction*", this is not true—the constructor is effectively a `static` method, so a `synchronized` constructor would actually lock the *class* object. We can reproduce this by creating our own `static` object and locking on that:

```
// concurrent/SynchronizedConstructor.java
import java.util.concurrent.atomic.*;

class SyncConstructor implements HasID {
  private final int id;
  private static Object
    constructorLock = new Object();
  SyncConstructor(SharedArg sa) {
    synchronized(constructorLock) {
      id = sa.get();
    }
  }
  @Override public int getID() { return id; }
}

public class SynchronizedConstructor {
  public static void main(String[] args) {
    Unsafe unsafe = new Unsafe();
    IDChecker.test(() ->
      new SyncConstructor(unsafe));
  }
}
/* Output:
0
*/
```

The shared use of the `Unsafe` class is now safe.

An alternate approach is to make the constructors `private` (thus preventing inheritance) and provide a `static` *Factory Method* to produce new objects:

```
// concurrent/SynchronizedFactory.java
import java.util.concurrent.atomic.*;

final class SyncFactory implements HasID {
  private final int id;
  private SyncFactory(SharedArg sa) {
    id = sa.get();
  }
  @Override public int getID() { return id; }
  public static synchronized
  SyncFactory factory(SharedArg sa) {
    return new SyncFactory(sa);
  }
}

public class SynchronizedFactory {
  public static void main(String[] args) {
    Unsafe unsafe = new Unsafe();
    IDChecker.test(() ->
      SyncFactory.factory(unsafe));
  }
}
/* Output:
0
*/
```

By synchronizing the `static` *Factory Method* you lock on the class object during construction.

These examples emphasize how insidiously difficult it is to detect and manage shared state in concurrent Java programs. Even if you take the "share nothing" strategy, it's remarkably easy for accidental sharing to take place.

Effort, Complexity, Cost

Suppose you are making a pizza. The amount of work required to get from the current step in the process to the next one is represented here as part of an enumeration:

```
// concurrent/Pizza.java
import java.util.function.*;
import onjava.Nap;

public class Pizza {
  public enum Step {
    DOUGH(4), ROLLED(1), SAUCED(1), CHEESED(2),
    TOPPED(5), BAKED(2), SLICED(1), BOXED(0);
    int effort; // Needed to get to the next step
    Step(int effort) { this.effort = effort; }
    Step forward() {
      if(equals(BOXED)) return BOXED;
      new Nap(effort * 0.1);
      return values()[ordinal() + 1];
    }
  }
  private Step step = Step.DOUGH;
  private final int id;
  public Pizza(int id) { this.id = id; }
  public Pizza next() {
    step = step.forward();
    System.out.println("Pizza " + id + ": " + step);
    return this;
  }
  public Pizza next(Step previousStep) {
    if(!step.equals(previousStep))
      throw new IllegalStateException("Expected " +
        previousStep + " but found " + step);
    return next();
  }
  public Pizza roll() { return next(Step.DOUGH); }
  public Pizza sauce() { return next(Step.ROLLED); }
  public Pizza cheese() { return next(Step.SAUCED); }
  public Pizza toppings() { return next(Step.CHEESED); }
  public Pizza bake() { return next(Step.TOPPED); }
  public Pizza slice() { return next(Step.BAKED); }
  public Pizza box() { return next(Step.SLICED); }
  public boolean complete() {
    return step.equals(Step.BOXED);
  }
  @Override public String toString() {
    return "Pizza" + id + ": " +
```

```
      (step.equals(Step.BOXED)? "complete" : step);
  }
}
```

This is another trivial state machine, like `Machina.java`. The endpoint is reached when the pizza is in a box.

If one person is making one pizza, all the steps happen linearly, one after the other:

```
// concurrent/OnePizza.java
import onjava.Timer;

public class OnePizza {
  public static void main(String[] args) {
    Pizza za = new Pizza(0);
    System.out.println(
      Timer.duration(() -> {
        while(!za.complete())
          za.next();
      }));
  }
}
/* Output:
Pizza 0: ROLLED
Pizza 0: SAUCED
Pizza 0: CHEESED
Pizza 0: TOPPED
Pizza 0: BAKED
Pizza 0: SLICED
Pizza 0: BOXED
1665
*/
```

The time is in milliseconds and agrees with what we'd expect by adding up the effort for all the steps.

If you made five pizzas this way, you'd expect it to take five times as long. But what if that isn't fast enough for you? We can start by trying the parallel stream approach:

```
// concurrent/PizzaStreams.java
import java.util.*;
import java.util.stream.*;
import onjava.Timer;

public class PizzaStreams {
  static final int QUANTITY = 5;
  public static void main(String[] args) {
    Timer timer = new Timer();
    IntStream.range(0, QUANTITY)
      .mapToObj(Pizza::new)
      .parallel()                    // [1]
      .forEach(za -> {
        while(!za.complete())
          za.next();
      });
    System.out.println(timer.duration());
  }
}
/* Output:
Pizza 0: ROLLED
Pizza 1: ROLLED
Pizza 2: ROLLED
Pizza 3: ROLLED
Pizza 4: ROLLED
Pizza 1: SAUCED
Pizza 0: SAUCED
Pizza 4: SAUCED
Pizza 2: SAUCED
Pizza 3: SAUCED
Pizza 3: CHEESED
Pizza 2: CHEESED
Pizza 4: CHEESED
Pizza 0: CHEESED
Pizza 1: CHEESED
Pizza 3: TOPPED
Pizza 2: TOPPED
Pizza 4: TOPPED
Pizza 1: TOPPED
Pizza 0: TOPPED
Pizza 4: BAKED
Pizza 2: BAKED
```

```
Pizza 3: BAKED
Pizza 0: BAKED
Pizza 1: BAKED
Pizza 4: SLICED
Pizza 3: SLICED
Pizza 2: SLICED
Pizza 0: SLICED
Pizza 1: SLICED
Pizza 2: BOXED
Pizza 4: BOXED
Pizza 3: BOXED
Pizza 1: BOXED
Pizza 0: BOXED
1797
*/
```

Now we've created five pizzas in about the same amount of time as it took to create a single pizza. Try removing the line marked **[1]** to verify that it takes five times longer otherwise. Also try changing `QUANTITY` to 4, 8, 10, 16, and 17 to see the difference, and guess why it happens that way.

`PizzaStreams.java` does all the work inside its `forEach()`. Would it make any difference if we mapped the individual steps?

```
// concurrent/PizzaParallelSteps.java
import java.util.*;
import java.util.stream.*;
import onjava.Timer;

public class PizzaParallelSteps {
  static final int QUANTITY = 5;
  public static void main(String[] args) {
    Timer timer = new Timer();
    IntStream.range(0, QUANTITY)
      .mapToObj(Pizza::new)
      .parallel()
      .map(Pizza::roll)
      .map(Pizza::sauce)
      .map(Pizza::cheese)
      .map(Pizza::toppings)
      .map(Pizza::bake)
      .map(Pizza::slice)
```

```
      .map(Pizza::box)
      .forEach(za -> System.out.println(za));
    System.out.println(timer.duration());
  }
}
/* Output:
Pizza 1: ROLLED
Pizza 2: ROLLED
Pizza 0: ROLLED
Pizza 3: ROLLED
Pizza 4: ROLLED
Pizza 4: SAUCED
Pizza 2: SAUCED
Pizza 0: SAUCED
Pizza 1: SAUCED
Pizza 3: SAUCED
Pizza 1: CHEESED
Pizza 3: CHEESED
Pizza 4: CHEESED
Pizza 2: CHEESED
Pizza 0: CHEESED
Pizza 3: TOPPED
Pizza 1: TOPPED
Pizza 0: TOPPED
Pizza 2: TOPPED
Pizza 4: TOPPED
Pizza 2: BAKED
Pizza 3: BAKED
Pizza 1: BAKED
Pizza 0: BAKED
Pizza 4: BAKED
Pizza 3: SLICED
Pizza 1: SLICED
Pizza 2: SLICED
Pizza 0: SLICED
Pizza 4: SLICED
Pizza 3: BOXED
Pizza3: complete
Pizza 1: BOXED
Pizza1: complete
Pizza 2: BOXED
Pizza 0: BOXED
```

```
Pizza0: complete
Pizza2: complete
Pizza 4: BOXED
Pizza4: complete
1766
*/
```

The answer is "no," and in hindsight this is not surprising because each pizza requires the steps to be executed in order, so there's no opportunity for a further speedup by doing things in discrete steps as in `PizzaParallelSteps.java`.

We can rewrite the example using `CompletableFutures`:

```
// concurrent/CompletablePizza.java
import java.util.*;
import java.util.concurrent.*;
import java.util.stream.*;
import onjava.Timer;

public class CompletablePizza {
  static final int QUANTITY = 5;
  public static CompletableFuture<Pizza>
  makeCF(Pizza za) {
    return CompletableFuture
      .completedFuture(za)
      .thenApplyAsync(Pizza::roll)
      .thenApplyAsync(Pizza::sauce)
      .thenApplyAsync(Pizza::cheese)
      .thenApplyAsync(Pizza::toppings)
      .thenApplyAsync(Pizza::bake)
      .thenApplyAsync(Pizza::slice)
      .thenApplyAsync(Pizza::box);
  }
  public static void
  show(CompletableFuture<Pizza> cf) {
    try {
      System.out.println(cf.get());
    } catch(Exception e) {
      throw new RuntimeException(e);
    }
  }
  public static void main(String[] args) {
```

```
    Timer timer = new Timer();
    List<CompletableFuture<Pizza>> pizzas =
      IntStream.range(0, QUANTITY)
        .mapToObj(Pizza::new)
        .map(CompletablePizza::makeCF)
        .collect(Collectors.toList());
    System.out.println(timer.duration());
    pizzas.forEach(CompletablePizza::show);
    System.out.println(timer.duration());
  }
}
/* Output:
98
Pizza 1: ROLLED
Pizza 3: ROLLED
Pizza 4: ROLLED
Pizza 2: ROLLED
Pizza 0: ROLLED
Pizza 1: SAUCED
Pizza 2: SAUCED
Pizza 3: SAUCED
Pizza 0: SAUCED
Pizza 4: SAUCED
Pizza 2: CHEESED
Pizza 0: CHEESED
Pizza 3: CHEESED
Pizza 1: CHEESED
Pizza 4: CHEESED
Pizza 2: TOPPED
Pizza 4: TOPPED
Pizza 1: TOPPED
Pizza 3: TOPPED
Pizza 0: TOPPED
Pizza 0: BAKED
Pizza 4: BAKED
Pizza 1: BAKED
Pizza 3: BAKED
Pizza 2: BAKED
Pizza 3: SLICED
Pizza 1: SLICED
Pizza 2: SLICED
Pizza 4: SLICED
```

```
Pizza 0: SLICED
Pizza 0: BOXED
Pizza 2: BOXED
Pizza 4: BOXED
Pizza 1: BOXED
Pizza0: complete
Pizza1: complete
Pizza2: complete
Pizza 3: BOXED
Pizza3: complete
Pizza4: complete
1762
*/
```

Parallel streams and `CompletableFutures` are the most well-developed techniques in the Java concurrency toolbox. You should always choose one of these first. The parallel stream approach is most appropriate when a problem is *embarrassingly parallel*, that is, when it is trivially easy to break your data into identical, easy-to-process pieces (when doing this yourself you must roll up your sleeves and delve into the `Spliterator` documentation). `CompletableFutures` work best when the pieces of *work* are distinct. `CompletableFuture` seems more task-oriented than data-oriented.

With the pizza problem, the results don't seem that different—in fact, the parallel stream approach looks cleaner, and for that reason alone I find parallel streams more attractive as a first attempt.

It takes a certain amount of time to make a pizza. No matter what concurrency approach you use, the best you can do is create `n` pizzas in the same amount of time it takes to create one pizza. It's easy to see that here, but when you're working on a more sophisticated problem you can miss it. Often, a back-of-the-envelope calculation at the beginning of a project quickly shows the maximum possible throughput, which prevents you from spinning your wheels trying to make it faster.

If you really do need to use concurrency, parallel `Streams` and `CompletableFutures` might easily produce significant benefits, but be careful when trying to push it further. The cost and effort can easily become far greater than any advantages you might wring out.

Summary

The only justification for concurrency is "too much waiting." This can also include the responsiveness of user interfaces, but as Java is effectively not used to build user interfaces,[177] this simply means "your program isn't running fast enough."

If concurrency were easy, there would be no reason to avoid it. Because it is hard, you should consider carefully whether it's worth the effort. Can you speed things up some other way? For example, move to faster hardware (which can be a lot less expensive than lost programmer time) or break your program into pieces and run those pieces on different machines?

Occam's (or Ockham's) razor is an oft-misunderstood principle. I've seen at least one movie where they define it as "the simplest solution is the correct one," as if it's some kind of law. It's actually a guideline: When faced with a number of approaches, first try the one that requires the fewest assumptions. In the programming world, this has evolved into "try the simplest thing that could possibly work." When you know something about a particular tool—as you now know something about concurrency—it can be quite tempting to use it, or to specify ahead of time that your solution must "run fast," to justify designing in concurrency from the beginning. But our programming version of Occam's razor says that you should try the simplest approach first (which will also be cheaper to develop) and see if it's good enough.

As I came from a low-level background (physics and computer engineering), I was prone to imagining the cost of all the little wheels turning. I can't count the number of times I was certain the simplest approach could never be fast enough, only to discover upon trying that it was more than adequate.

Drawbacks

The main drawbacks to concurrency are:

1. Slowdown while threads wait for shared resources.
2. Additional CPU overhead for thread management.
3. Unrewarded complexity from poor design decisions.

[177] The libraries are there and the language was intended to be used for this purpose but in practice it happens so rarely as to be able to say "never."

4. Pathologies such as starving, racing, deadlock, and livelock (multiple threads working individual tasks that the ensemble can't finish).
5. Inconsistencies across platforms. With some examples, I discovered race conditions that quickly appeared on some computers but not on others. If you develop a program on the latter, you might get badly surprised when you distribute it.

In addition, there's an art to the application of concurrency. Java is designed to allow you to create as many objects as necessary to solve your problem—at least in theory.[178] However, `Threads` are not typical objects: each has its own execution environment including a stack and other necessary elements, making it much larger than a normal object. In most environments it's only possible to create a few thousand `Thread` objects before running out of memory. You normally only need a handful of threads to solve a problem, so this is typically not much of a limit, but for some designs it becomes a constraint that might force you to use an entirely different scheme.

The Shared-Memory Pitfall

One of the main difficulties with concurrency occurs because more than one task might be sharing a resource—such as the memory in an object—and you must ensure that multiple tasks don't simultaneously read and change that resource.

I have spent years studying and struggling with concurrency. I've learned you can never believe that a program using shared-memory concurrency is working correctly. You can discover it's wrong, but you can never prove it's right. This is one of the well-know maxims of concurrency.[179]

I've met numerous people who have an impressive amount of confidence in their ability to write correct threaded programs. I occasionally start thinking I can get it right, too. For one particular program, I initially wrote it when we only had single-CPU machines. I was able to convince myself that, because of the promises I thought I understood about Java tools, the program was correct. And it didn't fail on my single-CPU machine.

[178] Creating millions of objects for finite-element analysis in engineering, for example, might not be practical in Java without the *Flyweight* design pattern.

[179] In science, a theory is never proved, but to be valid it must be *falsifiable*. With concurrency, we can't even get falsifiability most of the time.

Fast forward to machines with multiple CPUs. I was surprised when the program broke, but that's one of the problems. It's not Java's fault; "write once, run everywhere" cannot possibly extend to concurrency on single vs. multicore machines. It's a fundamental problem with concurrency. You *can* actually discover some concurrency problems on a single-CPU machine, but there are other problems that won't appear until you try it on a multi-CPU machine, where your threads are actually running in parallel.

As another example, the dining philosophers problem can easily be adjusted so deadlock rarely happens, giving you the impression that everything is copacetic.

You can never let yourself become too confident about your programming abilities when it comes to shared-memory concurrency.

This Albatross is Big

If you feel overwhelmed about Java concurrency, it turns out you're in good company. Go to the Javadoc page for the `Thread` class[180]. Now look and see how many of the methods are **Deprecated**. These are things that the Java language designers *got wrong*, because they didn't understand enough about concurrency when they were designing the language.

A number of library solutions added in subsequent versions of Java have turned out to be ineffective or even useless. Fortunately, both parallel `Streams` and `CompletableFutures` in Java 8 are quite valuable, but you will still encounter the old solutions when you work with legacy code.

Elsewhere in this book I've talked about one of the essential problems in Java: every failed experiment is forever embedded in the language or library. Java concurrency emphasizes this issue. It's not so much that there are lots of mistakes—although there are those—as there are lots of different attempts to solve the problem. The upside is that these attempts have yielded better and simpler designs. The downside is that you can easily get lost in the older designs before finding your way to the good stuff.

Other Libraries

This chapter focused on the relatively safe and easy tools of parallel streams and `CompletableFutures` and only touched on some of the more fine-grained tools in

[180] https://docs.oracle.com/javase/8/docs/api/java/lang/Thread.html

the Java standard library. To keep from overwhelming you, I didn't cover some libraries you might actually use in practice. We used a couple of the `Atomic` classes, `ConcurrentLinkedDeque`, `ExecutorService` and `ArrayBlockingQueue`. The Appendix: Low-Level Concurrency covers a few others, but you'll also want to explore the `java.util.concurrent` Javadocs. Be wary, however, as some of the library components have been superseded with new and better ones.

Consider a Language Designed for Concurrency

In general, use concurrency carefully and sparingly. If you need to use it, try as hard as possible to use the most modern approaches: parallel streams or `CompletableFutures`. These are designed to—as much as possible given Java's world—keep you out of trouble, assuming you don't attempt to share memory.

If your concurrency issues get larger and more complex than what high-level Java constructs can support, consider using a language designed for concurrency. It might be possible to use such a language only for the portions of your program that demand concurrency. At this writing, the purest functional languages on the JVM are Clojure (a version of Lisp) and Frege (an implementation of Haskell). These allow you to write the concurrent parts of your application in that language and easily interact with your main Java code via the JVM. Alternatively, you might choose to communicate off-JVM via a *foreign function interface* (FFI) to a different language that is designed for concurrency; at this writing *gRPC* is the most promising approach.

Its easy to become attached to a language and contort yourself trying to do everything using that language. A common example is building HTML/JavaScript user interfaces; those tools are indeed ugly and unpleasant to use, and there are numerous libraries that allow you to generate those by writing code in your favorite language (for example, `Scala.js` allows you to do it in Scala).

Mental convenience is a valid consideration. However, I hope I've shown in this chapter (and the Appendix: Low-Level Concurrency) that Java concurrency is a deep hole from which you might not escape. The knowledge required to visually inspect code while remembering all the pitfalls is more difficult than for any other part of the Java language.

Regardless of how simple concurrency can seem using a particular language or library, consider it a black art. There's always something that can bite you when

you least expect it.

Further Reading

Java Concurrency in Practice, by Brian Goetz, Tim Peierls, Joshua Bloch, Joseph Bowbeer, David Holmes, and Doug Lea (Addison-Wesley, 2006)—basically, the "who's who" in the Java concurrency world.

Concurrent Programming in Java, Second Edition, by Doug Lea (Addison-Wesley, 2000). Although this book significantly predates Java 5, much of Doug's work became the `java.util.concurrent` libraries, so this book is essential for a complete understanding of concurrency issues. It goes beyond Java and discusses concurrency across languages and technologies. Although it can be obtuse in places, it merits rereading several times (preferably with months in between to internalize the information). Doug is one of the few people in the world who actually understands concurrency, so this is a worthwhile endeavor.

Patterns

> The object-oriented *design patterns* movement is chronicled in the book *Design Patterns*, by Gamma, Helm, Johnson & Vlissides (Addison-Wesley 1995).[181]

That book shows 23 different solutions to particular classes of problems. In this chapter, the basic concepts of design patterns are introduced through examples. This should whet your appetite to read *Design Patterns*, a source of what has become an important vocabulary for OOP programmers.

The latter part of this chapter contains an example of the design evolution process, starting with an initial solution of a trash sorting simulation and moving through the logic and process of evolving that solution to more appropriate designs. You can look at that evolution as a prototype for the way a design can start as an adequate solution to a particular problem and grow into a flexible approach to a class of problems.

The Pattern Concept

Initially, you can think of a pattern as an especially clever and insightful way to solve a particular class of problems. A pattern looks like many people have worked out all the angles of a problem and have come up with the most general and flexible solution. The problem could be one you have seen and solved before, but your solution probably didn't have the kind of completeness you'll see embodied in a pattern.

Although they're called "design patterns," they really aren't tied to the realm of design. A pattern seems to stand apart from the traditional way of thinking about analysis, design, and implementation. Instead, a pattern embodies a complete idea within a program, and thus it can sometimes appear at the analysis phase or high-level design phase. Because a pattern has a direct implementation in code, you

[181] But be warned: the examples are in C++.

might not expect it to show up before low-level design or implementation—or even maintenance. Often you don't realize you need a particular pattern until you get to one of those phases.

The basic concept of a pattern can also be seen as the basic concept of program design: adding a layer of abstraction. Whenever you abstract something, you're isolating particular details, and one of the most compelling motivations behind this is to:

> *Separate things that change from things that stay the same.*

Another way to put this is that once you find some part of your program that's likely to change for one reason or another, an abstraction can keep those changes from causing other changes throughout your code.

Often, the most difficult part of developing an elegant and cheap-to-maintain design is in discovering what I call *the vector of change* ("vector" refers to the maximum gradient and not a collection class). This means finding the most important thing that changes in your system. To put it another way, finding your greatest cost. Once you discover the vector of change, you have the focal point around which to structure your design.

So the goal of design patterns is to isolate changes in your code. If you look at it this way, you've already seen design patterns in this book. For example, inheritance can be thought of as a design pattern (albeit one implemented by the compiler). It allows you to express differences in behavior (the thing that changes) in objects that all have the same interface (the thing that stays the same). Composition can also be considered a pattern because it allows you to change—dynamically or statically—the objects that implement your class, and thus the way that class behaves.

You've also seen another pattern that appears in *Design Patterns*: the *Iterator* (Java 1.0 and 1.1 capriciously called it the `Enumeration`; Java 2 collections use `Iterator`). This hides the particular implementation of the collection as you're stepping through and selecting the elements one by one. The iterator allows you to write code that performs an operation on all elements in a sequence without regard to the way that sequence is built. Thus your code can be used with any collection that produces an iterator.

Although patterns can be very helpful, some people assert that:

Design patterns represent language failures.

This is an important insight. Just because a pattern makes sense in C++, for example, it might not be necessary in Java or in another language. For that reason, just because a pattern appears in the *Design Patterns* book, it doesn't mean it is helpful when applied to your language.

I find the "language failure" observation useful, but I also think it's an oversimplification. If you're trying to solve a particular problem and the language doesn't have direct support for the technique you're using, you could argue that it's a failure of the language. But how often do you actually use that particular technique? Perhaps the balance is just right: while you must work harder when you use the technique, maybe you don't need it enough to justify including support in the language. On the other hand, without language support, it might be too messy to use the technique on a regular basis, but with language support you might change the way you program (Java 8 streams achieve this, for example).

Singleton

Possibly the simplest design pattern is the *Singleton*, which is a way to provide one and only one instance of an object. Consider a parameterized `Resource`:

```
// patterns/Resource.java

public interface Resource<T> {
  T get();
  void set(T x);
}
```

Suppose we want to allow only a single instance of each different `Resource<T>`. One approach is to create a customized singleton `Resource` class for each `T`:

```
// patterns/SingletonPattern.java

final class IntegerSingleton
  implements Resource<Integer> {
  private static IntegerSingleton value =
    new IntegerSingleton();
  private Integer i = Integer.valueOf(0);
  private IntegerSingleton() {
    System.out.println("IntegerSingleton()");
  }
  public static IntegerSingleton instance() {
    return value;
  }
  @Override public synchronized
  Integer get() { return i; }
  @Override public synchronized
  void set(Integer x) { i = x; }
}

public class SingletonPattern {
  public static <T> void show(Resource<T> r) {
    T val = r.get();
    System.out.println(val);
  }
  public static <T> void put(Resource<T> r, T val) {
    r.set(val);
  }
  public static void main(String[] args) {
    System.out.println("Inside main()");
    Resource<Integer> ir =
      IntegerSingleton.instance();
    Resource<Integer> ir2 =
      IntegerSingleton.instance();
    show(ir);
    put(ir2, Integer.valueOf(9));
    show(ir);
  }
}
/* Output:
Inside main()
IntegerSingleton()
0
```

```
9
*/
```

To ensure that only one object of the controlled type is created, the `IntegerSingleton` constructor is `private` so it's only usable within the class.

Because it is `static`, the object `value` is created the first time the client programmer calls the `static` method `instance()`. At that point the class loads and the `static` initializer for `value` is executed. The creation of `ir2` in `main()` does not cause a second constructor call.

Because of the way the JVM works, `static` initialization is thread-safe. To complete the thread-safety, the getters and setters for `IntegerSingleton` are `synchronized`. This is important because multiple threads can hold references to the same shared `IntegerSingleton` object. As described in Concurrent Programming, you must think about concurrency even though you yourself are not using the class in a concurrent program.

Java also allows the creation of objects through cloning (see the Appendix: Passing and Returning Objects). In this example, making the class `final` prevents cloning. `IntegerSingleton` has no explicit base class and is thus inherited directly from `Object`. The `clone()` method remains `protected` and it cannot be called (doing so produces a compile-time error). However, if you're inheriting from a class hierarchy that has overridden `clone()` as `public` and implemented `Cloneable`, the way to prevent cloning is to override `clone()` and throw a `CloneNotSupportedException` as described in the Appendix: Passing and Returning Objects. (You can also override `clone()` and simply return `this`, but that's deceiving because the client programmer thinks they are cloning the object, but instead are still dealing with the original.)

The `show()` and `put()` functions demonstrate that you can upcast to `Resource<T>` and use polymorphism.

You can also use this approach to create a limited pool of objects, although now you must manage the objects in the pool. If this is an issue, your solution can check the shared objects out and in.

Inheriting from a base class like `Resource<T>` might seem like unnecessary extra work. Shouldn't we be able to just create a pure generic class that implements the singleton concept? Here's one way to do it:

```
// patterns/Single.java

@SuppressWarnings("unchecked")
public final class Single<T> {
  private static Object single;       // [1]
  public Single(T val) {
    if(single != null)
      throw new RuntimeException(
        "Attempt to reassign Single<" +
        val.getClass().getSimpleName() + ">"
      );
    single = val;
  }
  public T get() { return (T)single; }
}
```

What we'd actually like to say in **[1]** is:

```
private static T single;
```

This produces the compile-time error: "non-static type variable T cannot be referenced from a static context." Basically, statics and generics do not play well together in Java, so we need to make `single` an `Object` and cast it when calling `get()`. We also cannot make `get()` a `static` method (if you try it you will learn a lot about `static` generic methods).

The first time you create a `Single` object, the constructor sets the `static single` value to the constructor argument. A second constructor call for the same `T` produces a runtime error message. Here's a basic test:

```
// patterns/TestSingle.java

public class TestSingle {
  public static void main(String[] args) {
    Single<String> ss = new Single<>("hello");
    System.out.println(ss.get());
    try {
      Single<String> ss2 = new Single<>("world");
    } catch(Exception e) {
      System.out.println(e.getMessage());
    }
```

```
  }
}
/* Output:
hello
Attempt to reassign Single<String>
*/
```

Using `Single` to create a `Double` singleton works reasonably well:

```
// patterns/SingletonPattern2.java

public class SingletonPattern2 {
  public static void main(String[] args) {
    Single<Double> pi =
      new Single<>(Double.valueOf(3.14159));
    Double x = pi.get();
    System.out.println(x);
  }
}
/* Output:
3.14159
*/
```

You cannot modify the contents of a `Double`. But what if the object in question is modifiable?

```
// patterns/SingletonPattern3.java

class MyString {
  private String s;
  public MyString(String s) {
    this.s = s;
  }
  public synchronized
  void change(String s) {
    this.s = s;
  }
  @Override public synchronized
  String toString() {
    return s;
  }
}
```

```
public class SingletonPattern3 {
  public static void main(String[] args) {
    Single<MyString> x =
      new Single<>(new MyString("Hello"));
    System.out.println(x.get());
    x.get().change("World!");
    System.out.println(x.get());
  }
}
/* Output:
Hello
World!
*/
```

When `Single` manages a modifiable object, you can produce a race condition in a concurrent program. In `MyString`, both `change()` and `toString()` are `synchronized` to prevent this.

Classifying Patterns

The *Design Patterns* book discusses 23 different patterns, classified under three purposes which revolve around the particular aspect that can vary. The three purposes are:

1. **Creational**: How an object is created. This often involves isolating the details of object creation so your code isn't dependent on what types of objects there are and thus doesn't change when you add a new type of object. The *Singleton* is classified as a creational pattern, and later in this chapter you'll see examples of *Factories*.
2. **Structural**: Designing objects to satisfy particular project constraints. These work with the way objects are connected with other objects to ensure that changes in the system don't require changes to those connections.
3. **Behavioral**: Objects that handle particular types of actions within a program. These encapsulate processes to perform, such as interpreting a language, fulfilling a request, moving through a sequence (as with an iterator), or implementing an algorithm. This chapter contains examples of the *Observer* and *Visitor* patterns.

The *Design Patterns* book has a section on each of its 23 patterns along with one or more examples for each, typically in C++ but sometimes in SmallTalk. This chapter does not repeat all the patterns shown in *Design Patterns* because that book stands on its own and should be studied separately (there are also a number of good online resources, many using Java). Instead, you'll see some examples that should provide you with a decent feel for what patterns are about and why they are important.

After years of looking at patterns, it began to occur to me that the patterns themselves use basic principles of organization, other than (and more fundamental than) those described in *Design Patterns*. These principles are based on the structure of the implementations, where I have seen significant similarities between patterns (more than those expressed in *Design Patterns*). Although we generally try to avoid implementation in favor of interface, I find it's often easier to think about, and especially to learn about, patterns in terms of these structural principles. This chapter attempts to present the patterns based on their structure instead of the categories in *Design Patterns*.

Template Method

An application framework helps you create a new application by reusing most of the code from the provided framework classes and overriding methods to customize the application to your needs.

A fundamental concept in an application framework is the *Template Method*, which is typically hidden and drives the application by calling various base-class methods which you override to create the application.

The Template Method is defined in the base class and cannot be changed. It's sometimes a `private` method but it's virtually always `final`. It calls other base-class methods (the ones you override) to do its job, but it is often called only as part of an initialization process; the client programmer isn't necessarily able to call it directly.

```
// patterns/TemplateMethod.java
// Basic Template Method pattern.
import java.util.stream.*;

abstract class ApplicationFramework {
  ApplicationFramework() {
    templateMethod();
  }
  abstract void customize1(int n);
  abstract void customize2(int n);
  // "private" means automatically "final":
  private void templateMethod() {
    IntStream.range(0, 5).forEach(
      n -> { customize1(n); customize2(n); });
  }
}

// Create a new application:
class MyApp extends ApplicationFramework {
  @Override void customize1(int n) {
    System.out.print("customize1 " + n);
  }
  @Override void customize2(int n) {
    System.out.println(" customize2 " + n);
  }
}

public class TemplateMethod {
  public static void main(String[] args) {
    new MyApp();
  }
}
/* Output:
customize1 0 customize2 0
customize1 1 customize2 1
customize1 2 customize2 2
customize1 3 customize2 3
customize1 4 customize2 4
*/
```

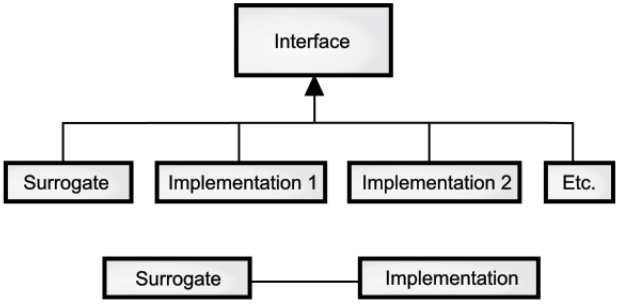
The base-class constructor is responsible for performing the necessary initialization, then starting the "engine" (the template method) that runs the application (in a GUI

application, this "engine" is the main event loop). The client programmer simply provides definitions for `customize1()` and `customize2()` and the "application" is ready to run.

Fronting for an Implementation

A common structure in design patterns combines a surrogate class that "fronts" for an implementing class which does the actual work. When you call a method in the surrogate, it turns around and calls a method in the implementing class. We'll look at three patterns that use this structure: *Proxy*, *State* and *State Machine*.

From a base class, the surrogate is derived along with the class or classes that provide the implementations:



The surrogate object is connected to an implementation, where it forwards all its method calls.

Structurally, a *Proxy* has only one implementation, while *State* has more than one. The application of the patterns is considered (in *Design Patterns*) to be distinct: *Proxy* is used to control access to its implementation, while *State* allows you to change the implementation dynamically. However, if you expand your notion of "controlling access to implementation" then the two fit neatly together.

Proxy

Implementing *Proxy* using the above diagram looks like this:

```
// patterns/ProxyDemo.java
// Basic demonstration of the Proxy pattern.

interface ProxyBase {
  void f();
  void g();
  void h();
}

class Proxy implements ProxyBase {
  private ProxyBase implementation =
    new Implementation();
  // Pass method calls to the implementation:
  @Override
  public void f() { implementation.f(); }
  @Override
  public void g() { implementation.g(); }
  @Override
  public void h() { implementation.h(); }
}

class Implementation implements ProxyBase {
  @Override public void f() {
    System.out.println("Implementation.f()");
  }
  @Override public void g() {
    System.out.println("Implementation.g()");
  }
  @Override public void h() {
    System.out.println("Implementation.h()");
  }
}

public class ProxyDemo {
  public static void main(String[] args) {
    Proxy p = new Proxy();
    p.f();
    p.g();
    p.h();
  }
}
/* Output:
```

```
Implementation.f()
Implementation.g()
Implementation.h()
*/
```

`Implementation` doesn't *need* the same interface as `Proxy`. As long as the `Proxy` somehow "speaks for" the implementation class, the basic idea is satisfied. However, it is convenient to have a common interface so `Implementation` is forced to fulfill all the methods that `Proxy` must call.

State

The *State* pattern adds more implementations to *Proxy*, along with a way to change implementations during the lifetime of the surrogate. Here's one approach to implementing the *State* pattern:

```
// patterns/StateDemo.java
// Basic demonstration of the State pattern.

class State {
  private State implementation;
  protected State() {}
  public State(State imp) {
    implementation = imp;
  }
  public void change(State newImp) {
    implementation = newImp;
  }
  // Forward method calls to the implementation:
  public void f() { implementation.f(); }
  public void g() { implementation.g(); }
  public void h() { implementation.h(); }
}

class Implementation1 extends State {
  @Override public void f() {
    System.out.println("Implementation1.f()");
  }
  @Override public void g() {
    System.out.println("Implementation1.g()");
```

```
  }
  @Override public void h() {
    System.out.println("Implementation1.h()");
  }
}

class Implementation2 extends State {
  @Override public void f() {
    System.out.println("Implementation2.f()");
  }
  @Override public void g() {
    System.out.println("Implementation2.g()");
  }
  @Override public void h() {
    System.out.println("Implementation2.h()");
  }
}

public class StateDemo {
  static void test(State s) {
    s.f();
    s.g();
    s.h();
  }
  public static void main(String[] args) {
    State s = new State(new Implementation1());
    test(s);
    System.out.println("Changing implementation");
    s.change(new Implementation2());
    test(s);
  }
}
/* Output:
Implementation1.f()
Implementation1.g()
Implementation1.h()
Changing implementation
Implementation2.f()
Implementation2.g()
Implementation2.h()
*/
```

`main()` starts by using the first implementation, then it swaps in the second implementation. `Implementation1` and `Implementation2` inherit `State`, but this is not required. As long as you have an object that forwards calls to a member object, and that member object can be dynamically changed, you are utilizing the *State* pattern. Here's a second example where the `State` controller is a distinct type from the object to which the actions are forwarded:

```
// patterns/CleanTheFloor.java
// State with different compositional interface.

class FloorCleaner {
  private Attachment attachment = new Vacuum();
  public void change(Attachment newAttachment) {
    attachment = newAttachment;
  }
  public void clean() { attachment.action(); }
}

interface Attachment {
  void action();
}

class Vacuum implements Attachment {
  @Override public void action() {
    System.out.println("Vacuuming");
  }
}

class Mop implements Attachment {
  @Override public void action() {
    System.out.println("Mopping");
  }
}

public class CleanTheFloor {
  public static void main(String[] args) {
    FloorCleaner floorCleaner = new FloorCleaner();
    floorCleaner.clean();
    floorCleaner.change(new Mop());
    floorCleaner.clean();
  }
```

```
}
/* Output:
Vacuuming
Mopping
*/
```

The difference between *Proxy* and *State* is in the problems they solve. The common uses for *Proxy* as described in *Design Patterns* are:

1. **Remote proxy**. This proxies for an object in a different address space. For example, a remote proxy is automatically created for you by Java's *Remote Method Invocation* (RMI) compiler `rmic`.
2. **Virtual proxy**. This provides "lazy initialization" to create expensive objects on demand. If you never need to create that object, you don't pay the price. You might also want to shorten your startup times by waiting to initialize the object until the point where you need it.
3. **Protection proxy**. Used when you don't want to give the client programmer full access to the proxied object.
4. **Smart reference**. Performs additional actions when the proxied object is accessed. For example:
   - Keep track of the number of references held for a particular object.
   - Implement the *copy-on-write* idiom to prevent object aliasing.
   - Count the number of calls to a particular method.

You can look at a Java reference as a kind of protection proxy because it controls access to the actual object on the heap (and ensures, for example, that you don't call a method on a `null` reference).

In the *Design Patterns* book, *Proxy* and *State* are not seen as related to each other. As we've seen, *State* can use a separate implementation hierarchy. Similarly, *Proxy* need not use the same base class for its implementation, as long as the proxy object is a surrogate for another object. Regardless of the specifics, in both *Proxy* and *State* a surrogate is passing method calls through to an implementation object.

State Machine

This pattern also appears in the Enumerations chapter.

While *State* allows the client programmer to change the implementation, *State Machine* imposes a structure to automatically change the implementation. The implementation represents the current state of the system. The system behaves differently from one state to the next because it incorporates the *State* pattern.

The code that moves the system from one state to the next is often a *Template Method*, as seen in this example:

```
// patterns/state/StateMachineDemo.java
// The State Machine pattern.
// {java patterns.state.StateMachineDemo}
package patterns.state;
import java.util.*;
import onjava.Nap;

interface State {
  void run();
}

abstract class StateMachine {
  protected State currentState;
  protected abstract boolean changeState();
  // Template method:
  protected final void runAll() {
    while(changeState())
      currentState.run();
  }
}

// A different subclass for each state:

class Wash implements State {
  @Override public void run() {
    System.out.println("Washing");
    new Nap(0.5);
  }
}

class Spin implements State {
  @Override public void run() {
    System.out.println("Spinning");
    new Nap(0.5);
```

```
  }
}

class Rinse implements State {
  @Override public void run() {
    System.out.println("Rinsing");
    new Nap(0.5);
  }
}

class Washer extends StateMachine {
  private int i = 0;
  private Iterator<State> states =
    Arrays.asList(
      new Wash(), new Spin(),
      new Rinse(), new Spin()
    ).iterator();
  Washer() { runAll(); }
  @Override public boolean changeState() {
    if(!states.hasNext())
      return false;
    // Set the surrogate reference
    // to a new State object:
    currentState = states.next();
    return true;
  }
}

public class StateMachineDemo {
  public static void main(String[] args) {
    new Washer();
  }
}
/* Output:
Washing
Spinning
Rinsing
Spinning
*/
```

Here, `StateMachine` controls the current state and decides the next state. However, the state objects themselves may also decide the next state, typically based on some

kind of input to the system. This is usually a more flexible solution.

Factories: Encapsulating Object Creation

When an existing system must accept new types, a sensible first step is to use polymorphism to create a common interface for those new types. This isolates the code in your system from knowing the specific types you add. New types may be added without disturbing existing code … or so it seems.

It can appear that the only place you must change the code is the place where you inherit a new type. This is not quite true. You must still create objects of your new type, and at the point of creation you must specify the exact constructor to use. If this kind of object creation is distributed throughout your application, adding new types means finding all the places where type matters.

A *Factory* forces object creation to occur through a common point, preventing creational code from appearing throughout your system. All the code in your program must go through this factory to create one of your objects. Now when you add a new class to the system, you only modify the factory.

Every object-oriented program creates objects. Because you will sometimes write programs that are extended by adding new types, you're likely to find factories useful.

We'll use a `Shape` hierarchy for the *Factory* examples. If a `Shape` cannot be created, we need an appropriate exception:

```
// patterns/shapes/BadShapeCreation.java
package patterns.shapes;

public class BadShapeCreation
extends RuntimeException {
  public BadShapeCreation(String msg) {
    super(msg);
  }
}
```

Next, the `Shape` base class:

```
// patterns/shapes/Shape.java
package patterns.shapes;

public class Shape {
  private static int counter = 0;
  private int id = counter++;
  @Override public String toString() {
    return
      getClass().getSimpleName() + "[" + id + "]";
  }
  public void draw() {
    System.out.println(this + " draw");
  }
  public void erase() {
    System.out.println(this + " erase");
  }
}
```

Shape automatically creates a unique `id` for each object. `toString()` uses reflection to discover the name of the specific Shape subtype.

Now we can create some Shape classes:

```
// patterns/shapes/Circle.java
package patterns.shapes;

public class Circle extends Shape {}


// patterns/shapes/Square.java
package patterns.shapes;

public class Square extends Shape {}


// patterns/shapes/Triangle.java
package patterns.shapes;

public class Triangle extends Shape {}
```

A *Factory* has a method to create objects:

```
// patterns/shapes/FactoryMethod.java
package patterns.shapes;

public interface FactoryMethod {
  Shape create(String type);
}
```

`create()` takes a parameter that determines the type of `Shape` to create. Here, the parameter happens to be a `String`, but it could be any set of data. This initialization data presumably comes from somewhere outside the system.

Here's a reusable test for `FactoryMethod` implementations:

```
// patterns/shapes/FactoryTest.java
package patterns.shapes;
import java.util.stream.*;

public class FactoryTest {
  public static void test(FactoryMethod factory) {
    Stream.of("Circle", "Square", "Triangle",
      "Square", "Circle", "Circle", "Triangle")
      .map(factory::create)
      .peek(Shape::draw)
      .peek(Shape::erase)
      .count(); // Terminal operation
  }
}
```

Remember that a `Stream` doesn't do anything until you put a terminal operation on the end. Here, the value of `count()` is discarded.

One approach to a *Factory* is explicitly creating each type:

```
// patterns/ShapeFactory1.java
// A basic static factory method.
import java.util.*;
import java.util.stream.*;
import patterns.shapes.*;

public class ShapeFactory1 implements FactoryMethod {
  @Override public Shape create(String type) {
    switch(type) {
      case "Circle": return new Circle();
      case "Square": return new Square();
      case "Triangle": return new Triangle();
      default:
        throw new BadShapeCreation(type);
    }
  }
  public static void main(String[] args) {
    FactoryTest.test(new ShapeFactory1());
  }
}
/* Output:
Circle[0] draw
Circle[0] erase
Square[1] draw
Square[1] erase
Triangle[2] draw
Triangle[2] erase
Square[3] draw
Square[3] erase
Circle[4] draw
Circle[4] erase
Circle[5] draw
Circle[5] erase
Triangle[6] draw
Triangle[6] erase
*/
```

`create()` is now the only other code in the system you must modify when adding a new type of `Shape`.

A Dynamic Factory

`ShapeFactory1.java` is a reasonable solution, as it encapsulates the process of creating objects. However, it would be nice if you didn't have to modify anything when you add a new class, not even the factory. `ShapeFactory2.java` uses reflection (introduced in the Reflection chapter) to dynamically load the `Constructor` for a particular `Shape` into the `factories` map the first time it is needed:

```
// patterns/ShapeFactory2.java
import java.util.*;
import java.lang.reflect.*;
import java.util.stream.*;
import patterns.shapes.*;

public class ShapeFactory2 implements FactoryMethod {
  private Map<String, Constructor> factories =
    new HashMap<>();
  private static Constructor load(String id) {
    System.out.println("loading " + id);
    try {
      return Class.forName("patterns.shapes." + id)
        .getConstructor();
    } catch(ClassNotFoundException |
            NoSuchMethodException e) {
      throw new BadShapeCreation(id);
    }
  }
  @Override public Shape create(String id) {
    try {
      return (Shape)factories
        .computeIfAbsent(id, ShapeFactory2::load)
        .newInstance();
    } catch(Exception e) {
      throw new BadShapeCreation(id);
    }
  }
  public static void main(String[] args) {
    FactoryTest.test(new ShapeFactory2());
  }
}
/* Output:
```

```
loading Circle
Circle[0] draw
Circle[0] erase
loading Square
Square[1] draw
Square[1] erase
loading Triangle
Triangle[2] draw
Triangle[2] erase
Square[3] draw
Square[3] erase
Circle[4] draw
Circle[4] erase
Circle[5] draw
Circle[5] erase
Triangle[6] draw
Triangle[6] erase
*/
```

The `create()` method generates new `Shapes` based on the `String` argument you pass it, using the `String` as a lookup key in the `factories` `HashMap`. The value returned is a `Constructor`, which is used to create the new `Shape` object by calling `newInstance()`.

However, when you begin running the program, the `factories` map is empty. `create()` uses `Map`'s `computeIfAbsent()` method to find the `Constructor` if it's already in the `Map`. If it isn't, `computeIfAbsent()` computes the `Constructor` using `load()` and inserts it into the `Map`. You can see from the output that each specific type of `Shape` is only loaded the first time it is requested, and from then on it is simply retrieved from the `Map`.

Polymorphic Factories

The *Factory Method* pattern allows different types of factories to be subclassed from the basic factory:

```
// patterns/ShapeFactory3.java
// Polymorphic factory methods.
import java.util.*;
import java.util.function.*;
import java.util.stream.*;
import patterns.shapes.*;

interface PolymorphicFactory {
  Shape create();
}

class RandomShapes implements Supplier<Shape> {
  private final PolymorphicFactory[] factories;
  private Random rand = new Random(42);
  RandomShapes(PolymorphicFactory... factories) {
    this.factories = factories;
  }
  @Override public Shape get() {
    return
      factories[rand.nextInt(factories.length)]
      .create();
  }
}

public class ShapeFactory3 {
  public static void main(String[] args) {
    RandomShapes rs = new RandomShapes(    // [1]
      Circle::new, Square::new, Triangle::new
    );
    Stream.generate(rs)
      .limit(6)
      .peek(Shape::draw)
      .peek(Shape::erase)
      .count();
  }
}
/* Output:
Triangle[0] draw
Triangle[0] erase
Circle[1] draw
Circle[1] erase
Circle[2] draw
```

```
Circle[2] erase
Triangle[3] draw
Triangle[3] erase
Circle[4] draw
Circle[4] erase
Square[5] draw
Square[5] erase
*/
```

Because `RandomShapes` is a `Supplier<Shape>`, it can produce a `Stream` using `Stream.generate()`. The `RandomShapes` constructor takes a variable argument list of `PolymorphicFactory` objects. A variable argument list comes through as an array, so that is how the list is stored internally. The `get()` method randomly indexes into this array and calls `create()` on the result to produce a new `Shape`.

The `RandomShapes` constructor call (**[1]**) is the only place that requires modification when we add a new type of `Shape`.

Whereas `ShapeFactory2.java` could potentially throw exceptions, there are none in this approach—it is deterministic at compile time.

Abstract Factories

The *Abstract Factory* pattern looks like the factory objects we've seen previously, but with several factory methods instead of just one. Each factory method produces a different kind of object. There are multiple factory types. Choosing a factory object determines the specific types of all objects created by that factory. The example given in *Design Patterns* implements portability across various *graphical user interfaces* (GUIs). You create a factory object appropriate to the GUI you're working with. When you ask it for a menu, button, slider, etc., it produces that GUI's version of that item. This isolates the effect of changing from one GUI to another.

Suppose you are creating a general-purpose gaming environment to support different types of games. Here's how it might look using an abstract factory:

```
// patterns/abstractfactory/GameEnvironment.java
// An example of the Abstract Factory pattern.
// {java patterns.abstractfactory.GameEnvironment}
package patterns.abstractfactory;
import java.util.function.*;

interface Obstacle {
  void action();
}

interface Player {
  void interactWith(Obstacle o);
}

class Kitty implements Player {
  @Override
  public void interactWith(Obstacle ob) {
    System.out.print("Kitty has encountered a ");
    ob.action();
  }
}

class Fighter implements Player {
  @Override
  public void interactWith(Obstacle ob) {
    System.out.print("Fighter now battles a ");
    ob.action();
  }
}

class Puzzle implements Obstacle {
  @Override public void action() {
    System.out.println("Puzzle");
  }
}

class Weapon implements Obstacle {
  @Override public void action() {
    System.out.println("Weapon");
  }
}
```

```
// The Abstract Factory:
class GameElementFactory {
  Supplier<Player> player;
  Supplier<Obstacle> obstacle;
}

// Concrete factories:
class KittiesAndPuzzles
extends GameElementFactory {
  KittiesAndPuzzles() {
    player = Kitty::new;
    obstacle = Puzzle::new;
  }
}

class Melee
extends GameElementFactory {
  Melee() {
    player = Fighter::new;
    obstacle = Weapon::new;
  }
}

public class GameEnvironment {
  private Player p;
  private Obstacle ob;
  public
  GameEnvironment(GameElementFactory factory) {
    p = factory.player.get();
    ob = factory.obstacle.get();
  }
  public void play() {
    p.interactWith(ob);
  }
  public static void main(String[] args) {
    GameElementFactory
      kp = new KittiesAndPuzzles(),
      ml = new Melee();
    GameEnvironment
      g1 = new GameEnvironment(kp),
      g2 = new GameEnvironment(ml);
    g1.play();
```

```
    g2.play();
  }
}
/* Output:
Kitty has encountered a Puzzle
Fighter now battles a Weapon
*/
```

`Players` interact with `Obstacles`. Each game has different types of `Players` and `Obstacles`. You determine the game by choosing a particular `GameElementFactory`. The `GameEnvironment` sets up and plays the game. In this example, setup and play is very simple, but those activities (the *initial conditions* and the *state change*) can determine much of the game's outcome. Here, `GameEnvironment` is not designed for inheritance, although that's an option.

This also contains an example of *Double Dispatching*, which is explained later.

Function Objects

A *Function Object* encapsulates a function. The point is to decouple the choice of function from the site where that function is called.

The term *Function Object* is mentioned but not used in *Design Patterns*. However, the theme of the *Function Object* is repeated in several patterns in that book.

Command

This is the *Function Object* in its purest sense: a method that's an object. You pass a *Function Object* as a parameter, and it produces different behavior.

The *Command* pattern was demonstrated in several places earlier in the book, within these chapters:

- Inner Classes
- Reflection
- Enumerations

In the following example, `show()`s code stays the same, and what differs is the `Command` objects that you pass to `show()`:

```
// patterns/CommandPattern.java
import java.util.*;

class Command {
  public final String msg;
  public Command(String msg) {
    this.msg = msg;
  }
}

public class CommandPattern {
  public static void show(Command cmd) {
    System.out.println(cmd.msg);
  }
  public static void main(String[] args) {
    show(new Command("First Command"));
    show(new Command("Second Command"));
  }
}
/* Output:
First Command
Second Command
*/
```

Here's a second example that creates a list of *Command* objects to form a "macro." Prior to Java 8, producing the effect of a standalone function required extra ceremony to explicitly wrap a method into an object. Java 8 lambdas create function objects, so the *Command* pattern becomes almost trivial:

```
// patterns/Macro.java
import java.util.*;

public class Macro {
  public static void main(String[] args) {
    List<Runnable> macro = new ArrayList<>(
      Arrays.asList(
        () -> System.out.print("Hello "),
        () -> System.out.println("World! ")
    ));
    macro.forEach(Runnable::run);
    macro.add(
      () -> System.out.print("I'm the command pattern!")
```

```
    );
    macro.forEach(Runnable::run);
  }
}
/* Output:
Hello World!
Hello World!
I'm the command pattern!
*/
```

The primary point of *Command* is to allow you to hand a desired action to a method or object. The `macro` queues a set of actions to be performed collectively. You dynamically create behavior, something you normally only do by writing new code.

`Macro.java` could be modified to interpret a script—see the *Interpreter* design pattern.

The *Design Patterns* book says that "Commands are an object-oriented replacement for callbacks." However, I think the word "back" is an essential part of the concept of callbacks. That is, I think a callback actually reaches back to the creator of the callback, but with a *Command* object you typically just create it and hand it to some method or object, and are not otherwise connected over time to the *Command* object. That's my take on it, anyway. Later in this chapter, I combine a group of design patterns under the heading of "callbacks."

Strategy

Strategy appears to be a family of *Command* classes, all inherited from the same base—a group of *Function Object*s. The difference is in the way these objects are used. *Command* solves a particular problem—for example, selecting files from a list by passing in a *Command* object that describes the desired files. The "thing that stays the same" is the body of the method that's called, and the part that varies is isolated in that *Command* object.

Strategy includes a "Context" which can be a surrogate class that controls the selection and use of the particular strategy object—similar to *State*! Here's what it looks like:

```
// patterns/strategy/StrategyPattern.java
// {java patterns.strategy.StrategyPattern}
package patterns.strategy;
import java.util.function.*;
import java.util.*;

// The common strategy base type:
class FindMinima {
  protected
  Function<List<Double>, List<Double>> algorithm;
}

// The various strategies, each producing dummy data:
class LeastSquares extends FindMinima {
  LeastSquares() {
    // Line is a sequence of points:
    algorithm = (line) -> Arrays.asList(1.1, 2.2);
  }
}

class Perturbation extends FindMinima {
  Perturbation() {
    algorithm = (line) -> Arrays.asList(3.3, 4.4);
  }
}

class Bisection extends FindMinima {
  Bisection() {
    algorithm = (line) -> Arrays.asList(5.5, 6.6);
  }
}

// The "Context" controls the strategy:
class MinimaSolver {
  private FindMinima strategy;
  MinimaSolver(FindMinima strategy) {
    this.strategy = strategy;
  }
  List<Double> minima(List<Double> line) {
    return strategy.algorithm.apply(line);
  }
  void changeAlgorithm(FindMinima newAlgorithm) {
```

```
    strategy = newAlgorithm;
  }
}

public class StrategyPattern {
  public static void main(String[] args) {
    MinimaSolver solver =
      new MinimaSolver(new LeastSquares());
    List<Double> line = Arrays.asList(
      1.0, 2.0, 1.0, 2.0, -1.0,
      3.0, 4.0, 5.0, 4.0 );
    System.out.println(solver.minima(line));
    solver.changeAlgorithm(new Perturbation());
    System.out.println(solver.minima(line));
    solver.changeAlgorithm(new Bisection());
    System.out.println(solver.minima(line));
  }
}
/* Output:
[1.1, 2.2]
[3.3, 4.4]
[5.5, 6.6]
*/
```

The `changeAlgorithm()` method in `MinimaSolver` plugs a different strategy into the `strategy` field. `minima()` now uses a different algorithm.

We can simplify the solution by incorporating the context into `FindMinima`:

```
// patterns/strategy/StrategyPattern2.java
// {java patterns.strategy.StrategyPattern2}
package patterns.strategy;
import java.util.function.*;
import java.util.*;

// "Context" is now incorporated:
class FindMinima2 {
  private
  Function<List<Double>, List<Double>> algorithm;
  FindMinima2() { leastSquares(); } // default
  // The various strategies:
  void leastSquares() {
```

```
    algorithm = (line) -> Arrays.asList(1.1, 2.2);
  }
  void perturbation() {
    algorithm = (line) -> Arrays.asList(3.3, 4.4);
  }
  void bisection() {
    algorithm = (line) -> Arrays.asList(5.5, 6.6);
  }
  List<Double> minima(List<Double> line) {
    return algorithm.apply(line);
  }
}

public class StrategyPattern2 {
  public static void main(String[] args) {
    FindMinima2 solver = new FindMinima2();
    List<Double> line = Arrays.asList(
      1.0, 2.0, 1.0, 2.0, -1.0,
      3.0, 4.0, 5.0, 4.0 );
    System.out.println(solver.minima(line));
    solver.perturbation();
    System.out.println(solver.minima(line));
    solver.bisection();
    System.out.println(solver.minima(line));
  }
}
/* Output:
[1.1, 2.2]
[3.3, 4.4]
[5.5, 6.6]
*/
```

`FindMinima2` encapsulates the different algorithms, and now also includes the context so it controls the choice of algorithm within a single class.

Chain of Responsibility

This pattern was introduced in the Enumerations chapter.

Chain of Responsibility might be thought of as a dynamic generalization of recursion using *Strategy* objects. You make a call, and each *Strategy* in a sequence tries to satisfy

the call. The process ends when one of the strategies is successful or the sequence ends. In recursion, one method calls itself over and over until it reaches a termination condition; with *Chain of Responsibility*, a method calls the same base-class method (with a different implementation) which calls another implementation of the base-class method, etc., until it reaches a termination condition.

Instead of calling a single method to satisfy a request, multiple methods in the chain have a chance to satisfy that request, so *Chain of Responsibility* has the flavor of an expert system. The chain is effectively a linked list, so it can be created or modified dynamically. You can also think of it as a more general, dynamically-built `switch` statement.

What you probably want from `StrategyPattern.java` is to find a solution by automatically trying the different algorithms until one works. *Chain of Responsibility* provides a way to do this.

`Result` is a *Messenger* containing a `success` flag so the recipient can tell whether the algorithm succeeded, and `line` to carry the actual data. You return `Result.fail` when an algorithm fails:

```
// patterns/chain/Result.java
// Carries the result or indicates failure.
package patterns.chain;
import java.util.*;

public class Result {
  public final boolean success;
  public final List<Double> line;
  public Result(List<Double> data) {
    success = true;
    line = data;
  }
  private Result() {
    success = false;
    line = Collections.<Double>emptyList();
  }
  public static final Result fail = new Result();
}
```

Because failure is expected, returning `Result.fail` is more appropriate than throwing an exception when a strategy fails.

```
// patterns/chain/ChainOfResponsibility.java
// {java patterns.chain.ChainOfResponsibility}
package patterns.chain;
import java.util.*;
import java.util.function.*;

interface Algorithm {
  Result algorithm(List<Double> line);
}

class FindMinima {
  public static Result test(
    boolean success, String id, double d1, double d2) {
    System.out.println(id);
    if(success) // Actual test/calculation here
      return new Result(Arrays.asList(d1, d2));
    else // Try the next one in the chain:
      return Result.fail;
  }
  public static Result leastSquares(List<Double> line) {
    return test(false, "LeastSquares", 1.1, 2.2);
  }
  public static Result perturbation(List<Double> line) {
    return test(false, "Perturbation", 3.3, 4.4);
  }
  public static Result bisection(List<Double> line) {
    return test(true, "Bisection", 5.5, 6.6);
  }
  static List<Function<List<Double>, Result>>
    algorithms = Arrays.asList(
      FindMinima::leastSquares,
      FindMinima::perturbation,
      FindMinima::bisection
    );
  public static Result minima(List<Double> line) {
    for(Function<List<Double>, Result> alg :
        algorithms) {
      Result result = alg.apply(line);
      if(result.success)
        return result;
    }
    return Result.fail;
```

```
  }
}

public class ChainOfResponsibility {
  public static void main(String[] args) {
    FindMinima solver = new FindMinima();
    List<Double> line = Arrays.asList(
      1.0, 2.0, 1.0, 2.0, -1.0,
      3.0, 4.0, 5.0, 4.0);
    Result result = solver.minima(line);
    if(result.success)
      System.out.println(result.line);
    else
      System.out.println("No algorithm found");
  }
}
/* Output:
LeastSquares
Perturbation
Bisection
[5.5, 6.6]
*/
```

Each `Algorithm` implementation has a different approach for the `algorithm()` method. In `FindMinima`, a `List` of `algorithms` is created (this is the "chain"), and the `minima()` method goes through this list searching for one that succeeds.

Changing the Interface

Sometimes the problem you're solving is as simple as "I don't have the interface I need." Two *Design Patterns* solve this problem: *Adapter* takes one type and produces an interface to some other type. *Facade* creates an interface to a set of classes, simply to provide a more comfortable way to deal with a library or bundle of resources.

Adapter

When you've got *this*, and you need *that*, *Adapter* solves the problem. The only requirement is to produce a *that*, and there are several ways to accomplish this adaptation:

```
// patterns/adapt/Adapter.java
// Variations on the Adapter pattern.
// {java patterns.adapt.Adapter}
package patterns.adapt;

class WhatIHave {
  public void g() {}
  public void h() {}
}

interface WhatIWant {
  void f();
}

class ProxyAdapter implements WhatIWant {
  WhatIHave whatIHave;
  ProxyAdapter(WhatIHave wih) {
    whatIHave = wih;
  }
  @Override public void f() {
    // Implement behavior using
    // methods in WhatIHave:
    whatIHave.g();
    whatIHave.h();
  }
}

class WhatIUse {
  public void op(WhatIWant wiw) {
    wiw.f();
  }
}

// Approach 2: build adapter use into op():
class WhatIUse2 extends WhatIUse {
  public void op(WhatIHave wih) {
    new ProxyAdapter(wih).f();
  }
}

// Approach 3: build adapter into WhatIHave:
class WhatIHave2 extends WhatIHave
```

```
implements WhatIWant {
  @Override public void f() {
    g();
    h();
  }
}

// Approach 4: use an inner class:
class WhatIHave3 extends WhatIHave {
  private class InnerAdapter implements WhatIWant{
    @Override public void f() {
      g();
      h();
    }
  }
  public WhatIWant whatIWant() {
    return new InnerAdapter();
  }
}

public class Adapter {
  public static void main(String[] args) {
    WhatIUse whatIUse = new WhatIUse();
    WhatIHave whatIHave = new WhatIHave();
    WhatIWant adapt= new ProxyAdapter(whatIHave);
    whatIUse.op(adapt);
    // Approach 2:
    WhatIUse2 whatIUse2 = new WhatIUse2();
    whatIUse2.op(whatIHave);
    // Approach 3:
    WhatIHave2 whatIHave2 = new WhatIHave2();
    whatIUse.op(whatIHave2);
    // Approach 4:
    WhatIHave3 whatIHave3 = new WhatIHave3();
    whatIUse.op(whatIHave3.whatIWant());
  }
}
```

With `ProxyAdapter`, I'm taking some liberties with the term "proxy". In the *Design Patterns* book, they assert that a proxy must have an identical interface to the object for which it is a surrogate. Taking the two words together as `ProxyAdapter` is perhaps more reasonable.

Facade

Here's a principle I apply when turning requirements into a first-cut object design:

> *If something is ugly, hide it inside an object.*

This is basically what *Facade* accomplishes. If you have a rather confusing collection of classes and interactions that the client programmer doesn't really need to see, you can hide it behind an interface that only presents what's necessary, in a more understandable form.

Facade is often implemented as singleton abstract factory. You can produce this effect using a class containing `static` factory methods:

```
// patterns/Facade.java

class A { A(int x) {} }
class B { B(long x) {} }
class C { C(double x) {} }

// Other classes that aren't exposed by the
// facade go here ...

public class Facade {
  static A makeA(int x) { return new A(x); }
  static B makeB(long x) { return new B(x); }
  static C makeC(double x) { return new C(x); }
  public static void main(String[] args) {
    // The client programmer gets the objects
    // by calling the static methods:
    A a = Facade.makeA(1);
    B b = Facade.makeB(1);
    C c = Facade.makeC(1.0);
  }
}
```

The example given in *Design Patterns* isn't really a *Facade* but just a class that uses the other classes.

Package as a Variation of *Facade*

To me, the *Facade* has a rather "procedural" (non-object-oriented) feel to it: you are just calling some functions that give you objects. And how different is it, really, from *Abstract Factory*? The point of *Facade* is to hide part of a library of classes (and their interactions) from the client programmer, to make the interface to that group of classes more digestible and easier to understand.

This is precisely what the packaging features in Java accomplish. Outside the library, you can only create and use `public` classes, and all the non-`public` classes are only accessible within the package. It's as if *Facade* is a built-in feature of Java.

To be fair, *Design Patterns* is written primarily for a C++ audience. Although C++ has namespaces to prevent clashes of globals and class names, this does not provide the class hiding mechanism you get with non-`public` classes in Java. Much of the time it seems that Java packages solve the *Facade* problem.

Interpreter: Run-Time Flexibility

In some applications, the user needs greater run-time flexibility, for example to create scripts describing the desired behavior of a system. The *Interpreter* design pattern creates and embeds a language interpreter into such a program.

Developing your own language and building an interpreter for it is a time-consuming distraction from the process of building your application. The best solution is to reuse code: embed an interpreter that's already been built and debugged for you. The Python language can be freely embedded in your for-profit application without any license agreement, royalties, or other encumbrances. In addition, there is a version of Python called Jython (outdated, but still works) which is entirely Java byte codes, so incorporating it into your Java application is relatively straightforward. Python is a scripting language that is easy to learn, logical to read and write, supports functions and objects, has a large set of available libraries, and runs on virtually every platform. You can download Python and learn more about it at www.Python.org[182].

[182] http://www.Python.org

Callbacks

Callbacks decouple code from behavior. These include *Observer*, and a category of callbacks called *Multiple Dispatching*, including *Visitor* from *Design Patterns*.

Observer

The *Observer* pattern solves a fairly common problem: What if a group of objects must update themselves when some other object changes state? This can be seen in the "model-view" aspect of SmallTalk's model-view-controller (MVC), or the almost-equivalent "Document-View Architecture." Suppose you have some data (the "document") and more than one view, say a plot and a textual view. When you change the data, the two views must update themselves, and that's what the observer facilitates. It's a common enough problem that its solution has long been part of the standard `java.util` library.

Observer allows you to decouple the source of the call from the called code in a completely dynamic way. Like other forms of callback, the *Observer* pattern contains a hook point where you can change code. The difference is in the *Observer*'s completely dynamic nature. It is often used for the specific case of changes triggered by other objects' change of state, but is also the basis of event management.

There are actually two "things that change" in the *Observer* pattern: the quantity of observing objects and the way an update occurs. The *Observer* pattern allows you to modify both of these without affecting the surrounding code.

`Observer` is an interface with a single method, `update(Observable o, Object arg)`. This method is called by the object that's observed—the `Observable`—when that object decides it's time to update all the `Observers`. The first argument is the `Observable` object that caused the update and the second argument provides any extra information the `Observer` might need to update itself.

An `Observer` can be registered with multiple `Observables`.

An `Observable` is the "observed object" that decides when and how to do the updating. The `Observable` class keeps track of every `Observer` that wants to be informed when a change happens. When something changes such that each

`Observer` should update itself, the `Observable` class calls its `notifyObservers()` method which calls `update()` on each `Observer`.

`Observable` has a flag to indicate whether it's been changed. This flag allows you to wait and only notify the `Observers` when you decide the time is right. Control of the flag's state is `protected`, so only an inheritor can decide what constitutes a change, not the end user of the resulting derived `Observer` class.

Most of the work is done in `notifyObservers()`. If the `changed` flag has not been set, calling `notifyObservers()` does nothing. Otherwise, it first clears the `changed` flag so repeated calls to `notifyObservers()` won't waste time. This is done before notifying the observers, in case the calls to `update()` cause changes to this `Observable` object. Then it calls the `update()` method for each `Observer`.

At first it may appear that you can use an ordinary `Observable` object to manage the updates. But this doesn't work; to get an effect, you *must* inherit from `Observable` and somewhere in your derived-class code call `setChanged()`. This is the method that sets the "changed" flag, which means when you call `notifyObservers()`, all observers will get notified. *Where* you call `setChanged()` depends on the logic of your program.

Note that `Observer` and `Observable` (which have been part of Java since version 1.0) are deprecated in Java 9 in favor of the `Flow` class introduced in Java 9.

Observing Flowers

Here is an example of the observer pattern:

```
// patterns/observer/ObservedFlower.java
// Demonstration of the Observer pattern.
// {java patterns.observer.ObservedFlower}
package patterns.observer;
import java.util.*;
import java.util.function.*;

@SuppressWarnings("deprecation")
class Flower {
  private boolean isOpen = false;
  private boolean alreadyOpen = false;
  private boolean alreadyClosed = false;
```

```
  Observable opening = new Observable() {
    @Override public void notifyObservers() {
      if(isOpen && !alreadyOpen) {
        setChanged();
        super.notifyObservers();
        alreadyOpen = true;
      }
    }
  };
  Observable closing = new Observable() {
    @Override public void notifyObservers() {
      if(!isOpen && !alreadyClosed) {
        setChanged();
        super.notifyObservers();
        alreadyClosed = true;
      }
    }
  };
  public void open() { // Opens its petals
    isOpen = true;
    opening.notifyObservers();
    alreadyClosed = false;
  }
  public void close() { // Closes its petals
    isOpen = false;
    closing.notifyObservers();
    alreadyOpen = false;
  }
}

@SuppressWarnings("deprecation")
class Bee {
  private String id;
  Bee(String name)  { id = name; }
  // Observe openings:
  public final Observer whenOpened = (ob, a) ->
    System.out.println(
      "Bee " + id + "'s breakfast time!");
  // Observe closings:
  public final Observer whenClosed = (ob, a) ->
    System.out.println(
      "Bee " + id + "'s bed time!");
```

```
}

@SuppressWarnings("deprecation")
class Hummingbird {
  private String id;
  Hummingbird(String name) { id = name; }
  public final Observer whenOpened = (ob, a) ->
    System.out.println("Hummingbird " +
      id + "'s breakfast time!");
  public final Observer whenClosed = (ob, a) ->
    System.out.println("Hummingbird " +
      id + "'s bed time!");
}

public class ObservedFlower {
  public static void main(String[] args) {
    Flower f = new Flower();
    Bee
      ba = new Bee("A"),
      bb = new Bee("B");
    Hummingbird
      ha = new Hummingbird("A"),
      hb = new Hummingbird("B");
    f.opening.addObserver(ha.whenOpened);
    f.opening.addObserver(hb.whenOpened);
    f.opening.addObserver(ba.whenOpened);
    f.opening.addObserver(bb.whenOpened);
    f.closing.addObserver(ha.whenClosed);
    f.closing.addObserver(hb.whenClosed);
    f.closing.addObserver(ba.whenClosed);
    f.closing.addObserver(bb.whenClosed);
    // Hummingbird B decides to sleep in.
    // Removing whenOpened stops open updates.
    f.opening.deleteObserver(hb.whenOpened);
    // A change that interests observers:
    f.open();
    f.open(); // No effect: it's already open.
    System.out.println("---------------");
    // Bee A doesn't want to go to bed.
    // Removing whenClosed stops close updates.
    f.closing.deleteObserver(ba.whenClosed);
    f.close();
```

```
    System.out.println("++++++++++++++++");
    f.close(); // No effect: it's already closed.
    System.out.println("================");
    f.opening.deleteObservers();
    f.open(); // No observers to update.
    System.out.println("################");
    f.close(); // Close observers are still there.
  }
}
/* Output:
Bee B's breakfast time!
Bee A's breakfast time!
Hummingbird A's breakfast time!
---------------
Bee B's bed time!
Hummingbird B's bed time!
Hummingbird A's bed time!
++++++++++++++++
================
################
Bee B's bed time!
Hummingbird B's bed time!
Hummingbird A's bed time!
*/
```

A `Flower` can `open()` or `close()`. Using anonymous inner classes, both events are separately-observable phenomena. `opening` and `closing` both inherit `Observable`, so they have access to `setChanged()`.

`Observer` is a functional interface (also known as a SAM type), so `whenOpened` and `whenClosed` in `Bee` and `Hummingbird` are defined using lambdas. `whenOpened` and `whenClosed` independently observe `Flower` openings and closings.

In `main()`, you see one of the prime benefits of the observer pattern: the ability to change behavior at runtime by dynamically registering and un-registering `Observers` with `Observables`.

Notice you can create other completely different observing objects; the only connection the `Observers` have with `Flowers` is the `Observer` interface.

A Visual Example of Observers

The following example cheats by using the Swing library to create graphics. Swing is not introduced in this book (See the Swing Documentation[183] or *Thinking in Java, 4th edition*, available at www.MindviewLLC.com). Boxes are placed in a grid on the screen and each one is initialized to a random color. In addition, each box `implements` the `Observer` interface and is registered with an `Observable` object. When you click on a box, all other boxes are notified because the `Observable` object automatically calls each `Observer` object's `update()` method. Inside this method, the box checks to see whether it is adjacent to the one that was clicked, and if so it changes its color to match the clicked box.

The `java.awt.event` library has a `MouseListener` class with multiple methods, but we are only interested in the `mouseClicked()` method. You can't write a lambda expression if you just want to implement `mouseClicked()` because `MouseListener` is not a functional interface, due to its multiple methods. Java 8 allows us to simplify our code using the `default` keyword to create a helper interface which solves the problem:

```
// onjava/MouseClick.java
// Helper interface to allow lambda expressions.
package onjava;
import java.awt.event.*;

// Default everything except mouseClicked():
public interface MouseClick extends MouseListener {
    @Override
    default void mouseEntered(MouseEvent e) {}
    @Override
    default void mouseExited(MouseEvent e) {}
    @Override
    default void mousePressed(MouseEvent e) {}
    @Override
    default void mouseReleased(MouseEvent e) {}
}
```

Now you can successfully cast a lambda expression to a `MouseClick` and pass it to `addMouseListener()`.

[183] https://docs.oracle.com/javase/8/docs/technotes/guides/swing/index.html

```
// patterns/BoxObserver.java
// Demonstration of the Observer pattern using
// Java's built-in observer classes.
// {ExcludeFromGradle} // Won't work under WSL2
import javax.swing.*;
import java.awt.*;
import java.awt.event.*;
import java.util.*;
import onjava.*;
import onjava.MouseClick;

@SuppressWarnings("deprecation")
class Boxes extends JFrame {
  Observable notifier = new Observable() {
    @Override
    public void notifyObservers(Object b) {
      setChanged();
      super.notifyObservers(b);
    }
  };
  public Boxes(int grid) {
    setTitle("Demonstrates Observer pattern");
    Container cp = getContentPane();
    cp.setLayout(new GridLayout(grid, grid));
    for(int x = 0; x < grid; x++)
      for(int y = 0; y < grid; y++)
        cp.add(new Box(x, y, notifier));
    setSize(500, 400);
    setVisible(true);
    setDefaultCloseOperation(DISPOSE_ON_CLOSE);
  }
}

@SuppressWarnings("deprecation")
class Box extends JPanel implements Observer {
  int x, y; // Location in grid
  Color color = COLORS[
    (int)(Math.random() * COLORS.length)
  ];
  static final Color[] COLORS = {
    Color.black, Color.blue, Color.cyan,
    Color.darkGray, Color.gray, Color.green,
```

```
    Color.lightGray, Color.magenta,
    Color.orange, Color.pink, Color.red,
    Color.white, Color.yellow
  };
  Box(int x, int y, Observable notifier) {
    this.x = x;
    this.y = y;
    notifier.addObserver(this);
    addMouseListener((MouseClick)
      e -> notifier.notifyObservers(Box.this));
  }
  @Override public void paintComponent(Graphics g) {
    super.paintComponent(g);
    g.setColor(color);
    Dimension s = getSize();
    g.fillRect(0, 0, s.width, s.height);
  }
  @Override
  public void update(Observable o, Object arg) {
    Box clicked = (Box)arg;
    if(nextTo(clicked)) {
      color = clicked.color;
      repaint();
    }
  }
  private boolean nextTo(Box b) {
    return Math.abs(x - b.x) <= 1 &&
           Math.abs(y - b.y) <= 1;
  }
}

public class BoxObserver {
  public static void main(String[] args) {
    int grid = 8;
    if(args.length > 0)
      grid = Integer.parseInt(args[0]);
    new Boxes(grid);
  }
}
```

When you first look at the online documentation for `Observable`, it's a bit confusing because it appears you can use an ordinary `Observable` object to manage the

updates. But this doesn't work; to get an effect, you *must* inherit from `Observable` and somewhere in your derived-class code call `setChanged()`, as seen in `Boxes.notifier`. `setChanged()` sets the "changed" flag, which means when you call `notifyObservers()`, all observers will get notified. Here, `setChanged()` is called within `notifyObservers()`, but you can use any criterion to decide when to call `setChanged()`.

`Boxes` contains a single `Observable` object called `notifier`, and every time a `Box` is created, it is tied to `notifier`. In `Box`, whenever you click the mouse, `notifyObservers()` is called, passing the clicked object as an argument so all boxes receiving the message (in their `update()` method) know who was clicked and can decide whether to change themselves. Using a combination of code in `notifyObservers()` and `update()` you can work out some fairly complex schemes.

Multiple Dispatching

Programs can get especially messy when dealing with multiple interacting types. For example, consider a system that parses and executes mathematical expressions. You want to say `Number + Number`, `Number * Number`, etc., where `Number` is the base class for a family of numerical objects. But when you say `a + b`, and you don't know the exact type of either `a` or `b`, how can you get them to interact properly?

The answer starts with something you probably don't think about: Java performs only single dispatching. That is, when operating on more than one object whose type is unknown, Java can invoke the dynamic binding mechanism on only one of those types. This doesn't solve the problem of working with multiple upcast objects, so you end up detecting some types manually and effectively producing your own dynamic binding behavior.

The solution is called *Multiple Dispatching* (Often there are only two dispatches, which is called *Double Dispatching*). Because polymorphism only occurs via method calls, if you want double dispatching to occur, there must be two method calls: the first to determine the first unknown type, and the second to determine the second unknown type.

Generally, you'll create a method that produces more than one dynamic method call and thus determines more than one type in the process.

The "paper-scissors-rock" game was introduced in the Enumerations chapter in the RoShamBo1.java example. We'll create a variation using that framework, but with a standalone ItemFactory and Compete.match() function. The methods compete() and eval() are both members of the same Item type (If you have two different interacting type hierarchies, you need a polymorphic method call in each hierarchy):

```
// patterns/PaperScissorsRock.java
// Demonstration of multiple dispatching.
import java.util.*;
import java.util.function.*;
import java.util.stream.*;
import enums.Outcome;
import static enums.Outcome.*;
import enums.Item;
import enums.Paper;
import enums.Scissors;
import enums.Rock;
import onjava.*;
import static onjava.Tuple.*;

class ItemFactory {
  static List<Supplier<Item>> items =
    Arrays.asList(
      Scissors::new, Paper::new, Rock::new);
  static final int SZ = items.size();
  private static SplittableRandom rand =
    new SplittableRandom(47);
  public static Item newItem() {
    return items.get(rand.nextInt(SZ)).get();
  }
  public static Tuple2<Item,Item> newPair() {
    return tuple(newItem(), newItem());
  }
}

class Compete {
  public static Outcome match(Tuple2<Item,Item> p) {
    System.out.print(p.a1 + " -> " + p.a2 + " : ");
    return p.a1.compete(p.a2);
  }
}
```

```
public class PaperScissorsRock {
  public static void main(String[] args) {
    Stream.generate(ItemFactory::newPair)
      .limit(20)
      .map(Compete::match)
      .forEach(System.out::println);
  }
}
/* Output:
Scissors -> Rock : LOSE
Scissors -> Paper : WIN
Rock -> Paper : LOSE
Rock -> Rock : DRAW
Rock -> Paper : LOSE
Paper -> Scissors : LOSE
Rock -> Paper : LOSE
Scissors -> Scissors : DRAW
Scissors -> Rock : LOSE
Scissors -> Paper : WIN
Scissors -> Rock : LOSE
Paper -> Scissors : LOSE
Rock -> Rock : DRAW
Scissors -> Scissors : DRAW
Paper -> Paper : DRAW
Scissors -> Paper : WIN
Scissors -> Rock : LOSE
Scissors -> Paper : WIN
Rock -> Paper : LOSE
Rock -> Scissors : WIN
*/
```

The `Item` interface contains the structure of the double dispatch: `compete()` performs the first dispatch, and the second dispatch occurs through the call to `eval()`.

Suppose you have two `Items`, `a` and `b`, and you don't know the type of either. Here's what happens as you call `a.compete(b)`: The `compete()` method is dynamically dispatched, so you wake up inside the particular `compete()` body for the type of `a`. If `a` is type `Paper`, then we wake up in `Paper`'s `compete()` body, thus determining the type of the first unknown object via the first dispatch. But now `compete()` turns around and calls `eval()` on `b`, the second unknown type, while passing `a` as an argument, so the overloaded version of `eval()` is called for the type of `b`—and that's

the second dispatch. At that point, you're inside an `eval()` that knows the type of both objects.

Visitor, a Type of Multiple Dispatching

The assumption for *Visitor* is that you have a primary class hierarchy that is unchangeable—perhaps it's from a vendor and you can't make modifications to that hierarchy. However, you'd like to add a new polymorphic operation to that hierarchy, which normally means you'd have to add a method to the base class interface. The dilemma is that you must add methods to the base class, but you can't touch the base class. How do you get around this?

Visitor (the final pattern in the *Design Patterns* book) solves this problem by building on the double-dispatching scheme from the previous section.

Visitor allows you to extend the interface of the primary type by creating a separate class hierarchy of type `Visitor` to virtualize the operations performed upon the primary type. The objects of the primary type simply "accept" the visitor, then call the visitor's dynamically-bound method.

```
// patterns/visitor/BeeAndFlowers.java
// Demonstration of the Visitor pattern.
// {java patterns.visitor.BeeAndFlowers}
package patterns.visitor;
import java.util.*;
import java.util.function.*;
import java.util.stream.*;

interface Visitor {
  void visit(Gladiolus g);
  void visit(Ranunculus r);
  void visit(Chrysanthemum c);
}

// The Flower hierarchy cannot be changed:
interface Flower {
  void accept(Visitor v);
}

class Gladiolus implements Flower {
```

```
  @Override
  public void accept(Visitor v) { v.visit(this);}
}

class Ranunculus implements Flower {
  @Override
  public void accept(Visitor v) { v.visit(this);}
}

class Chrysanthemum implements Flower {
  @Override
  public void accept(Visitor v) { v.visit(this);}
}

// Add the ability to produce a String:
class StringVal implements Visitor {
  private String s;
  @Override public String toString() { return s; }
  @Override public void visit(Gladiolus g) {
    s = "Gladiolus";
  }
  @Override public void visit(Ranunculus r) {
    s = "Ranunculus";
  }
  @Override public void visit(Chrysanthemum c) {
    s = "Chrysanthemum";
  }
}

// Add the ability to do "Bee" activities:
class Bee implements Visitor {
  @Override public void visit(Gladiolus g) {
    System.out.println("Bee and Gladiolus");
  }
  @Override public void visit(Ranunculus r) {
    System.out.println("Bee and Ranunculus");
  }
  @Override public void visit(Chrysanthemum c) {
    System.out.println("Bee and Chrysanthemum");
  }
}
```

```
class FlowerFactory {
  static List<Supplier<Flower>> flowers =
    Arrays.asList(Gladiolus::new,
      Ranunculus::new, Chrysanthemum::new);
  static final int SZ = flowers.size();
  private static SplittableRandom rand =
    new SplittableRandom(47);
  public static Flower newFlower() {
    return flowers.get(rand.nextInt(SZ)).get();
  }
}

public class BeeAndFlowers {
  public static void main(String[] args) {
    List<Flower> flowers =
      Stream.generate(FlowerFactory::newFlower)
        .limit(10)
        .collect(Collectors.toList());
    StringVal sval = new StringVal();
    flowers.forEach(f -> {
      f.accept(sval);
      System.out.println(sval);
    });
    // Perform "Bee" operation on all Flowers:
    Bee bee = new Bee();
    flowers.forEach(f -> f.accept(bee));
  }
}
/* Output:
Gladiolus
Chrysanthemum
Gladiolus
Ranunculus
Chrysanthemum
Ranunculus
Chrysanthemum
Chrysanthemum
Chrysanthemum
Ranunculus
Bee and Gladiolus
Bee and Chrysanthemum
Bee and Gladiolus
```

```
Bee and Ranunculus
Bee and Chrysanthemum
Bee and Ranunculus
Bee and Chrysanthemum
Bee and Chrysanthemum
Bee and Chrysanthemum
Bee and Ranunculus
*/
```

Note the similarity with the previous example: `Flower` accepts a `Visitor` for the first dispatch, then turns around and calls `visit()` (passing itself as an argument, to end up in the overloaded method according to the `Flower` type) as the second dispatch.

In `main()`, it's almost as if I've added methods to `Flower` to produce a `String` representation and to perform `Bee` activities. This is the point of `Visitor`: adding methods to a frozen hierarchy.

Pattern Refactoring

The remainder of this chapter looks at the process of solving a problem by applying design patterns in an evolutionary fashion. A first-cut design is used for the initial solution, then this solution is examined and various design patterns are applied to the problem—some work and some don't. The key question in seeking improved solutions is always "what will change?"

This process is similar to what Martin Fowler talks about in his book *Refactoring: Improving the Design of Existing Code* (Addison-Wesley, 1999), although he tends to talk about pieces of code more than pattern-level designs. You start with a solution, and when you discover it doesn't continue to meet your needs, you fix it. This is a natural tendency, but in computer programming it's been difficult to accomplish with procedural programs. Accepting the idea that we *can* refactor code and designs is the first step toward improving our systems.

The example we refactor is a trash recycling system. Trash arrives unsorted at the trash recycling plant, and we model the sorting and evaluation of that trash. In the initial solution, reflection (described in the Reflection chapter) takes anonymous pieces of trash and discovers their type in order to sort them.

`Trash` and Subclasses

The base class for `Trash` contains `weight` and `price()` information, along with a `toString()` that uses reflection to produce the exact name of the `Trash` subclass. There's also an `accept()` method that we will use later for the *Visitor* pattern, but until then you can ignore it.

```
// patterns/trash/Trash.java
// Base class for Trash recycling examples.
package patterns.trash;

public abstract class Trash {
  public final double weight;
  public Trash(double weight) {
    this.weight = weight;
  }
  public abstract double price();
  @Override public String toString() {
    return String.format(
      "%s weight: %.2f * price: %.2f = %.2f",
      getClass().getSimpleName(),
      weight, price(), weight * price());
  }
  // Ignore for now; to be used later:
  public abstract void accept(Visitor v);
}
```

`price()` is an `abstract` method that returns the current selling price for that material. This price does not change from one piece of `Trash` to the next—the price of `Aluminum`, for example, is constant. It is convenient to put all the prices in one place so they can be easily changed. With an `interface`, each field is automatically `public`, `static` and `final`:

```
// patterns/trash/Price.java
package patterns.trash;

public interface Price {
  double
    ALUMINUM = 1.67,
    PAPER = 0.10,
    GLASS = 0.23,
    CARDBOARD = 0.11;
}
```

The `price()` for each different type of `Trash` returns the appropriate field from `Price`:

```
// patterns/trash/Aluminum.java
package patterns.trash;

public class Aluminum extends Trash {
  public Aluminum(double wt) { super(wt); }
  @Override public double price() {
    return Price.ALUMINUM;
  }
  // Ignore for now; to be used later:
  @Override public void accept(Visitor v) {
    v.visit(this);
  }
}
```

```
// patterns/trash/Paper.java
package patterns.trash;

public class Paper extends Trash {
  public Paper(double wt) { super(wt); }
  @Override public double price() {
    return Price.PAPER;
  }
  // Ignore for now; to be used later:
  @Override public void accept(Visitor v) {
    v.visit(this);
  }
}
```

```
// patterns/trash/Glass.java
package patterns.trash;

public class Glass extends Trash {
  public Glass(double wt) { super(wt); }
  @Override public double price() {
    return Price.GLASS;
  }
  // Ignore for now; to be used later:
  @Override public void accept(Visitor v) {
    v.visit(this);
  }
}
```

```
// patterns/trash/Cardboard.java
package patterns.trash;

public class Cardboard extends Trash {
  public Cardboard(double wt) { super(wt); }
  @Override public double price() {
    return Price.CARDBOARD;
  }
  // Ignore for now; to be used later:
  @Override public void accept(Visitor v) {
    v.visit(this);
  }
}
```

`TrashValue` is a utility class with a single `static` function, `sum()`. This takes a `List` of `Trash` and displays each piece, and finally the total value of all `Trash` in the `List`:

```
// patterns/trash/TrashValue.java
// Sums the value of Trash in a bin.
package patterns.trash;
import java.util.*;

public class TrashValue {
  private static double total;
  public static void
  sum(List<? extends Trash> bin, String type) {
    total = 0.0;
    bin.forEach( t -> {
      System.out.println(t);
      total += t.weight * t.price();
    });
    System.out.printf(
      "Total %s value = %.2f%n", type, total);
  }
}
```

Defining `total` outside of `sum()` looks unnecessary because it is never used by anything but `sum()`. However, if you try to define it as a local variable within `sum()`, you'll get the error message "local variables referenced from a lambda expression must be final or effectively final."

Once a `List<Trash>` is full of `Trash` objects, the `Bins` constructor sorts that `List` into its typed bins using `instanceof`:

```
// patterns/trash/Bins.java
package patterns.trash;
import java.util.*;

public class Bins {
  final List<Trash> bin;
  final List<Aluminum> aluminum = new ArrayList<>();
  final List<Paper> paper = new ArrayList<>();
  final List<Glass> glass = new ArrayList<>();
  final List<Cardboard> cardboard = new ArrayList<>();
  public Bins(List<Trash> source) {
    bin = new ArrayList<>(source); // Copy
    bin.forEach( t -> {
      // Use reflection to discover Trash type:
      if(t instanceof Aluminum)
```

```
        aluminum.add((Aluminum)t);
      if(t instanceof Paper)
        paper.add((Paper)t);
      if(t instanceof Glass)
        glass.add((Glass)t);
      if(t instanceof Cardboard)
        cardboard.add((Cardboard)t);
    });
  }
  public void show() {
    TrashValue.sum(aluminum, "Aluminum");
    TrashValue.sum(paper, "Paper");
    TrashValue.sum(glass, "Glass");
    TrashValue.sum(cardboard, "Cardboard");
    TrashValue.sum(bin, "Trash");
  }
}
```

Adding a new type of trash means adding a new `List` and a new `instanceof` to `Bins`, along with another line in `show()`.

Now we're ready to create a simple factory, which keeps a `List` called `constructors` containing constructors that create new `Trash` objects. When you add a new type of `Trash`, you must add its constructor to `constructors`.

```
// patterns/recyclea/RecycleA.java
// Recycling with reflection.
// {java patterns.recyclea.RecycleA}
package patterns.recyclea;
import java.util.*;
import java.util.function.*;
import java.util.stream.*;
import patterns.trash.*;

class SimpleFactory {
  static final
  List<Function<Double, Trash>> constructors =
    Arrays.asList(
      Aluminum::new, Paper::new, Glass::new);
  static final int SIZE = constructors.size();
  private static SplittableRandom rand =
    new SplittableRandom(42);
```

```
  public static Trash random() {
    return constructors
      .get(rand.nextInt(SIZE))
      .apply(rand.nextDouble());
  }
}

public class RecycleA {
  public static void main(String[] args) {
    List<Trash> bin =
      Stream.generate(SimpleFactory::random)
        .limit(10)
        .collect(Collectors.toList());
    Bins bins = new Bins(bin);
    bins.show();
  }
}
/* Output:
Aluminum weight: 0.34 * price: 1.67 = 0.57
Aluminum weight: 0.62 * price: 1.67 = 1.03
Aluminum weight: 0.49 * price: 1.67 = 0.82
Aluminum weight: 0.50 * price: 1.67 = 0.83
Total Aluminum value = 3.26
Paper weight: 0.69 * price: 0.10 = 0.07
Total Paper value = 0.07
Glass weight: 0.16 * price: 0.23 = 0.04
Glass weight: 0.87 * price: 0.23 = 0.20
Glass weight: 0.80 * price: 0.23 = 0.18
Glass weight: 0.52 * price: 0.23 = 0.12
Glass weight: 0.20 * price: 0.23 = 0.05
Total Glass value = 0.59
Total Cardboard value = 0.00
Glass weight: 0.16 * price: 0.23 = 0.04
Aluminum weight: 0.34 * price: 1.67 = 0.57
Glass weight: 0.87 * price: 0.23 = 0.20
Glass weight: 0.80 * price: 0.23 = 0.18
Aluminum weight: 0.62 * price: 1.67 = 1.03
Aluminum weight: 0.49 * price: 1.67 = 0.82
Glass weight: 0.52 * price: 0.23 = 0.12
Glass weight: 0.20 * price: 0.23 = 0.05
Aluminum weight: 0.50 * price: 1.67 = 0.83
Paper weight: 0.69 * price: 0.10 = 0.07
```

```
Total Trash value = 3.91
*/
```

Notice that `SimpleFactory` doesn't produce `Cardboard`, because there's no `Cardboard::new` in `constructors`.

The `constructors` type is not a `List<Constructor>`; instead the `List` holds `Function<Double, Trash>`. This describes a `Function` object that takes a `Double` argument and returns a `Trash` object—which is exactly what the single-parameter `Trash` constructors do. They, however, take `doubles` instead of the wrapped `Doubles`, so Java is applying autoboxing.

`random()` fetches the `Function` for a random type of `Trash` and invokes `apply()` to call that `Function`, passing it a randomly-generated `double` as the argument.

In `main()`, the argument to `Stream.generate()` is *not* a `Supplier<T>` which is the specified parameter type for `generate()`. It works even though the function that produces `Trash` is named `random()` and not `get()`.

The program satisfies the design requirements: it works. This might be fine as long as it's a one-shot solution. However, a useful program tends to evolve over time, so you must ask, "What if the situation changes?" For example, plastic is a valuable recyclable commodity. How should we integrate it into the system, especially if the program is large and complicated?. Although `SimpleFactory` does encapsulate creation, the rest of the program has type-check coding scattered about, so you must go find that code every time a new type is added. If you miss a spot, the compiler doesn't produce any helpful errors.

You know you're misusing reflection when *every type is tested*. If you're looking for only a subset of types because that subset needs special treatment, that's probably fine. But if you're hunting for every type inside a switch statement, then there's probably a way to improve the design to make it more maintainable. In the remainder of the chapter we evolve the program through multiple phases to make it more flexible. This should prove a valuable example in program design.

A *Messenger* Object

A general object-oriented design principle I first heard spoken by Grady Booch is: "If the design is too complicated, make more objects." This is simultaneously

counterintuitive and ludicrously simple, and I find it quite useful. ("Make more objects" is often equivalent to "add another level of abstraction.") In general, if you find a place with messy code, consider what sort of class might clean it up. Often the side effect of cleaning up the code is a more flexible system with better structure.

`SimpleFactory` is a reasonable first-cut solution, but what if the derived `Trash` constructor requires more or different arguments? "Creating more objects" solves this problem. To hide creational data, `TrashInfo` contains all necessary information that a factory needs to create the appropriate `Trash` object:

```
// patterns/trash/TrashInfo.java
// Messenger class carrying Trash creation data.
package patterns.trash;

public class TrashInfo {
  public final String type;
  public final double data;
  public TrashInfo(String type, double data) {
    this.type = type;
    this.data = data;
  }
  @Override public String toString() {
    return "TrashInfo(" + type + ", " + data + ")";
  }
}
```

A `TrashInfo` object's only job is to hold and convey information, so it's called a *Messenger Object* or a *Data Transfer Object*. Messengers are typically immutable, so both fields are `final` and `public`.

If something changes so the factory needs more or different information to create a new type of `Trash` object, you don't need to change the factory parameters. The `TrashInfo` class can be modified directly by adding new data and new constructors, or by subclassing.

Generalizing the *Factory*

If types are added to `RecycleA.java`, `SimpleFactory` must be modified for each new type. In `DynaFactory`, we'll use reflection (as seen in `ShapeFactory2.java`) to produce a `Constructor` for that type of `Trash`. Now the factory doesn't change when a new type is added to the system:

```
// patterns/trash/DynaFactory.java
// Dynamic discovery of Trash types.
package patterns.trash;
import java.util.*;
import java.util.function.*;
import java.lang.reflect.*;

public class DynaFactory {
  private Map<String, Constructor> constructors =
    new HashMap<>();
  private String packageName;
  public DynaFactory(String packageName) {
    this.packageName = packageName;
  }
  @SuppressWarnings("unchecked")
  public
  <T extends Trash> T create(TrashInfo info) {
    try {
      String typename =
        "patterns." + packageName + "." + info.type;
      return (T)constructors.computeIfAbsent(
        typename, this::findConstructor
      ).newInstance(info.data);
    } catch(Exception e) {
      throw new RuntimeException(
        "Cannot create() Trash: " + info, e);
    }
  }
  private
  Constructor findConstructor(String typename) {
    try {
      System.out.println("Loading " + typename);
      return Class.forName(typename)
        .getConstructor(double.class);
    } catch(Exception e) {
      throw new RuntimeException(
        "Trash(double) Constructor Not Found: " +
        typename, e);
    }
  }
}
```

`constructors` maps a `String` (the name of the subtype of `Trash`) to its corresponding `Constructor`. The name for the specific subclass of `Trash` comes from the `TrashInfo` `type` property and is used to construct the `typename`. `typename` is used to look up the `Constructor` in `constructors`. If there is no entry for `typename`, `computeIfAbsent()` calls `findConstructor()` to produce the `Constructor` corresponding to that `typename`, which is then added to `constructors`. Finally, `newInstance(info.data)` calls the `Constructor`, passing it the weight.

There are many ways you could determine which object to make—the `info.type` approach is used so text information read from a file can be turned into objects.

`DynaFactory` assumes that every `Trash` type has a constructor that takes a `double` (notice that `double.class` is distinct from `Double.class`). It's also possible, for a more flexible solution, to call `getConstructors()`, which returns an array of possible constructors.

As you'll see, the beauty of this design is that the code doesn't change, regardless of the different situations where it's used.

Parsing `Trash` from a File

Next, instead of randomly generating `Trash` objects, we'd like to create them by reading information from a text file. This file contains all necessary information about each piece of trash, in the format `Trash:weight`:

```
// patterns/trash/Trash.dat
Cardboard:4.4
Paper:8.0
Aluminum:1.8
Glass:5.4
Aluminum:3.4
Cardboard:2.2
Glass:4.3
Cardboard:1.2
Paper:6.6
Aluminum:2.7
Paper:9.1
Glass:3.6
```

`ParseTrash.fillBin()` opens `Trash.dat`, creates a piece of `Trash` for each line, and adds it to a `Fillable`:

```
// patterns/trash/ParseTrash.java
// Opens a file and parses its contents into
// Trash objects, placing each into a Fillable.
// {java patterns.trash.ParseTrash}
package patterns.trash;
import java.util.*;
import java.util.stream.*;
import java.io.*;
import java.nio.file.*;
import java.nio.file.Files;

public class ParseTrash {
  public static String source = "Trash.dat";
  public static <T extends Trash> void
  fillBin(String packageName, Fillable<T> bin) {
    DynaFactory factory =
      new DynaFactory(packageName);
    try {
      Files.lines(Paths.get("trash", source))
        // Remove comments and empty lines:
        .filter(line -> line.trim().length() != 0)
        .filter(line -> !line.startsWith("//"))
        .forEach(line -> {
          String type = line.substring(
              0, line.indexOf(':')).trim();
          double weight = Double.valueOf(
            line.substring(line.indexOf(':') + 1)
            .trim());
          bin.addTrash(factory.create(
            new TrashInfo(type, weight)));
        });
    } catch(IOException | NumberFormatException e) {
      throw new RuntimeException(e);
    }
  }
  // Special case to handle a List:
  public static <T extends Trash> void
  fillBin(String packageName, List<T> bin) {
    fillBin(packageName, new FillableList<>(bin));
  }
  // Basic test:
  public static void main(String[] args) {
```

```
    List<Trash> bin = new ArrayList<>();
    fillBin("trash", bin);
    bin.forEach(System.out::println);
  }
}
/* Output:
Loading patterns.trash.Cardboard
Loading patterns.trash.Paper
Loading patterns.trash.Aluminum
Loading patterns.trash.Glass
Cardboard weight: 4.40 * price: 0.11 = 0.48
Paper weight: 8.00 * price: 0.10 = 0.80
Aluminum weight: 1.80 * price: 1.67 = 3.01
Glass weight: 5.40 * price: 0.23 = 1.24
Aluminum weight: 3.40 * price: 1.67 = 5.68
Cardboard weight: 2.20 * price: 0.11 = 0.24
Glass weight: 4.30 * price: 0.23 = 0.99
Cardboard weight: 1.20 * price: 0.11 = 0.13
Paper weight: 6.60 * price: 0.10 = 0.66
Aluminum weight: 2.70 * price: 1.67 = 4.51
Paper weight: 9.10 * price: 0.10 = 0.91
Glass weight: 3.60 * price: 0.23 = 0.83
*/
```

The file path is relative to the parent directory (`patterns`) because related examples will also be packaged off of that directory. `Trash.dat` is turned into a `Stream` of lines which are filtered to remove comments and empty lines. Then the `String` method `indexOf()` produces the index of the `:`. The `String` method `substring()` extracts the name of the trash type. `substring()` is then used to find the `weight`, which is turned into a `double` using `Double.valueOf()`. The `trim()` method removes white space at both ends of a `String`.

In `RecycleA.java`, an `ArrayList` holds the `Trash` objects. To allow other types of collections to be used, `fillBin()` takes a reference to a `Fillable`, which is simply an interface that supports a method called `addTrash()`:

```
// patterns/trash/Fillable.java
// Any object that can be filled with Trash.
package patterns.trash;

public interface Fillable<T extends Trash> {
  void addTrash(T t);
}
```

Anything that supports this interface can be used with `fillBin()`. However, `List` doesn't implement `Fillable`, so it won't work without some help. `List` is used in most of the examples so it makes sense to add a second overloaded `fillBin()` method that takes a `List<Trash>`. The `List<Trash>` can be treated as a `Fillable` by using an *Adapter* class:

```
// patterns/trash/FillableList.java
// Adapter that makes a List Fillable.
package patterns.trash;
import java.util.*;

public class FillableList<T extends Trash>
implements Fillable<T> {
  private List<T> list;
  public FillableList(List<T> list) {
    this.list = list;
  }
  @Override public void addTrash(T t) {
    list.add(t);
  }
}
```

The only job of this class is to connect `Fillable`'s `addTrash()` method to `List`'s `add()`. With this class in hand, the overloaded `fillBin()` method can be used with a `List` in `ParseTrash.java`.

This approach works for any frequently-used collection class. Alternatively, the collection class can provide its own adapter that implements `Fillable`. (You'll see this later, in `TypeMapTrash.java`.)

Recycling with `DynaFactory`

Here's the revised version of `RecycleA.java` using `ParseTrash.fillBin()`, which in turn uses a `DynaFactory` for `Trash` creation:

```
// patterns/recycleb/RecycleB.java
// {java patterns.recycleb.RecycleB}
package patterns.recycleb;
import patterns.trash.*;
import java.util.*;

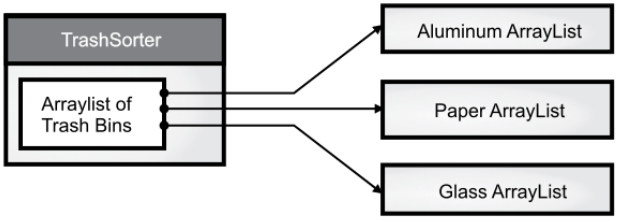
public class RecycleB {
  public static void main(String[] args) {
    List<Trash> bin = new ArrayList<>();
    ParseTrash.fillBin("trash", bin);
    Bins bins = new Bins(bin);
    bins.show();
  }
}
/* Output:
Loading patterns.trash.Cardboard
Loading patterns.trash.Paper
Loading patterns.trash.Aluminum
Loading patterns.trash.Glass
Aluminum weight: 1.80 * price: 1.67 = 3.01
Aluminum weight: 3.40 * price: 1.67 = 5.68
Aluminum weight: 2.70 * price: 1.67 = 4.51
Total Aluminum value = 13.19
Paper weight: 8.00 * price: 0.10 = 0.80
Paper weight: 6.60 * price: 0.10 = 0.66
Paper weight: 9.10 * price: 0.10 = 0.91
Total Paper value = 2.37
Glass weight: 5.40 * price: 0.23 = 1.24
Glass weight: 4.30 * price: 0.23 = 0.99
Glass weight: 3.60 * price: 0.23 = 0.83
Total Glass value = 3.06
Cardboard weight: 4.40 * price: 0.11 = 0.48
Cardboard weight: 2.20 * price: 0.11 = 0.24
Cardboard weight: 1.20 * price: 0.11 = 0.13
Total Cardboard value = 0.86
Cardboard weight: 4.40 * price: 0.11 = 0.48
Paper weight: 8.00 * price: 0.10 = 0.80
Aluminum weight: 1.80 * price: 1.67 = 3.01
Glass weight: 5.40 * price: 0.23 = 1.24
Aluminum weight: 3.40 * price: 1.67 = 5.68
Cardboard weight: 2.20 * price: 0.11 = 0.24
Glass weight: 4.30 * price: 0.23 = 0.99
```

```
Cardboard weight: 1.20 * price: 0.11 = 0.13
Paper weight: 6.60 * price: 0.10 = 0.66
Aluminum weight: 2.70 * price: 1.67 = 4.51
Paper weight: 9.10 * price: 0.10 = 0.91
Glass weight: 3.60 * price: 0.23 = 0.83
Total Trash value = 19.48
*/
```

The process of parsing `Trash.dat` is wrapped into `ParseTrash.fillBin()`, so it's no longer a part of our design focus. Throughout the rest of the chapter, no matter what new classes are added, `ParseTrash.fillBin()` works without change.

Abstracting Usage

With creation out of the way, we can tackle the remainder of the design: where the classes are used. It's the act of sorting into bins that's particularly ugly and exposed so let's apply the principle of *hide ugliness inside methods or classes*:



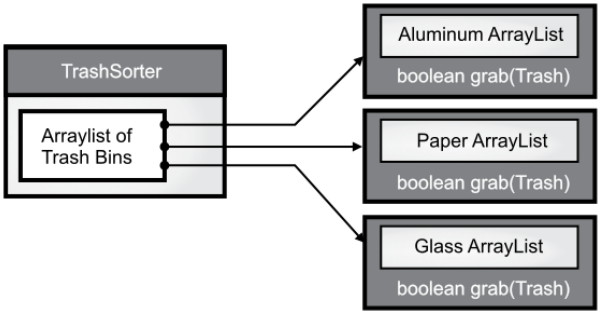
The `TrashSorter` class might be implemented like this:

```
class TrashSorter extends ArrayList<ArrayList<Trash>> {
  void sort(Trash t) { /* ... */ }
}
```

That is, `TrashSorter` is a `List` of `Lists` of `Trash`, and with `List`'s `add()` method you can install additional `List<Trash>` objects. Thus, we must change the way the `TrashSorter` is initialized whenever a new type of `Trash` is added to the model.

Now, however, `sort()` becomes a problem. How does the statically-coded method deal with the fact that a new type was added? To solve this, the type information must be removed from `sort()`. The new version of `sort()` calls a general-purpose method

that handles type details. This is another way to describe a dynamically-bound method. So `sort()` will simply move through the sequence and call a dynamically-bound method for each `List<Trash>`. The job of this method is to grab the pieces of trash it is interested in, so its name is `grab(Trash)`. The structure now looks like:



`TrashSorter` calls each `grab()` method and gets a different result depending on what type of `Trash` the current `List<Trash>` is holding. That is, each `List` must be aware of the type it holds. The classic approach to this problem is to create a base `TrashBin` class and inherit a new derived class for each different type to hold. Further observation can produce a better approach.

A basic OOP design principle is:

> *Use data members for variation in state, use polymorphism for variation in behavior.*

Your first thought might be that the `grab()` method certainly behaves differently for a `List<Paper>` and a `List<Glass>`. But what it does is strictly dependent on the type, and nothing else. This could be interpreted as a different state. In Java, `Class` is the runtime representation for a type, and this can be used to establish the type of `Trash` a particular `TrashBin` will hold.

You provide the `TrashBin` constructor with the `Class` of your choice. This tells the `List` what type it is supposed to hold. Then the `grab()` method uses `binType` and reflection to see if the `Trash` object you've handed it matches the type it's supposed to grab:

```
// patterns/recyclec/RecycleC.java
// {java patterns.recyclec.RecycleC}
package patterns.recyclec;
import patterns.trash.*;
import java.util.*;

// A List that only admits the right type:
class
TrashBin<T extends Trash> extends ArrayList<T> {
  final Class<T> binType;
  TrashBin(Class<T> binType) {
    this.binType = binType;
  }
  @SuppressWarnings("unchecked")
  boolean grab(Trash t) {
    // Compare class types:
    if(t.getClass().equals(binType)) {
      add((T)t); // Downcast to this TrashBin type
      return true; // Trash grabbed
    }
    return false; // Trash not grabbed
  }
}

class TrashBinList<T extends Trash>
extends ArrayList<TrashBin<? extends T>> {    // [1]
  @SuppressWarnings("unchecked")
  public TrashBinList(Class<? extends T>... types) {
    for(Class<? extends T> type : types)
      add(new TrashBin<>(type));
  }
  public boolean sort(T t) {
    for(TrashBin<? extends T> ts : this)
      if(ts.grab(t))
        return true;
    return false; // TrashBin not found for t
  }
  public void sortBin(TrashBin<T> bin) {      // [2]
    for(T trash : bin)
      if(!sort(trash))
        throw new RuntimeException(
          "Bin not found for " + trash);
```

```
  }
  public void show() {
    for(TrashBin<? extends T> tbin : this) {
      String typeName = tbin.binType.getSimpleName();
      TrashValue.sum(tbin, typeName);
    }
  }
}

public class RecycleC {
  public static void main(String[] args) {
    TrashBin<Trash> bin =
      new TrashBin<>(Trash.class);
    ParseTrash.fillBin("trash", bin);
    @SuppressWarnings("unchecked")
    TrashBinList<Trash> trashBins =
      new TrashBinList<>(
        Aluminum.class, Paper.class, Glass.class,
        // Add one item:
        Cardboard.class                       // [3]
      );
    trashBins.sortBin(bin);                   // [4]
    trashBins.show();
    TrashValue.sum(bin, "Trash");
  }
}
/* Output:
Loading patterns.trash.Cardboard
Loading patterns.trash.Paper
Loading patterns.trash.Aluminum
Loading patterns.trash.Glass
Aluminum weight: 1.80 * price: 1.67 = 3.01
Aluminum weight: 3.40 * price: 1.67 = 5.68
Aluminum weight: 2.70 * price: 1.67 = 4.51
Total Aluminum value = 13.19
Paper weight: 8.00 * price: 0.10 = 0.80
Paper weight: 6.60 * price: 0.10 = 0.66
Paper weight: 9.10 * price: 0.10 = 0.91
Total Paper value = 2.37
Glass weight: 5.40 * price: 0.23 = 1.24
Glass weight: 4.30 * price: 0.23 = 0.99
Glass weight: 3.60 * price: 0.23 = 0.83
```

```
Total Glass value = 3.06
Cardboard weight: 4.40 * price: 0.11 = 0.48
Cardboard weight: 2.20 * price: 0.11 = 0.24
Cardboard weight: 1.20 * price: 0.11 = 0.13
Total Cardboard value = 0.86
Cardboard weight: 4.40 * price: 0.11 = 0.48
Paper weight: 8.00 * price: 0.10 = 0.80
Aluminum weight: 1.80 * price: 1.67 = 3.01
Glass weight: 5.40 * price: 0.23 = 1.24
Aluminum weight: 3.40 * price: 1.67 = 5.68
Cardboard weight: 2.20 * price: 0.11 = 0.24
Glass weight: 4.30 * price: 0.23 = 0.99
Cardboard weight: 1.20 * price: 0.11 = 0.13
Paper weight: 6.60 * price: 0.10 = 0.66
Aluminum weight: 2.70 * price: 1.67 = 4.51
Paper weight: 9.10 * price: 0.10 = 0.91
Glass weight: 3.60 * price: 0.23 = 0.83
Total Trash value = 19.48
*/
```

- **[1]** `TrashBinList` contains a `List` of `TrashBin` references, so `sort()` can iterate through the `TrashBins` when it's looking for a match for each `Trash` object.
- **[2]** `sortBin()` picks out each piece of `Trash` from its `TrashBin` argument, sorting it into the appropriate specific `TrashBin`. Notice that this code doesn't change at all if new types are added. If the bulk of your code doesn't need changing when a new type is added (or when some other change occurs) then you have an easily-extensible system.
- **[3]** Adding this one line adds the `Cardboard` type.
- **[4]** One method call causes the contents of `bin` to be sorted into the respective specifically-typed bins.

Redesign with Multiple Dispatching

The above design is certainly satisfactory. Adding new types consists of adding or modifying distinct classes without producing additional code changes in the system. Reflection is not "misused" as it was in `RecycleA.java`. However, it's possible to go one step further—we can take a purist viewpoint about reflection and say it should be eliminated altogether from the operation of sorting trash into bins.

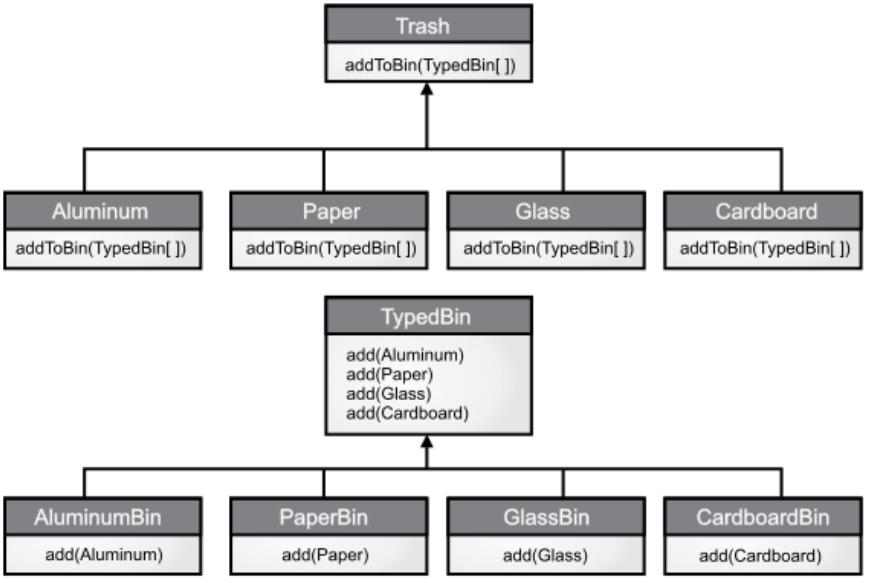
To accomplish this, you must first take the perspective that all type-dependent activities—such as detecting the type of a piece of trash and putting it into the appropriate bin—should be controlled through polymorphism.

The previous examples first sorted by type, then acted on sequences of elements that were all of a particular type. Whenever you find yourself picking out particular types, stop and think. The whole idea of polymorphism (dynamically-bound method calls) is to handle type-specific behavior for you. So why are you hunting for types?

You hunt for types because Java only performs single dispatching. That is, if an operation acts on more than one object whose type is unknown, Java invokes the dynamic binding mechanism on only one of those types. This doesn't solve the problem, so you end up detecting some types manually and effectively producing your own dynamic binding behavior.

The solution is *Multiple Dispatching*. We configure our code such that a single method call produces more than one dynamic method call, and thus determines more than one type. To get this effect, you typically work with multiple type hierarchies: one hierarchy for each dispatch. The following example works with two hierarchies: the existing `Trash` family and a hierarchy of the types of trash bins that the trash is placed into. When there are only two dispatches it's called *Double Dispatching*.

To implement the double dispatch, we add a new method in the `Trash` hierarchy called `addToBin()`, which takes an argument of `List<TypedBin>`. A piece of `Trash` calls `addToBin()` to step through this `List` and add itself to the appropriate `TypedBin` using double dispatching.



The new hierarchy is `TypedBin`, and it contains an `add()` method that is also used polymorphically. But here's an additional twist: `add()` is *overloaded* to take arguments of the different types of `Trash`. So an essential part of the double-dispatching scheme also involves overloading.

Redesigning the program produces a dilemma: it's now necessary for the `Trash` classes to contain an `addToBin()` method. One approach is to create a new hierarchy by copying the code from the existing `Trash` hierarchy and incorporating `addToBin()`. Another approach, which you can take when you don't have control of the source code, or you don't want to risk introducing new bugs into existing code, is to put `addToBin()` into an interface, leave `Trash` alone, and inherit new specific types of `Aluminum`, `Paper`, `Glass`, and `Cardboard`. Here is the interface:

```
// patterns/doubledispatch/TypedBinMember.java
// Adapts the double-dispatching
// method into the trash hierarchy without
// modifying the original hierarchy.
package patterns.doubledispatch;
import java.util.*;

public interface TypedBinMember {
  boolean addToBin(List<TypedBin> bins);
}
```

`addToBin()` is implemented in each adapted subtype of `Aluminum`, `Paper`, `Glass`, and `Cardboard`, but it *looks* like the code for `addToBin()` is identical in each case:

```
// patterns/doubledispatch/Aluminum.java
// Aluminum with double dispatching.
package patterns.doubledispatch;
import patterns.trash.*;
import java.util.*;

public class Aluminum extends patterns.trash.Aluminum
implements TypedBinMember {
  public Aluminum(double wt) { super(wt); }
  @Override
  public boolean addToBin(List<TypedBin> tbins) {
    return tbins.stream()
      .anyMatch(tb -> tb.add(this));
  }
}

// patterns/doubledispatch/Paper.java
// Paper with double dispatching.
package patterns.doubledispatch;
import patterns.trash.*;
import java.util.*;

public class Paper extends patterns.trash.Paper
implements TypedBinMember {
  public Paper(double wt) { super(wt); }
  @Override
  public boolean addToBin(List<TypedBin> tbins) {
    return tbins.stream()
```

```
      .anyMatch(tb -> tb.add(this));
  }
}

// patterns/doubledispatch/Glass.java
// Glass with double dispatching.
package patterns.doubledispatch;
import patterns.trash.*;
import java.util.*;

public class Glass extends patterns.trash.Glass
implements TypedBinMember {
  public Glass(double wt) { super(wt); }
  @Override
  public boolean addToBin(List<TypedBin> tbins) {
    return tbins.stream()
      .anyMatch(tb -> tb.add(this));
  }
}

// patterns/doubledispatch/Cardboard.java
// Cardboard with double dispatching.
package patterns.doubledispatch;
import patterns.trash.*;
import java.util.*;

public class Cardboard extends patterns.trash.Cardboard
implements TypedBinMember {
  public Cardboard(double wt) { super(wt); }
  @Override
  public boolean addToBin(List<TypedBin> tbins) {
    return tbins.stream()
      .anyMatch(tb -> tb.add(this));
  }
}
```

`anyMatch(tb -> tb.add(this))` calls `add()` for each `TypedBin` object in the array. `add()` returns a `boolean` indicating whether the `Trash` was successfully added to that particular `TypedBin`, and `anyMatch()` discovers whether *any* of those `add()`s were successful. If it was, then this piece of `Trash` was sorted into a bin.

The code for each `addToBin()` appears to be identical. But notice the `this` argument in `tb.add(this)`. The type of `this` is different for each subclass of `Trash`, so the code is actually different. This is the first part of the double dispatch, and inside `addToBin()` you know you have `Trash` that's specifically `Aluminum`, or `Paper`, etc. During the call to `add()`, this information is passed via the type of `this`. The compiler resolves the call to the proper overloaded version of `add()`. But `tb` produces a reference to the base type `TypedBin` so this dynamic call will end up calling a different method depending on the type of `TypedBin` that's currently selected. This is the second dispatch.

Here's the base class for `TypedBin`. Notice that the `private` `typedBin` field is never directly exposed—the `ArrayList` is copied when you call `bin()`. This way, the `typedBin` `Arraylist` can't be modified by external code:

```
// patterns/doubledispatch/TypedBin.java
// A List that can grab the right type.
package patterns.doubledispatch;
import patterns.trash.*;
import java.util.*;

public class TypedBin {
  private List<Trash> typedBin =
    new ArrayList<>();
  public final String type;
  public TypedBin(String type) {
    this.type = type;
  }
  public List<Trash> bin() {
    // Returns a copy of typedBin:
    return new ArrayList<Trash>(typedBin);
  }
  protected boolean addIt(Trash t) {
    typedBin.add(t);
    return true;
  }
  public boolean add(Aluminum a) {
    return false;
  }
  public boolean add(Paper a) {
    return false;
  }
```

```
  public boolean add(Glass a) {
    return false;
  }
  public boolean add(Cardboard a) {
    return false;
  }
}
```

The overloaded `add()` methods all return `false`. If an `add()` method is not overridden in a derived class, it continues to return `false`, and the caller (`addToBin()`) assumes that the current `Trash` object has not been added to a collection.

Each subclass of `TypedBin` only overrides one of the `add()` methods, appropriate to the type of `TypedBin` being defined. For example, `CardboardBin` overrides `add(Cardboard)`. The overridden method adds the `Trash` object to its collection and returns `true`, while all the rest of the `add()` methods in `CardboardBin` continue to return `false`, as they haven't been overridden.

Here's the rest of the program:

```
// patterns/doubledispatch/DoubleDispatch.java
// Using multiple dispatching to handle more
// than one unknown type during a method call.
// {java patterns.doubledispatch.DoubleDispatch}
package patterns.doubledispatch;
import patterns.trash.*;
import java.util.*;

class AluminumBin extends TypedBin {
  public AluminumBin() { super("Aluminum"); }
  @Override public boolean add(Aluminum a) {
    return addIt(a);
  }
}

class PaperBin extends TypedBin {
  public PaperBin() { super("Paper"); }
  @Override public boolean add(Paper a) {
    return addIt(a);
  }
}
```

```
class GlassBin extends TypedBin {
  public GlassBin() { super("Glass"); }
  @Override public boolean add(Glass a) {
    return addIt(a);
  }
}

class CardboardBin extends TypedBin {
  public CardboardBin() { super("Cardboard"); }
  @Override public boolean add(Cardboard a) {
    return addIt(a);
  }
}

class TrashBinSet {
  public final List<TypedBin> binSet =
    Arrays.asList(
      new AluminumBin(), new PaperBin(),
      new GlassBin(), new CardboardBin()
    );
  @SuppressWarnings("unchecked")
  public void sortIntoBins(List bin) {
    bin.forEach( aBin -> {
      TypedBinMember t = (TypedBinMember)aBin;
      if(!t.addToBin(binSet))
        throw new RuntimeException(
          "sortIntoBins() couldn't add " + t);
    });
  }
}

public class DoubleDispatch {
  public static void main(String[] args) {
    List<Trash> bin = new ArrayList<>();
    ParseTrash.fillBin("doubledispatch", bin);
    TrashBinSet bins = new TrashBinSet();
    // Sort from the master bin into the
    // individually-typed bins:
    bins.sortIntoBins(bin);
    // Sum value of each bin...
    bins.binSet.forEach(tb ->
      TrashValue.sum(tb.bin(), tb.type));
```

```
    // ... and for the master bin:
    TrashValue.sum(bin, "Trash");
  }
}
/* Output:
Loading patterns.doubledispatch.Cardboard
Loading patterns.doubledispatch.Paper
Loading patterns.doubledispatch.Aluminum
Loading patterns.doubledispatch.Glass
Aluminum weight: 1.80 * price: 1.67 = 3.01
Aluminum weight: 3.40 * price: 1.67 = 5.68
Aluminum weight: 2.70 * price: 1.67 = 4.51
Total Aluminum value = 13.19
Paper weight: 8.00 * price: 0.10 = 0.80
Paper weight: 6.60 * price: 0.10 = 0.66
Paper weight: 9.10 * price: 0.10 = 0.91
Total Paper value = 2.37
Glass weight: 5.40 * price: 0.23 = 1.24
Glass weight: 4.30 * price: 0.23 = 0.99
Glass weight: 3.60 * price: 0.23 = 0.83
Total Glass value = 3.06
Cardboard weight: 4.40 * price: 0.11 = 0.48
Cardboard weight: 2.20 * price: 0.11 = 0.24
Cardboard weight: 1.20 * price: 0.11 = 0.13
Total Cardboard value = 0.86
Cardboard weight: 4.40 * price: 0.11 = 0.48
Paper weight: 8.00 * price: 0.10 = 0.80
Aluminum weight: 1.80 * price: 1.67 = 3.01
Glass weight: 5.40 * price: 0.23 = 1.24
Aluminum weight: 3.40 * price: 1.67 = 5.68
Cardboard weight: 2.20 * price: 0.11 = 0.24
Glass weight: 4.30 * price: 0.23 = 0.99
Cardboard weight: 1.20 * price: 0.11 = 0.13
Paper weight: 6.60 * price: 0.10 = 0.66
Aluminum weight: 2.70 * price: 1.67 = 4.51
Paper weight: 9.10 * price: 0.10 = 0.91
Glass weight: 3.60 * price: 0.23 = 0.83
Total Trash value = 19.48
*/
```

`TrashBinSet` encapsulates all different types of `TypedBins`, along with the `sortIntoBins()` method, which starts the double dispatch. Once the structure is set up,

sorting into the different `TypedBins` is straightforward.

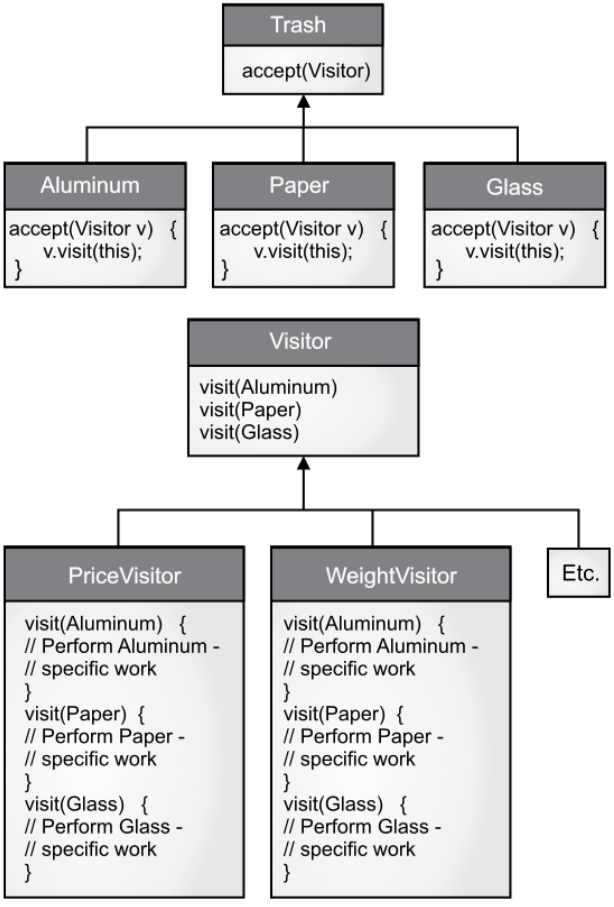
The changes necessary to add a new type are relatively isolated: you inherit the new type of `Trash` with its `addToBin()` method, then you inherit a new `TypedBin`, and finally you add a new type into the initialization for `TrashBinSet.binSet`.

The *Visitor* Pattern

Now consider applying a design pattern with an entirely different goal.

For this pattern, we are no longer concerned with optimizing the addition of new types of `Trash` to the system. In some ways, *Visitor* makes adding a new type of `Trash` *more* complicated. The assumption is that you have a primary class hierarchy that is frozen—perhaps it's from a vendor. Because it's frozen, you can't make changes. However, you'd like to add new polymorphic methods to that hierarchy. Normally you'd add methods to the base class, but you can't touch anything in the hierarchy. How do you get around this?

The *Visitor* pattern builds on *Double Dispatching*. *Visitor* allows you to extend the interface of the primary type by creating a separate class hierarchy of type `Visitor` to virtualize the operations performed upon the primary type. The objects of the primary type simply "accept" the visitor, then call the visitor's dynamically-bound method. It looks like this:



If `v` is a reference to an `Aluminum` object, the code:

```
PriceVisitor pv = new PriceVisitor();
v.accept(pv);
```

causes two polymorphic method calls: the first one to select `Aluminum`'s version of `accept()`, and the second one within `accept()`, when the specific version of `visit()` is called dynamically using the `Visitor` reference `pv`.

This configuration means new functionality can be added to the system in the form of new subclasses of `Visitor`. The `Trash` hierarchy doesn't need modification. This is the prime benefit of the *Visitor* pattern: you can add new polymorphic functionality to a class hierarchy without touching that hierarchy.

There's a caveat: the provider of the frozen hierarchy needs to anticipate the use of the *Visitor* pattern, either by explicitly including the `accept()` method (as was done in the `Trash` hierarchy) or by making the classes in the hierarchy accessible enough that you can overlay an interface containing `accept()`, similar to the double dispatching example.

Visitor is helpful, but it's not what we set out to accomplish, so you might decide this isn't the desired solution. But look at one thing it achieves: the *Visitor* solution avoids sorting from the master `Trash` sequence into individual typed containers. Thus, you can leave everything in the single master sequence and simply pass through it using an appropriate visitor to accomplish your goal.

The double dispatching in the *Visitor* pattern determines both the type of `Trash` and the type of `Visitor` without using reflection. In the following example, there are two implementations of `Visitor`: `PriceVisitor` to display and sum the price, and `WeightVisitor` to display and sum the weight.

Now you understand the inclusion of the `accept()` method in the `Trash` hierarchy (which you were previously told to ignore). Look back at the classes representing the various types of `Trash`: each `accept()` is simply `v.visit(this);`. Just as you saw in the double dispatching example, the `accept()` methods look identical, but each is different because the type of `this` is different.

`Visitor` is an `abstract class` rather than an `interface` because it contains `doubles` to hold the totals for the different types of trash, along with common code for `show()` and `total()` to display results. The `descriptor` is set by the derived classes to "price" and "weight" so `show()` produces meaningful output:

```
// patterns/trash/Visitor.java
// The base class for Visitors.
package patterns.trash;

public abstract class Visitor {
  protected double alTotal; // Aluminum
  protected double pTotal;  // Paper
  protected double gTotal;  // Glass
  protected double cTotal;  // Cardboard
  protected String descriptor;
  protected Visitor(String descriptor) {
    this.descriptor = descriptor;
```

```
  }
  protected void show(String type, double value) {
    System.out.printf(
      "%s %s: %.2f%n", type, descriptor, value);
  }
  public void total() {
    show("Total Aluminum", alTotal);
    show("Total Paper", pTotal);
    show("Total Glass", gTotal);
    show("Total Cardboard", cTotal);
  }
  abstract void visit(Aluminum a);
  abstract void visit(Paper p);
  abstract void visit(Glass g);
  abstract void visit(Cardboard c);
}
```

Note that `Visitor` and its subclasses must be modified when adding a new type of `Trash`. This actually isn't too bad, because the changes are isolated to the `Visitor` hierarchy. If you were, for example, to add something like `visit(Plastic p)` to `Visitor`, the compiler ensures that all `Visitor` subclasses implement the new overloaded `visit()`.

The rest of the program creates specific `Visitor` classes and sends them through a single list of `Trash` objects:

```
// patterns/trash/TrashVisitor.java
// Related algorithms packaged in
// each implementation of Visitor.
// {java patterns.trash.TrashVisitor}
package patterns.trash;
import java.util.*;

class PriceVisitor extends Visitor {
  public PriceVisitor() { super("price"); }
  @Override public void visit(Aluminum al) {
    double price = al.weight * al.price();
    show("Aluminum", price);
    alTotal += price;
  }
  @Override public void visit(Paper p) {
    double price = p.weight * p.price();
```

```
    show("Paper", price);
    pTotal += price;
  }
  @Override public void visit(Glass g) {
    double price = g.weight * g.price();
    show("Glass", price);
    gTotal += price;
  }
  @Override public void visit(Cardboard c) {
    double price = c.weight * c.price();
    show("Cardboard", price);
    cTotal += price;
  }
}

class WeightVisitor extends Visitor {
  public WeightVisitor() { super("weight"); }
  @Override public void visit(Aluminum al) {
    show("Aluminum", al.weight);
    alTotal += al.weight;
  }
  @Override public void visit(Paper p) {
    show("Paper", p.weight);
    pTotal += p.weight;
  }
  @Override public void visit(Glass g) {
    show("Glass", g.weight);
    gTotal += g.weight;
  }
  @Override public void visit(Cardboard c) {
    show("Cardboard", c.weight);
    cTotal += c.weight;
  }
}

public class TrashVisitor {
  public static void main(String[] args) {
    List<Trash> bin = new ArrayList<>();
    ParseTrash.fillBin("trash", bin);
    List<Visitor> visitors = Arrays.asList(
      new PriceVisitor(), new WeightVisitor());
    bin.forEach(
```

```
      trash -> visitors.forEach(trash::accept)
    );
    visitors.forEach(Visitor::total);
  }
}
/* Output:
Loading patterns.trash.Cardboard
Loading patterns.trash.Paper
Loading patterns.trash.Aluminum
Loading patterns.trash.Glass
Cardboard price: 0.48
Cardboard weight: 4.40
Paper price: 0.80
Paper weight: 8.00
Aluminum price: 3.01
Aluminum weight: 1.80
Glass price: 1.24
Glass weight: 5.40
Aluminum price: 5.68
Aluminum weight: 3.40
Cardboard price: 0.24
Cardboard weight: 2.20
Glass price: 0.99
Glass weight: 4.30
Cardboard price: 0.13
Cardboard weight: 1.20
Paper price: 0.66
Paper weight: 6.60
Aluminum price: 4.51
Aluminum weight: 2.70
Paper price: 0.91
Paper weight: 9.10
Glass price: 0.83
Glass weight: 3.60
Total Aluminum price: 13.19
Total Paper price: 2.37
Total Glass price: 3.06
Total Cardboard price: 0.86
Total Aluminum weight: 7.90
Total Paper weight: 23.70
Total Glass weight: 13.30
Total Cardboard weight: 7.80
```

```
*/
```

In `main()`, there's only a single `Trash` bin. The two `Visitor` objects are accepted into every element in the sequence. The visitors keep their own internal data to tally the total weights and prices.

In the double-dispatching solution, only one of the overloaded `add()` methods was overridden when each subclass was created. Here, *each* of the overloaded `visit()` methods is overridden in every subclass of `Visitor`.

Reflection Considered Harmful?

Various designs in this chapter attempt to remove reflection, which might give you the impression it is "considered harmful." This isn't true; the problem is the *misuse* of reflection. The reason our designs removed reflection is because the misapplication of that feature prevented extensibility—our stated goal of adding a new type to the system with low impact on surrounding code. Reflection is often misused by looking for every single type in your system. This creates non-extensible code, because when you add a new type, you must hunt for all the code where reflection is used, and if you miss any you won't get help from the compiler.

However, reflection doesn't automatically create non-extensible code. Let's revisit the trash recycler and introduce a new tool, which I call a `TypeMap`, containing a `Map<Class, List<T>>`. The interface is simple: you can `add()` a new object, and `values()` produces a `Stream` of `Lists`, with each `List` containing all objects of a particular type. Whenever you add a new type to the system, the `TypeMap` dynamically adapts.

```
// patterns/TypeMap.java
// Generic TypeMap works for any types.
package patterns;
import java.util.*;
import java.util.stream.*;

public class TypeMap<T> {
  public final Map<Class, List<T>> map =
    new HashMap<>();
  public void add(T o) {
    Class type = o.getClass();
```

```
    map.computeIfAbsent(type,
      k -> new ArrayList<T>()).add(o);
  }
  public Stream<List<T>> values() {
    return map.values().stream();
  }
}
```

The `add()` method does all the work. When you `add()` a new object, its `Class` is extracted. This is used as a key to determine whether a `List` holding objects of that type is already present in the `Map`. If so, that `List` is produced. If not, `computeIfAbsent()` adds a key-value pair consisting of the `Class` object together with a new `ArrayList`. In either case, the object is added to the `List`.

The next two examples `show()` the contents of a `Map<Class, List<Trash>>`, so we'll make that a utility function:

```
// patterns/trash/ClassToListOfTrashMap.java
// Display a Map<Class, List<Trash>>.
package patterns.trash;
import java.util.*;

public class ClassToListOfTrashMap {
  public static void
  show(Map<Class, List<Trash>> map) {
    map.values().forEach( bin -> {
      String typeName = "Trash";
      if(!bin.isEmpty())
        typeName =
          bin.get(0).getClass().getSimpleName();
      TrashValue.sum(bin, typeName);
    });
  }
}
```

`ParseTrash.fillBin()` performs a sort every time it inserts a `Trash` object into `bin`:

```
// patterns/trash/TypeMapTrash.java
// Using a Map of Lists and reflection to sort
// Trash into Lists. This is extensible, despite
// the use of reflection.
// {java patterns.trash.TypeMapTrash}
package patterns.trash;
import patterns.TypeMap;

// Adapter class for ParseTrash.fillBin():
class TypeMapAdapter implements Fillable {
  private TypeMap<Trash> map;
  TypeMapAdapter(TypeMap<Trash> map) {
    this.map = map;
  }
  @Override
  public void addTrash(Trash t) { map.add(t); }
}

public class TypeMapTrash {
  @SuppressWarnings("unchecked")
  public static void main(String[] args) {
    TypeMap<Trash> bin = new TypeMap<>();
    ParseTrash.fillBin(
      "trash", new TypeMapAdapter(bin));
    ClassToListOfTrashMap.show(bin.map);
  }
}
/* Output:
Loading patterns.trash.Cardboard
Loading patterns.trash.Paper
Loading patterns.trash.Aluminum
Loading patterns.trash.Glass
Paper weight: 8.00 * price: 0.10 = 0.80
Paper weight: 6.60 * price: 0.10 = 0.66
Paper weight: 9.10 * price: 0.10 = 0.91
Total Paper value = 2.37
Glass weight: 5.40 * price: 0.23 = 1.24
Glass weight: 4.30 * price: 0.23 = 0.99
Glass weight: 3.60 * price: 0.23 = 0.83
Total Glass value = 3.06
Aluminum weight: 1.80 * price: 1.67 = 3.01
Aluminum weight: 3.40 * price: 1.67 = 5.68
```

```
Aluminum weight: 2.70 * price: 1.67 = 4.51
Total Aluminum value = 13.19
Cardboard weight: 4.40 * price: 0.11 = 0.48
Cardboard weight: 2.20 * price: 0.11 = 0.24
Cardboard weight: 1.20 * price: 0.11 = 0.13
Total Cardboard value = 0.86
*/
```

Much of `class TypeMapTrash` should be familiar from previous examples. This time, instead of placing the new `Trash` objects into a `bin` of type `List`, the `bin` is of type `TypeMap`. When the trash is thrown into `bin` it's immediately sorted by `TypeMap`'s internal sorting mechanism. After that, operating on each individual `List` becomes a simple matter.

Adding a new type to the system won't affect this code at all, nor the code in `TypeMap`. This solution does rely heavily on reflection, but notice that each key-value pair in the `Map` is looking for only one type. In addition, there's no way you can forget to add or modify code when you add a new type because there's nothing you need to change.

Java 8 provides the `Stream` function `groupingBy()` which can also yield a `Map<Class, List<Trash>>`:

```
// patterns/trash/GroupingBy.java
// {java patterns.trash.GroupingBy}
package patterns.trash;
import java.util.*;
import java.util.stream.*;

public class GroupingBy {
  public static void main(String[] args) {
    List<Trash> bin = new ArrayList<>();
    ParseTrash.fillBin("trash", bin);
    Map<Class, List<Trash>> m =
      bin.stream().collect(
        Collectors.groupingBy(Object::getClass));
    ClassToListOfTrashMap.show(m);
  }
}
/* Output:
Loading patterns.trash.Cardboard
```

```
Loading patterns.trash.Paper
Loading patterns.trash.Aluminum
Loading patterns.trash.Glass
Glass weight: 5.40 * price: 0.23 = 1.24
Glass weight: 4.30 * price: 0.23 = 0.99
Glass weight: 3.60 * price: 0.23 = 0.83
Total Glass value = 3.06
Paper weight: 8.00 * price: 0.10 = 0.80
Paper weight: 6.60 * price: 0.10 = 0.66
Paper weight: 9.10 * price: 0.10 = 0.91
Total Paper value = 2.37
Cardboard weight: 4.40 * price: 0.11 = 0.48
Cardboard weight: 2.20 * price: 0.11 = 0.24
Cardboard weight: 1.20 * price: 0.11 = 0.13
Total Cardboard value = 0.86
Aluminum weight: 1.80 * price: 1.67 = 3.01
Aluminum weight: 3.40 * price: 1.67 = 5.68
Aluminum weight: 2.70 * price: 1.67 = 4.51
Total Aluminum value = 13.19
*/
```

This produces the same effect as `TypeMapTrash`, but without the need to define the `TypeMap` or `TypeMapAdapter` classes. Also, there is no need to suppress "unchecked" warnings.

To help internalize the ideas in the last part of this chapter (starting at Pattern Refactoring), try adding a subtype of `Trash` called `Plastic`, and see how it incorporates into all the examples.

Summary

Discovering the vector of change is no trivial matter. It's not something an analyst can usually detect before the program sees its initial design. The necessary insight probably won't appear until later phases in the project. Sometimes only at the implementation phase do you discover a deeper or more subtle need in your system. In the case of adding new types (as explored in the latter part of this chapter) you might only realize you need (for example) a particular inheritance hierarchy when you are in the maintenance phase and you begin extending the system.

Design patterns say that OOP isn't just about polymorphism. It's about "separating things that change from things that stay the same." Polymorphism is an important way to do this, and it turns out to be helpful if the programming language directly supports polymorphism (so you don't have to wire it in yourself, which tends to make it prohibitively expensive). But design patterns in general show *other* ways to accomplish the basic goal, and once your eyes open to this you begin searching for more creative designs.

Appendix: Supplements

There are several supplements to this book, available at **OnJava8.com/supplements**.

On Java 8 Example Code

The code for this book is freely available for download here[184]. This includes the Gradle build files and other support files necessary for a successful build and execution of all the examples in the book.

Thinking in C: Foundations for Java

The free Thinking in C[185] seminar, created by Chuck Allison and developed by MindView LLC, is an eSeminar which gives you an introduction to the C syntax, operators and functions that Java syntax is based upon.

Go to **OnJava8.com/supplements** to find the direct link for the download.

Hands-On Java eSeminar

The *Hands-On Java eSeminar* is based on the 2nd edition of *Thinking in Java*. There is an audio lecture and slides corresponding to every chapter in that book. I created the seminar and I narrate the material on the eSeminar. The eSeminar uses HTML5, so it should run on most modern browsers. There are trial demos of the product. The *Hands-On Java eSeminar* is for sale; go to **OnJava8.com/supplements** to find the direct link.

[184] https://github.com/BruceEckel/Onjava8-examples/
[185] https://archive.org/details/ThinkingInC

Appendix: Programming Guidelines

> This appendix contains suggestions to help guide you in low-level program design and writing code.

Naturally, these are guidelines and not rules. The idea is to use them as inspirations and to remember there are occasional situations where they should be bent or broken.

Design

1. **Elegance always pays off**. In the short term it might seem like it takes much longer to come up with a truly graceful solution to a problem, but when it works the first time and easily adapts to new situations instead of requiring hours, days, or months of struggle, you'll see the rewards (even if no one can measure them). Not only does it give you a program that's easier to build and debug, but it's also easier to understand and maintain, and that's where the financial value lies. This point can take some experience to understand, because it can appear that you're not productive while you're making a piece of code elegant. Resist the urge to hurry; it will only slow you down.
2. **First make it work, then make it fast**. This is true even if you are certain a piece of code is really important and it is a principal bottleneck in your system. Don't do it. Get the system going first with as simple a design as possible. Then if it isn't going fast enough, profile it. You'll almost always discover that "your" bottleneck isn't the problem. Save your time for the really important stuff.
3. **Remember the "divide and conquer" principle**. If the problem you're looking at is too confusing, imagine the basic operation of the program, given the existence of a magic "piece" that handles the hard parts. That "piece" is an object—write the code that uses the object, then look at the object and encapsulate *its* hard parts into other objects, etc.

4. **Separate the class creator from the class user (*client programmer*)**. The class user is the "customer" and doesn't need or want to know what's going on behind the scenes of the class. The class creator must be the expert in class design and write the class so it can be used by the most novice programmer possible, yet still work robustly in the application. Think of the class as a *service provider* for other classes. Library use is easy only if it's transparent.
5. **When you create a class, attempt to make your names so clear that comments are unnecessary**. Your goal should be to make the client programmer's interface conceptually simple. To this end, use method overloading when appropriate to create an intuitive, easy-to-use interface.
6. **Your analysis and design must produce, at minimum, the classes in your system, their public interfaces, and their relationships to other classes, especially base classes**. If your design method produces more than that, ask yourself if all the pieces produced by that method have value over the lifetime of the program. If they do not, maintaining them will cost you. Members of development teams tend not to maintain anything that does not contribute to their productivity; this is a fact of life that many design methods don't consider.
7. **Automate everything**. Write the test code first (before you write the class, if you can), and keep it with the class. Automate the running of your tests through a build tool—you'll probably use Gradle, the defacto standard Java build tool. This way, any changes are automatically verified by running the test code, and you'll immediately discover errors. Because you know you have the safety net of your test framework, you can be bolder about making sweeping changes when you discover the need. Remember that great improvements in languages come from the built-in testing provided by type checking, exception handling, etc., but those features take you only so far. You must go the rest of the way to create a robust system by filling in the tests that verify features specific to your class or program.
8. **Write the test code first (before you write the class) to verify that your class design is complete**. If you can't write test code, you don't know what your class looks like. In addition, the act of writing the test code often flushes out additional features or constraints you need in the class—these features or constraints don't always appear during analysis and design. Tests also provide example code showing how your class can be used.
9. **All software design problems can be simplified by introducing an extra**

level of conceptual indirection. This fundamental rule of software engineering[186] is the basis of abstraction, the primary feature of object-oriented programming. In OOP, we could also say this as: "If your code is too complicated, make more objects."

10. **An indirection should have a meaning** (in concert with guideline 9). This meaning can be something as simple as "putting commonly used code in a single method." If you add levels of indirection (abstraction, encapsulation, etc.) that don't have meaning, it can be as bad as not having adequate indirection.
11. **Make classes as atomic as possible**. Give each class a single, clear purpose—a cohesive service it provides to other classes. If your classes or your system design grows too complicated, break complex classes into simpler ones. The most obvious indicator of this is sheer size; if a class is big, chances are it's doing too much and should be broken up. Clues to suggest redesign of a class are:
    - A complicated switch statement: consider using polymorphism.
    - A large number of methods that cover broadly different types of operations: consider using several classes.
    - A large number of member variables that concern broadly different characteristics: consider using several classes.
    - Other suggestions are found in *Refactoring: Improving the Design of Existing Code* by Martin Fowler (Addison-Wesley 1999).
12. **Watch for long argument lists**. Method calls then become difficult to write, read, and maintain. Instead, try to move the method to a class where it is (more) appropriate, and/or pass objects in as arguments.
13. **Don't repeat yourself**. If a piece of code appears in many methods in derived classes, put that code into a single method in the base class and call it from the derived-class methods. Not only do you save code space, but you enable easy propagation of changes. Sometimes the discovery of this common code will add valuable functionality to your interface. A simpler version of this guideline also occurs without inheritance: If a class has methods that repeat code, factor that code into a common method and call it from the other methods.
14. **Watch for *switch* statements or chained *if-else* clauses**. This can be an indicator of *type-check coding*, which means you are choosing what code to execute based on some kind of type information (the exact type may not be

[186] Explained to me by Andrew Koenig.

obvious at first). You can often replace this kind of code with inheritance and polymorphism; a polymorphic method call will perform the type checking for you and allow for more reliable and easier extensibility.

15. **From a design standpoint**, **look for and separate things that change from things that stay the same**. That is, search for the elements in a system you might want to change without forcing a redesign, then encapsulate those elements in classes.
16. **Don't extend fundamental functionality by subclassing**. If an interface element is essential to a class it should be in the base class, not added during derivation. If you're adding methods during inheritance, consider rethinking the design.
17. **Less is more**. Start with a minimal interface to a class, as small and simple as you need to solve the problem at hand, but don't try to anticipate all the ways your class *might* be used. As the class is used, you'll discover how you must expand the interface. However, once a class is in use, you cannot shrink the interface without breaking client code. If you must add more methods, that's fine; it won't break code. But even if new methods replace the functionality of old ones, leave the existing interface alone (you can combine the functionality in the underlying implementation if you want). If you must expand the interface of an existing method by adding more arguments, create an overloaded method with the new arguments; this way, you won't disturb any calls to the existing method.
18. **Read your classes aloud to make sure they're logical**. Refer to the relationship between a base class and derived class as "is-a" and member objects as "has-a."
19. **When deciding between inheritance and composition**, **ask if you must upcast to the base type**. If not, prefer composition (member objects) to inheritance. This can eliminate the perceived need for multiple base types. If you inherit, users think they should upcast.
20. **Watch for overloading**. A method should not conditionally execute code based on the value of an argument. Here, create two or more overloaded methods instead.
21. **Use exception hierarchies**—preferably derived from specific appropriate classes in the standard Java exception hierarchy. The person catching the exceptions can then write handlers for the specific types of exceptions, followed by

handlers for the base type. If you add new derived exceptions, existing client code will still catch the exception through the base type. If you are using an exception to report a programming error, that report will be easier to understand.

22. **Sometimes simple aggregation does the job**. A "passenger comfort system" on an airline consists of disconnected elements: seat, air conditioning, video, etc., and yet you must create many of these in a plane. Do you make private members and build a whole new interface? No—in this case, the components are also part of the public interface, so create public member objects. Those objects have their own private implementations, which are still safe. Be aware that simple aggregation is not a solution to use often, but it does happen.
23. **Consider the perspective of the client programmer and the person maintaining the code**. Design your class to be as obvious as possible to use. Anticipate the kind of changes to be made, and design your class so those changes are easy.
24. **Watch out for "giant object syndrome."** This is often an affliction of procedural programmers who are new to OOP and who end up writing a procedural program and sticking it inside one or two giant objects. With the exception of application frameworks, objects represent concepts in your application, not the application itself.
25. **If you must do something ugly, at least localize the ugliness inside a class**.
26. **If you must do something nonportable, make an abstraction for that service and localize it within a class**. This extra level of indirection prevents the nonportability from being distributed throughout your program. (This idiom is embodied in the *Bridge* Pattern, among others).
27. **Objects should not simply hold some data**. They should also have well-defined behaviors. However, "data transfer objects" aka *Messenger*s are appropriate when used expressly to package and transport a group of items when a generalized collection is inappropriate.
28. **Choose composition first when creating new classes from existing classes**. Only use inheritance if it is required by your design. If you use inheritance where composition will work, your designs become needlessly complicated.
29. **Use inheritance and method overriding to express differences in behavior, and fields to express variations in state**. If you find a class using state variables along with methods switching behavior based on those variables, re-

design it to express the differences in behavior within subclasses and overridden methods. An extreme anti-example is inheriting different classes to represent colors instead of using a "color" field.

30. **Watch out for *variance***. Two semantically different objects may have identical actions or responsibilities. There is a natural temptation to try to make one a subclass of the other just to benefit from inheritance. This is called variance, but there's no real justification to force a superclass/subclass relationship where it doesn't exist. A better solution is to create a general base class that produces an interface for both as derived classes. You still benefit from inheritance and will probably make an important discovery about the design.
31. **Watch out for *limitation* during inheritance**. The clearest designs add new capabilities to inherited ones. A suspicious design removes old capabilities during inheritance without adding new ones. But rules are made to be broken, and if you are working from an old class library, it may be more efficient to restrict an existing class in its subclass than it would be to restructure the hierarchy so your new class fits in where it should, above the old class.
32. **Use design patterns to eliminate "naked functionality."** That is, if only one object of your class should be created, don't bolt ahead to the application and write a comment "Make only one of these." Wrap it in a singleton. If you have a lot of messy code in your main program that creates your objects, look for a creational pattern like a factory method where you can encapsulate that creation. Eliminating "naked functionality" will not only make your code much easier to understand and maintain, but it will also make it more bulletproof against the well-intentioned maintainers that come after you.
33. **Review utility classes**. A *utility class* has all `static` methods; it is basically a way to create standalone functions. Sometimes `static` fields will creep into your utility class. This means your class is now also holding state, and it might work better as a regular class. Try removing all the `statics` and see if that improves it.
34. **Watch out for "analysis paralysis."** Remember you must usually move forward in a project before you know everything. Often the best and fastest way to learn about some of your unknown factors is to go to the next step rather than trying to figure it out in your head. You can't know the solution until you *have* the solution. Java has built-in firewalls; let them work for you. Your mistakes in a class or set of classes won't destroy the integrity of the whole

system.

35. **When you think you've got a good analysis, design, or implementation, do a walkthrough**. Bring someone in from outside your group—this doesn't have to be a consultant, but can be someone from another group within your company. Reviewing your work with a fresh pair of eyes can reveal problems at a stage when it's much easier to fix them, and more than pays for the time and money overhead for the walkthrough process.

Implementation

1. **Follow coding conventions**. There are plenty of different conventions, for example, Google uses these[187] (the code in this book follows these conventions as much as I was able). If you doggedly stick to the coding style you've always used for some other language, you make it harder for your reader. Whatever coding conventions you decide on, ensure they are consistent throughout the project. Integrated development environments usually have reformatters and checkers built in.
2. **Whatever coding style you use, it really does make a difference if your team (and even better, your company) standardizes on it**. This means to the point that everyone considers it fair game to fix someone else's coding style if it doesn't conform. The value of standardization is it takes less brain cycles to parse the code, so you can focus more on what the code means.
3. **Follow standard capitalization rules**. Capitalize the first letter of class names. The first letter of fields, methods, and objects (references) should be lowercase. All identifiers should run their words together, and capitalize the first letter of all intermediate words. For example:
   - `ThisIsAClassName`
   - `thisIsAMethodOrFieldName`

   Capitalize *all* the letters (and use underscore word separators) of `static final` primitive identifiers that have constant initializers in their definitions. This indicates they are compile-time constants.
   - **Packages are a special case**—they are all lowercase letters, even for intermediate words. The domain extension (com, org, net, edu, etc.) should also be lowercase. (This was a change between Java 1.1 and Java 2.)

[187] https://google.github.io/styleguide/javaguide.html

4. **Don't create your own "decorated" private field names**. This is usually seen in the form of prepended underscores and characters. Hungarian notation is the worst example of this, where you attach extra characters that indicate data type, use, location, etc., as if you were writing assembly language and the compiler provided no extra assistance at all. These notations are confusing, difficult to read, and unpleasant to enforce and maintain. Let classes and packages do the name scoping for you. If you feel you must decorate names to prevent confusion, your code is probably too confusing anyway and should be simplified.
5. **Follow a "canonical form"** when creating a class for general-purpose use. Include definitions for `equals()`, `hashCode()`, `toString()`, `clone()` (implement `Cloneable`, or choose some other object copying approach, like serialization), and implement `Comparable` and `Serializable`. For *Messenger* classes, the upcoming "record" feature in Java 17 will do most of this for you.
6. **Use the "get," "set," and "is" naming conventions** for methods that read and change `private` fields. Not only does it make it easy to use your class, but it's a standard way to name these kinds of methods, so it is more easily understood by the reader.
7. **For each class you create, include JUnit tests for that class** (see *junit.org*, and the example in Validating Your Code). You don't need to remove the test code to use the class in a project, and if you make changes, you can easily rerun the tests. The test code also becomes an example of how to use your class.
8. **Sometimes you need inheritance to access *protected* members of the base class**. This can lead to a perceived need for multiple base types. If you don't need to upcast, first derive a new class to perform the protected access. Then make that new class a member object inside any class that uses it, instead of inheriting.
9. **Avoid the use of *final* methods for efficiency purposes**. Use `final` for this purpose only when profiling shows a method invocation is the bottleneck.
10. **If two classes are associated with each other in some functional way (such as collections and iterators), try to make one an inner class of the other**. This not only emphasizes the association between the classes, but it allows class name reuse within a single package by nesting it within another class. The Java collections library does this by defining an inner `Iterator` class inside each collection class, thereby providing the collections with a common interface. The other reason to use an inner class is as part of the `private` implementation.

Here, the inner class is beneficial for implementation hiding rather than the class association and prevention of namespace pollution noted above.

11. **Anytime you notice that classes appear to have high coupling with each other, consider the coding and maintenance improvements you might get by using inner classes**. The use of inner classes will not uncouple the classes, but rather make the coupling explicit and more convenient.
12. **Don't fall prey to premature optimization**. This way lies madness. In particular, don't worry about writing (or avoiding) native methods, making some methods `final`, or tweaking code to be efficient when you are first constructing the system. Your primary goal should be to prove the design. Even if the design requires a certain efficiency, *first make it work, then make it fast.*
13. **Keep scopes as small as possible so the visibility and lifetime of your objects are as small as possible**. This reduces the chance of using an object in the wrong context and hiding a difficult-to-find bug. For example, suppose you have a collection and a piece of code that iterates through it. If you copy that code to use with a new collection, you may accidentally end up using the size of the old collection as the upper bound of the new one. If, however, the old collection is out of scope, the error is caught at compile time.
14. **Use the collections in the standard Java library**. Become proficient with their use and you'll greatly increase your productivity. Prefer `ArrayList` for sequences, `HashSet` for sets, `HashMap` for associative arrays, and `LinkedList` for stacks (rather than `Stack`, although you may create an adapter to give a stack interface) and queues (which may also warrant an adapter, as shown in this book). When you use the first three, upcast to `List`, `Set`, and `Map`, respectively, so you can easily change to a different implementation if necessary.
15. **For a program to be robust, each component must be robust**. Use all the tools provided by Java—access control, exceptions, type checking, synchronization, and so on—in each class you create. That way you can safely move to the next level of abstraction when building your system.
16. **Prefer compile-time errors to run-time errors**. Try to handle an error as close to the point of its occurrence as possible. Catch any exceptions in the nearest handler that has enough information to deal with them. Do what you can with the exception at the current level; if that doesn't solve the problem, rethrow the exception.
17. **Watch for long method definitions**. Methods should be brief, functional units

that describe and implement a discrete part of a class interface. A long and complicated method is difficult and expensive to maintain, and is probably trying to do too much all by itself. If you see such a method, it indicates that, at the least, it should be broken up into multiple methods. It may also suggest the creation of a new class. Small methods will also foster reuse within your class. (Sometimes methods must be large, but they should still do just one thing.)

18. **Keep things as "*private* as possible"**. Once you publicize an aspect of your library (a method, a class, a field), you can never take it out. If you do, you'll wreck somebody's existing code, forcing them to rewrite and redesign. If you publicize only what you must, you can change everything else with impunity, and because designs tend to evolve, this is an important freedom. In this way, implementation changes have minimal impact on derived classes. Privacy is especially important when dealing with multithreading—only `private` fields can be protected against un-`synchronized` use. Classes with package access should still have `private` fields, but it usually makes sense to give the methods of package access rather than making them `public`.
19. **Use comments liberally, and use the *Javadoc* comment-documentation syntax to produce your program documentation**. However, the comments should add genuine meaning to the code; comments that only reiterate what the code is clearly expressing are annoying. Note that the typical verbose detail of Java class and method names reduce the need for some comments.
20. **Avoid using "magic numbers"**. These are numbers hard-wired into code. These are a nightmare if you must change them because you never know if "100" means "the array size" or "something else entirely." Instead, create a constant with a descriptive name and use the constant identifier throughout your program. This makes the program easier to understand and much easier to maintain.
21. **When creating constructors, consider exceptions**. In the best case, the constructor won't do anything that throws an exception. In the next-best scenario, the class is composed and inherited from robust classes only, so it needs no cleanup if an exception is thrown. Otherwise, you must clean up composed classes inside a `finally` clause. If a constructor must fail, the appropriate action is to throw an exception, so the caller doesn't continue blindly, thinking that the object was created correctly.
22. **Inside constructors, do only what is necessary to set the object into**

the proper state. Actively avoid calling other methods (except for `final` methods), because those methods can be overridden by someone else to produce unexpected results during construction. (See the Housekeeping chapter for details.) Smaller, simpler constructors are less likely to throw exceptions or cause problems.

23. **If your class requires any cleanup when the client programmer is finished with the object, place the cleanup code in a single, well-defined method**, with a name like `dispose()` that clearly suggests its purpose. In addition, place a `boolean` flag in the class to indicate whether `dispose()` was called so `finalize()` can check for "the termination condition" (see the Housekeeping chapter).
24. **The responsibility of *finalize()* can only be to verify "the termination condition" of an object for debugging**. (See the Housekeeping chapter.) In special cases, it might be needed to release memory that would not otherwise be released by the garbage collector. The garbage collector might not get called for your object so you cannot use `finalize()` to perform necessary cleanup. For that you must create your own `dispose()` method. In the `finalize()` method for the class, check to make sure that the object was cleaned up and throw a class derived from `RuntimeException` if it hasn't, to indicate a programming error. Before relying on such a scheme, ensure that `finalize()` works on your system. (You might need to call `System.gc()` to ensure this behavior.)
25. **If an object must be cleaned up (other than by garbage collection) within a particular scope, use the following idiom**: initialize the object and, if successful, immediately enter a `try` block with a `finally` clause that performs the cleanup.
26. **When overriding *finalize()* during inheritance, call *super.finalize()***. (This is not necessary if `Object` is your immediate superclass.) Call `super.finalize()` as the *final* act of your overridden `finalize()` rather than the first, to ensure that base-class components are still valid if you need them.
27. **When creating a fixed-size collection of objects, transfer them to an array**, especially if you're returning this collection from a method. This way you get the benefit of the array's compile-time type checking, and the recipient of the array might not need to cast the objects in the array to use them. Note that the base-class of the collections library, `java.util.Collection`, has two `toArray()` methods to accomplish this.

28. **Choose *interfaces* over *abstract* classes**. If you know something is a base class, your first choice should be to make it an interface, and only if you need method definitions or member variables should you change it to an `abstract` class. An interface talks about what the client wants to do, while a class tends to focus on (or allow) implementation details.
29. **To avoid a highly frustrating experience**, **make sure there is only one unpackaged class of each name anywhere in your classpath**. Otherwise, the compiler can find the identically-named other class first, and report error messages that make no sense. If you suspect you are having a classpath problem, try looking for `.class` files with the same names at each of the starting points in your classpath. Ideally, put all your classes within packages.
30. **Watch for accidental overloading**. If you attempt to override a base-class method and you don't get the spelling right, you'll end up adding a new method rather than overriding an existing method. However, this is perfectly legal, so you won't get any error message from the compiler or run-time system; your code simply won't work correctly. Always use the `@Override` annotation to prevent this.
31. **Watch for premature optimization**. First make it work, then make it fast—but only if you must, and only if you've proved there is a performance bottleneck in a particular section of your code. Unless you use a profiler to discover a bottleneck, you waste your time. The hidden extra cost of performance tweaks is that your code becomes less understandable and maintainable.
32. **Remember that code is read much more than it is written**. Clean designs make for easy-to-understand programs, but comments, detailed explanations, tests, and examples are invaluable. They help both you and everyone who comes after you. If nothing else, the frustration of trying to ferret out useful information from the JDK documentation should convince you.

Appendix: Javadoc

Possibly the biggest problem with documenting code is maintaining that documentation. If the documentation and the code are separate, it becomes tedious to change the documentation every time you change the code. The solution seems simple: Link the code to the documentation. The easiest way to do this is to put everything in the same file. To complete the picture, however, you need a special comment syntax to mark the documentation and a tool to extract those comments into a useful form. This is what Java has done.

The tool to extract the comments is called *Javadoc*, and it comes as part of the JDK installation. It uses some of the technology from the Java compiler to look for special comment tags. It not only extracts the information marked by these tags, but it also pulls out the class name or method name that adjoins the comment. This way you can get away with the minimal amount of work to generate decent program documentation.

The output of Javadoc is an HTML file you can view with your Web browser. With Javadoc, you have a straightforward standard for creating documentation, so you can expect documentation for all Java libraries.

In addition, you can write your own Javadoc handlers, called *doclets*, to perform special operations on the information processed by Javadoc (to produce output in a different format, for example).

What follows is only an introduction and overview of the basics of Javadoc. A thorough description is found in the JDK documentation.

Syntax

All Javadoc directives occur within comments that begin with `/**` (but still end with `*/`). There are two primary ways to use Javadoc: Embed HTML or use "doc tags." *Standalone doc tags* are directives that start with an `@` and are placed at the beginning

of a comment line. (A leading *, however, is ignored.) *Inline doc tags* can appear anywhere within a Javadoc comment and also start with an @ but are surrounded by curly braces.

There are three types of comment documentation, which correspond to the element the comment precedes: class, field, or method. That is, a class comment appears right before the definition of a class, a field comment appears right before the definition of a field, and a method comment appears right before the definition of a method. As a simple example:

```
// javadoc/Documentation1.java
/** A class comment */
public class Documentation1 {
  /** A field comment */
  public int i;
  /** A method comment */
  public void f() {}
}
```

Javadoc processes comment documentation only for `public` and `protected` members. Comments for `private` and package-access members (see the Implementation Hiding chapter) are ignored by default, and you'll see no output. This makes sense because only `public` and `protected` members are available outside the file, which is the client programmer's perspective. You can use the `-private` flag to include `private` members.

To process the preceding code through Javadoc, the command is:

```
javadoc Documentation1.java
```

This produces a set of HTML files; if you open `index.html` in your browser you'll see that the result has the same standard format as all the rest of the Java documentation, so users are comfortable with the format and can easily navigate your classes.

Embedded HTML

Javadoc passes HTML code untouched to the generated HTML document. This allows you full use of HTML; however, the primary motive is to let you format code, such as:

```
// javadoc/Documentation2.java
/** <pre>
 * System.out.println(new Date());
 * </pre>
 */
public class Documentation2 {}
```

You can also use HTML just as you would in any other Web document to format the regular text in your descriptions:

```
// javadoc/Documentation3.java

/** You can <em>even</em> insert a list:
 * <ol>
 * <li> Item one
 * <li> Item two
 * <li> Item three
 * </ol>
 */

public class Documentation3 {}
```

Note that within the documentation comment, asterisks at the beginning of a line are discarded by Javadoc, along with leading spaces. Javadoc reformats everything so it conforms to the standard documentation appearance. Don't use headings such as `<h1>` or `<hr>` as embedded HTML, because Javadoc inserts its own headings and yours will interfere with them.

All types of comment documentation—class, field, and method—can support embedded HTML.

Some Example Tags

Here are some of the Javadoc tags available for code documentation. Before trying to do anything serious using Javadoc, consult the Javadoc reference in the JDK documentation to learn all the different ways you can use Javadoc.

@see

This tag refers to documentation in other classes. Javadoc will generate HTML with the `@see` tags hyperlinked to the other documentation. The forms are:

```
@see classname
@see fully-qualified-classname
@see fully-qualified-classname#method-name
```

Each adds a hyperlinked "See Also" entry to the generated documentation. Javadoc does not check the validity of the hyperlinks.

{@link *package.class#member label*}

Very similar to `@see`, except it can be used inline and uses **label** as the hyperlink text rather than "See Also."

{@docRoot}

Produces the relative path to the documentation root directory. Useful for explicit hyperlinking to pages in the documentation tree.

{@inheritDoc}

Inherits the documentation from the nearest base class of this class into the current doc comment.

@version

This is of the form:

```
@version version-information
```

where `version-information` is any significant information you see fit to include. When the `-version` flag is placed on the Javadoc command line, the version information is called out specially in the generated HTML documentation.

@author

This is of the form:

```
@author author-information
```

where `author-information` is presumably your name, but can also include your email address or any other appropriate information. When the `-author` flag is placed on the Javadoc command line, the author information is called out specially in the generated HTML documentation.

You can use multiple author tags for a list of authors, but they must be placed consecutively. All author information is lumped together into a single paragraph in the generated HTML.

@since

This tag indicates the version of this code that began using a particular feature. It appears, for example, in the HTML Java documentation to indicate the version of the JDK where a feature first appeared.

@param

This produces documentation for method arguments:

```
@param parameter-name description
```

where `parameter-name` is the identifier in the method parameter list, and `description` is text that can continue on subsequent lines. The description is considered finished when a new documentation tag is encountered. You can have any number of these, presumably one for each parameter.

@return

This documents the return value:

```
@return description
```

where `description` gives you the meaning of the return value. It can continue on subsequent lines.

@throws

A method can produce any number of different types of exceptions, all of which need descriptions. The form for the exception tag is:

```
@throws fully-qualified-class-name description
```

where *fully-qualified-class-name* gives an unambiguous name of an exception class, and *description* (which can continue on subsequent lines) tells you why this particular type of exception can emerge from the method call.

@deprecated

This indicates features that are superseded by an improved feature. The deprecated tag suggests that you no longer use this particular feature, because sometime in the future it is likely to be removed. A method marked `@deprecated` causes the compiler to issue a warning if it is used. In Java 5, the `@deprecated` Javadoc tag was superseded by the `@Deprecated` *annotation* (described in the Annotations chapter).

Documentation Example

Here is `objects/HelloDate.java` with documentation comments:

```
// javadoc/HelloDateDoc.java
import java.util.*;

/** The first On Java 8 example program.
 * Displays a String and today's date.
 * @author Bruce Eckel
 * @author www.MindviewInc.com
 * @version 5.0
 */
public class HelloDateDoc {
  /** Entry point to class & application.
   * @param args array of String arguments
   * @throws exceptions No exceptions thrown
   */
  public static void main(String[] args) {
    System.out.println("Hello, it's: ");
    System.out.println(new Date());
  }
}
/* Output:
```

```
Hello, it's:
Sun Jan 24 08:49:10 MST 2021
*/
```

You can find many examples of Javadoc comment documentation in the source code for the Java standard libraries.

Appendix: Passing and Returning Objects

> By now you are reasonably comfortable with the idea that when you're "passing" an object, you're actually passing a reference.

In many programming languages you can use that language's "regular" way to pass objects around, and most of the time everything works fine. But there usually comes a point at which you must do something irregular, and suddenly things get a bit more complicated. Java is no exception, and it's important you understand exactly what's happening as you pass objects and manipulate them. This appendix provides that insight.

Another way to pose the question of this appendix, if you're coming from a programming language so equipped, is "Does Java have pointers?" Every object identifier in Java (except for primitives) is one of these pointers, but their use is restricted and guarded not only by the compiler but by the run-time system. Or to put it another way, Java has pointers, but no pointer arithmetic. These are what I've been calling "references," and you can think of them as "safety pointers," not unlike the safety scissors of elementary school—they aren't sharp, so you cannot hurt yourself without great effort, but they can sometimes be tedious.

Passing References

When you pass a reference into a method, you're still pointing to the same object. A simple experiment demonstrates this:

```
// references/PassReferences.java

public class PassReferences {
  public static void f(PassReferences h) {
    System.out.println("h inside f(): " + h);
  }
  public static void main(String[] args) {
    PassReferences p = new PassReferences();
    System.out.println("p inside main(): " + p);
    f(p);
  }
}
/* Output:
p inside main(): PassReferences@19e0bfd
h inside f(): PassReferences@19e0bfd
*/
```

The method `toString()` is automatically invoked in the print statements, and `PassReferences` inherits directly from `Object` with no redefinition of `toString()`. Thus, `Object`'s version of `toString()` is used, which prints out the class of the object followed by the address where that object is located (not the reference, but the actual object storage).

The output shows that that both `p` and `h` refer to the same object. This is far more efficient than duplicating a new `PassReferences` object just so you can send an argument to a method. But it brings up an important issue.

Aliasing

Aliasing means that more than one reference is tied to the same object, as in the preceding example. The problem with aliasing occurs when someone *writes* to that object. If the owners of the other references aren't expecting that object to change, they'll be surprised. This can be demonstrated:

```
// references/Alias1.java
// Aliasing two references to one object

public class Alias1 {
  private int i;
  public Alias1(int ii) { i = ii; }
  public static void main(String[] args) {
    Alias1 x = new Alias1(7);
    Alias1 y = x; // Assign the reference    // [1]
    System.out.println("x: " + x.i);
    System.out.println("y: " + y.i);
    System.out.println("Incrementing x");
    x.i++;                                  // [2]
    System.out.println("x: " + x.i);
    System.out.println("y: " + y.i);
  }
}
/* Output:
x: 7
y: 7
Incrementing x
x: 8
y: 8
*/
```

- **[1]** Here, a new `Alias1` reference is created, but instead of being assigned to a fresh object created with `new`, it's assigned to an existing reference. So the contents of reference `x`, which is the address of the object `x` is pointing to, is assigned to `y`, and thus both `x` and `y` are attached to the same object.
- **[2]** When `x`s `i` is incremented, `y`s `i` is affected as well, as shown in the output.

The best solution is simply not to do it; don't consciously alias more than one reference to an object at the same scope. Your code is then much easier to understand and debug. However, when you're passing a reference in as an argument—which is the way Java is supposed to work—you automatically alias, because the local reference that's created can modify the "outside object" (the object that was created outside the scope of the method):

```
// references/Alias2.java
// Method calls implicitly alias their arguments

public class Alias2 {
  private int i;
  public Alias2(int i) { this.i = i; }
  public static void f(Alias2 reference) {
    reference.i++;
  }
  public static void main(String[] args) {
    Alias2 x = new Alias2(7);
    System.out.println("x: " + x.i);
    System.out.println("Calling f(x)");
    f(x);
    System.out.println("x: " + x.i);
  }
}
/* Output:
x: 7
Calling f(x)
x: 8
*/
```

The method is changing its argument, the outside object. When this kind of situation arises, you must decide whether it makes sense, whether the user expects it, and whether it's going to cause problems. The answers are usually no, no, and yes, which is why pure functional languages disallow this behavior.

In general, you call a method to produce a return value and/or change the state of the object *the method is called for*. It's much less common to call a method to manipulate its arguments; this is one form of "calling a method for its *side effects*." Thus, when you create a method that modifies its arguments, the user must be clearly instructed and warned about the use of that method and its potential surprises. Because of the confusion and pitfalls, it's much better to avoid changing the argument.

If you must modify an argument during a method call and you don't intend to modify the outside argument, then protect that argument by making a copy inside your method. That's the subject of much of this appendix.

Making Local Copies

All argument passing in Java is performed by passing references. That is, when you pass "an object," you're really passing only a reference to an object that lives outside the method, so if you perform any modifications through that reference, you modify the outside object. In addition:

- Aliasing happens automatically during argument passing.
- There are no local objects, only local references.
- References have scopes, objects do not.
- Object lifetime is never an issue in Java.

There is no language support (e.g., "const") to prevent objects from being modified and to stop the negative effects of aliasing. You can't simply use the `final` keyword in the argument list; that only prevents you from rebinding the reference to a different object.

If you're strictly reading information from an object and not modifying that object, passing a reference is the most efficient form of argument passing. This is nice; the default way of doing things is also the most efficient. However, sometimes it's necessary to treat the object as if it were "local" so changes you make affect only a local copy and do not modify the outside object. Many programming languages support the ability to automatically make a local copy of the outside object, inside the method.[188] Java does not, but it allows you to produce this effect.

Pass By Value

This brings up the terminology issue, which always seems good for an argument. The term is "pass by value," and the meaning depends on how you perceive the operation of the program. The concept is that you get a local copy of whatever you're passing, but the real question is how you think about what you're passing. When it comes to the meaning of "pass by value," there are two fairly distinct camps:

[188] In C, which generally handles small bits of data, the default is pass by value. C++ had to follow this form, but with objects, pass by value isn't usually the most efficient approach. In addition, coding classes to support pass by value in C++ is a big headache.

1. Java passes everything by value. When you pass primitives into a method, you get a distinct copy of the primitive. When you pass a reference into a method, you get a copy of the reference. Ergo, everything is pass by value. The assumption is that you're always thinking (and caring) that references are passed, but it seems like the Java design has gone a long way toward allowing you to ignore (most of the time) that you're working with a reference. That is, it seems to allow you to think of the reference as "the object" because it implicitly dereferences it whenever you make a method call.
2. Java passes primitives by value, but objects are passed by reference. This is the world view that the reference is an alias for the object, so you *don't* think about passing references, but instead say "I'm passing the object." You don't get a local copy of the object when you pass it into a method so objects are clearly not passed by value. There appeared to be some support for this view within Sun because at one time one of the "reserved but not implemented" keywords was `byvalue` (which will never be implemented).

Having given both camps a good airing, and after saying "It depends on how you think of a reference," I attempt to sidestep the issue. In the end, it isn't *that* important—what is important is you understand that passing a reference allows the caller's object to be changed unexpectedly.

Cloning Objects

Before you wade too far into cloning, check out the alternatives at the end of the summary.

The most likely reason for making a local copy of an object is if you modify that object but you don't want to modify the caller's object. One approach for making a local copy is to use the `clone()` method. `clone()` is defined as `protected` in the base class `Object`. You must override `clone()` as `public` in any derived classes you want to clone. For example, the standard library class `ArrayList` overrides `clone()`, so we can call `clone()` for `ArrayList`:

```
// references/CloneArrayList.java
// The clone() operation works for only a few
// items in the standard Java library
import java.util.*;
import java.util.stream.*;

class Int {
  private int i;
  Int(int ii) { i = ii; }
  public void increment() { i++; }
  @Override public String toString() {
    return Integer.toString(i);
  }
}

public class CloneArrayList {
  public static void main(String[] args) {
    ArrayList<Int> v = IntStream.range(0, 10)
      .mapToObj(Int::new)
      .collect(Collectors
        .toCollection(ArrayList::new));
    System.out.println("v: " + v);
    @SuppressWarnings("unchecked")
    ArrayList<Int> v2 = (ArrayList<Int>)v.clone();
    // Increment all v2's elements:
    v2.forEach(Int::increment);
    // See if it changed v's elements:
    System.out.println("v: " + v);
  }
}
/* Output:
v: [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
v: [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
*/
```

The `clone()` method produces an `Object`, which must then be recast to the proper type. This example shows how `ArrayList`'s `clone()` method *does not* automatically try to clone each of the objects that the `ArrayList` contains—the old `ArrayList` and the cloned `ArrayList` are aliased to the same objects. This is a *shallow copy* because it's copying only the "surface" portion of an object. The actual object consists of this "surface," plus all the objects that the references are pointing to, plus all the objects

those objects are pointing to, etc. This is often called the "web of objects." Making a full copy of the entire mess is called a *deep copy*.

You see the effect of the shallow copy in the output, where the actions performed on `v2` affect `v`. Not `clone()`ing the objects contained in the `ArrayList` is probably a fair assumption, because there's no guarantee that those objects *are* cloneable.[189]

Adding Cloneability to a Class

Even though `clone()` is defined in the base-of-all-classes `Object`, cloning is *not* automatically available in every class. This would seem counterintuitive to the idea that base-class methods are always available in derived classes. Cloning in Java does indeed go against this idea; if you want it to exist for a class, you must specifically add code to make cloning work.

Using a Trick with `Protected`

To prevent default cloneability in every class you create, the `clone()` method is `protected` in the base class `Object`. This means it's not available by default to the client programmer who is simply using the class (not subclassing it). It also means you cannot call `clone()` via a reference to the base class. It is, in effect, a way to give you, at compile time, the information that your object is not cloneable—and oddly enough, most classes in the standard Java library are not cloneable. Thus, if you say:

```
Integer x = 1;
x = x.clone();
```

You get, at compile time, an error message that says `clone()` is not accessible (`Integer` doesn't override it and it defaults to the `protected` version).

If, however, you're in a method of a class *derived* from `Object` (as all classes are), then you have permission to call `Object.clone()` because it's `protected` and you're an inheritor. The base class `clone()` has useful functionality; it performs the actual bitwise duplication *of the derived-class object*, thus acting as the common cloning operation. However, you must then make *your* clone operation `public` for it to be accessible. So, two key issues when you clone are:

[189] This is not the dictionary spelling of the word, but it's what is used in the Java library, so I've used it here, too, in some hopes of reducing confusion.

- Call `super.clone()`
- Make your clone `public`

You'll probably override `clone()` in any further derived classes; otherwise, your (now `public`) `clone()` is used, and that might not do the right thing (although, because `Object.clone()` makes a copy of the actual object, it might). The `protected` trick works only once: the first time you inherit from a class that has no cloneability and you want to make a class that's cloneable. In any classes inherited from your class, the `clone()` method is available because it's not possible in Java to reduce the access of a method during derivation. That is, once a class is cloneable, everything derived from it is cloneable unless you use provided mechanisms (described later) to "turn off" cloning.

Implementing the `Cloneable` Interface

There's one more action to complete the cloneability of an object: implement the `Cloneable` interface. This is an empty (tagging) interface.

There are two reasons for the existence of the `Cloneable` interface. First, you might have an upcast reference to a base type and not know whether it's possible to clone that object. Here, you can use the `instanceof` keyword (described in the Reflection chapter) to find out whether the reference is connected to an object that can be cloned:

```
if(myReference instanceof Cloneable) // ...
```

The second reason is that mixed into this design for cloneability is the thought that maybe you don't want all types of objects to be cloneable. So `Object.clone()` verifies that a class implements the `Cloneable` interface. If not, it throws a `CloneNotSupportedException` exception. So in general, you're forced to `implement Cloneable` as part of support for cloning.

Successful Cloning

Once you understand the details of implementing the `clone()` method, you're able to create classes that can be easily duplicated to provide a local copy:

```
// references/LocalCopy.java
// Creating local copies with clone()

class Duplo implements Cloneable {
  private int n;
  Duplo(int n) { this.n = n; }
  @Override public Duplo clone() {              // [1]
    try {
      return (Duplo)super.clone();
    } catch(CloneNotSupportedException e) {
      throw new RuntimeException(e);
    }
  }
  public int getValue() { return n; }
  public void setValue(int n) { this.n = n; }
  public void increment() { n++; }
  @Override public String toString() {
    return Integer.toString(n);
  }
}

public class LocalCopy {
  public static Duplo g(Duplo v) {
    // Passing a reference, modifies outside object:
    v.increment();
    return v;
  }
  public static Duplo f(Duplo v) {
    v = v.clone(); // Local copy                 // [2]
    v.increment();
    return v;
  }
  public static void main(String[] args) {
    Duplo a = new Duplo(11);
    Duplo b = g(a);
    // Reference equivalence, not object equivalence:
    System.out.println("a == b: " + (a == b) +
      "\na = " + a + "\nb = " + b);
    Duplo c = new Duplo(47);
    Duplo d = f(c);
    System.out.println("c == d: " + (c == d) +
      "\nc = " + c + "\nd = " + d);
```

```
  }
}
/* Output:
a == b: true
a = 12
b = 12
c == d: false
c = 47
d = 48
*/
```

First of all, for `clone()` to be accessible, you must make it `public`. Second, for the initial part of your `clone()` operation, call the base-class version of `clone()`. The `clone()` that's called here is the one that's predefined inside `Object`, and you can call it because it's `protected` and thereby accessible in derived classes.

`Object.clone()` figures out how big the object is, creates enough memory for a new one, and copies all the bits from the old to the new. This is called a *bitwise copy*, and is typically what you'd expect a `clone()` method to do. But before `Object.clone()` performs its operations, it first checks to see if a class is `Cloneable`—that is, whether it implements the `Cloneable` interface. If it doesn't, `Object.clone()` throws a `CloneNotSupportedException` to indicate you can't clone it. Thus, you've got to surround your call to `super.clone()` with a `try` block to catch an exception that should never happen (because you've implemented the `Cloneable` interface).

- **[1]** Notice the use of covariant return types here. The base class `Object`'s `clone()` can only return `Object`, but the derived class `clone()` can return a more specific type. Before covariant return types, you had to cast the return type down to the proper type, but now it can be validated at compile time.
- **[2]** Here, you see a call to `clone()` which returns a `Duplo`, with no cast required.

In `LocalCopy`, the methods `g()` and `f()` demonstrate the difference between the two approaches for argument passing. The `g()` method shows passing by reference; it modifies the outside object and returns a reference to that outside object, whereas `f()` clones the argument, thereby decoupling it and leaving the original object alone. It can then proceed to do whatever it wants—even return a reference to this new object without any ill effects to the original.

In `main()`, the difference between the effects of the two different argument-passing approaches is tested. It's important to notice that the equivalence tests in Java do not look inside the objects being compared to see if their values are the same. The `==` and `!=` operators are simply comparing the *references*. If the addresses inside the references are the same, the references are pointing to the same object and are therefore "equal." So what the operators are really testing is whether the references are aliased to the same object!

The Effect of `Object.clone()`

What actually happens when `Object.clone()` is called that makes it so essential to call `super.clone()` when you override `clone()` in your class? The `clone()` method in the root class is responsible for creating the correct amount of storage and making the bitwise copy of the bits from the original object into the new object's storage. That is, it doesn't just make storage and copy an `Object`; it actually figures out the size of the *real* object (not just the base-class object, but the derived object) that's copied, and duplicates that. All this happens from the code in the `clone()` method defined within the root class (that has no idea what's been inherited) so you can guess that the process involves reflection to determine the actual object that's cloned. This way, the `clone()` method can create the proper amount of storage and do the correct bitwise copy for that type.

The first part of the cloning process should usually be a call to `super.clone()`. This establishes the groundwork for the cloning operation by making an exact duplicate. At this point you can perform other operations necessary to complete the cloning.

To know for sure what those other operations are, you must understand exactly what `Object.clone()` buys you. In particular, does it automatically clone the destination of all the references? We can test this:

```
// references/Snake.java
// Tests cloning to see if reference
// destinations are also cloned

public class Snake implements Cloneable {
  private Snake next;
  private char c;
  // Value of i == number of segments
  public Snake(int i, char x) {
    c = x;
    if(--i > 0)
      next = new Snake(i, (char)(x + 1));
  }
  public void increment() {
    c++;
    if(next != null)
      next.increment();
  }
  @Override public String toString() {
    String s = ":" + c;
    if(next != null)
      s += next.toString();
    return s;
  }
  @Override public Snake clone() {
    try {
      return (Snake)super.clone();
    } catch(CloneNotSupportedException e) {
      throw new RuntimeException(e);
    }
  }
  public static void main(String[] args) {
    Snake s = new Snake(5, 'a');
    System.out.println("s = " + s);
    Snake s2 = s.clone();
    System.out.println("s2 = " + s2);
    s.increment();
    System.out.println(
      "after s.increment, s2 = " + s2);
  }
}
/* Output:
```

```
s = :a:b:c:d:e
s2 = :a:b:c:d:e
after s.increment, s2 = :a:c:d:e:f
*/
```

A `Snake` is made up of a bunch of segments, each of type `Snake`. Thus, it's a singly linked list. The segments are created recursively, decrementing the first constructor argument for each segment until zero is reached. To give each segment a unique tag, the second argument, a `char`, is incremented for each recursive constructor call.

The `increment()` method recursively increments each tag to show the change, and the `toString()` recursively prints each tag. The output shows that only the first segment is duplicated by `Object.clone()`, therefore it does a shallow copy. If you want the whole snake duplicated—a deep copy—you must perform the additional operations inside your overridden `clone()`.

You'll typically call `super.clone()` in any class derived from a cloneable class to make sure that all base-class operations (including `Object.clone()`) take place. This is followed by an explicit call to `clone()` for every reference in your object; otherwise those references are aliased to those of the original object. It's analogous to the way constructors are called: base-class constructor first, then the next-derived constructor, and so on, to the most-derived constructor. The difference is that `clone()` is not a constructor, so there's nothing to make it happen automatically. You must make sure to do it yourself.

Cloning a Composed Object

There's a problem you'll encounter when trying to deep copy a composed object. You must assume that the `clone()` method in the member objects will in turn perform a deep copy on *their* references, and so on. This is a commitment. It effectively means that for a deep copy to work, you must either control all code in all classes, or at least have enough knowledge about all classes involved in the deep copy to know they are performing their own deep copy correctly.

Here's what you must do to accomplish a deep copy when dealing with a composed object:

```
// references/DepthReading.java
// Cloning a composed object
package references;

public class DepthReading implements Cloneable {
  private double depth;
  public DepthReading(double depth) {
    this.depth = depth;
  }
  @Override public DepthReading clone() {
    try {
      return (DepthReading)super.clone();
    } catch(CloneNotSupportedException e) {
      throw new RuntimeException(e);
    }
  }
  public double getDepth() { return depth; }
  public void setDepth(double depth) {
    this.depth = depth;
  }
  @Override public String toString() {
    return String.valueOf(depth);
  }
}

// references/TemperatureReading.java
// Cloning a composed object
package references;

public class TemperatureReading implements Cloneable {
  private long time;
  private double temperature;
  public TemperatureReading(double temperature) {
    time = System.currentTimeMillis();
    this.temperature = temperature;
  }
  @Override public TemperatureReading clone() {
    try {
      return (TemperatureReading)super.clone();
    } catch(CloneNotSupportedException e) {
      throw new RuntimeException(e);
    }
```

```
  }
  public double getTemperature() {
    return temperature;
  }
  public void setTemperature(double temp) {
    this.temperature = temp;
  }
  @Override public String toString() {
    return String.valueOf(temperature);
  }
}

// references/OceanReading.java
// Cloning a composed object
package references;

public class OceanReading implements Cloneable {
  private DepthReading depth;
  private TemperatureReading temperature;
  public
  OceanReading(double tdata, double ddata) {
    temperature = new TemperatureReading(tdata);
    depth = new DepthReading(ddata);
  }
  @Override public OceanReading clone() {
    OceanReading or = null;
    try {
      or = (OceanReading)super.clone();
    } catch(CloneNotSupportedException e) {
      throw new RuntimeException(e);
    }
    // Must clone references:
    or.depth = (DepthReading)or.depth.clone();
    or.temperature =
      (TemperatureReading)or.temperature.clone();
    return or;
  }
  public TemperatureReading getTemperatureReading() {
    return temperature;
  }
  public void
  setTemperatureReading(TemperatureReading tr) {
```

```
    temperature = tr;
  }
  public DepthReading getDepthReading() {
    return depth;
  }
  public void setDepthReading(DepthReading dr) {
    this.depth = dr;
  }
  @Override public String toString() {
    return "temperature: " + temperature +
      ", depth: " + depth;
  }
}
```

Now we can test it using JUnit:

```
// references/tests/DeepCopyTest.java
package references;
import org.junit.jupiter.api.*;
import static org.junit.jupiter.api.Assertions.*;

public class DeepCopyTest {
  @Test
  public void testClone() {
    OceanReading reading =
      new OceanReading(33.9, 100.5);
    // Now clone it:
    OceanReading clone = reading.clone();
    TemperatureReading tr =
      clone.getTemperatureReading();
    tr.setTemperature(tr.getTemperature() + 1);
    clone.setTemperatureReading(tr);
    DepthReading dr = clone.getDepthReading();
    dr.setDepth(dr.getDepth() + 1);
    clone.setDepthReading(dr);
    assertEquals(reading.toString(),
      "temperature: 33.9, depth: 100.5");
    assertEquals(clone.toString(),
      "temperature: 34.9, depth: 101.5");
  }
}
```

`DepthReading` and `TemperatureReading` are similar; they both contain only primitives. Therefore, the `clone()` method can be simple: it calls `super.clone()` and returns the result. Note that the `clone()` code for both classes is identical.

`OceanReading` is composed of `DepthReading` and `TemperatureReading` objects and so, to produce a deep copy, its `clone()` must clone the references inside `OceanReading`. To accomplish this, the result of `super.clone()` must be cast to an `OceanReading` object (so you can access the `depth` and `temperature` references).

A Deep Copy with `ArrayList`

Let's revisit `CloneArrayList.java` from earlier in this appendix. This time the `Int2` class is cloneable, so the `ArrayList` can be deep copied:

```
// references/AddingClone.java
// You must go through a few gyrations
// to add cloning to your own class
import java.util.*;
import java.util.stream.*;

class Int2 implements Cloneable {
  private int i;
  Int2(int ii) { i = ii; }
  public void increment() { i++; }
  @Override public String toString() {
    return Integer.toString(i);
  }
  @Override public Int2 clone() {
    try {
      return (Int2)super.clone();
    } catch(CloneNotSupportedException e) {
      throw new RuntimeException(e);
    }
  }
}

// Inheritance doesn't remove cloneability:
class Int3 extends Int2 {
  private int j; // Automatically duplicated
  Int3(int i) { super(i); }
```

```
}

public class AddingClone {
  @SuppressWarnings("unchecked")
  public static void main(String[] args) {
    Int2 x = new Int2(10);
    Int2 x2 = x.clone();
    x2.increment();
    System.out.println(
      "x = " + x + ", x2 = " + x2);
    // Anything inherited is also cloneable:
    Int3 x3 = new Int3(7);
    x3 = (Int3)x3.clone();
    ArrayList<Int2> v = IntStream.range(0, 10)
      .mapToObj(Int2::new)
      .collect(Collectors
        .toCollection(ArrayList::new));
    System.out.println("v: " + v);
    ArrayList<Int2> v2 =
      (ArrayList<Int2>)v.clone();
    // Now clone each element:
    IntStream.range(0, v.size())
      .forEach(i -> v2.set(i, v.get(i).clone()));
    // Increment all v2's elements:
    v2.forEach(Int2::increment);
    System.out.println("v2: " + v2);
    // See if it changed v's elements:
    System.out.println("v: " + v);
  }
}
/* Output:
x = 10, x2 = 11
v: [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
v2: [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
v: [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
*/
```

`Int3` inherits `Int2`, and a new primitive member, `int j`, is added. You might think you'd need to override `clone()` again to make sure `j` is copied, but that's not the case. When `Int2`'s `clone()` is called as `Int3`'s `clone()`, it calls `Object.clone()`, which determines it's working with an `Int3` and duplicates all the bits in the `Int3`. As long as you don't add references that need cloning, the one call to `Object.clone()`

performs all necessary duplication regardless of how far down in the hierarchy `clone()` is defined.

Here's what's necessary to do a deep copy of an `ArrayList`: After the `ArrayList` is cloned, you have to step through and clone each one of the objects pointed to by the `ArrayList`. You'd also have to do something similar to this to do a deep copy of, for example, a `HashMap`.

The remainder of the example demonstrates that cloning actually happened by showing that, once an object is cloned, you can change it, and the original object is left untouched.

Deep Copy Via Serialization

When you consider Java's object serialization (introduced in the Appendix: Object Serialization), you might observe that an object that's first serialized and then deserialized is, in effect, cloned.

So why not use serialization to perform deep copying? Here's an example that compares the two approaches by timing them:

```
// references/Compete.java
import java.io.*;
import onjava.Timer;

class Thing1 implements Serializable {}
class Thing2 implements Serializable {
  Thing1 t1 = new Thing1();
}

class Thing3 implements Cloneable {
  @Override public Thing3 clone() {
    try {
      return (Thing3)super.clone();
    } catch(CloneNotSupportedException e) {
      throw new RuntimeException(e);
    }
  }
}
```

```
class Thing4 implements Cloneable {
  private Thing3 t3 = new Thing3();
  @Override public Thing4 clone() {
    Thing4 t4 = null;
    try {
      t4 = (Thing4)super.clone();
    } catch(CloneNotSupportedException e) {
      throw new RuntimeException(e);
    }
    // Clone the field, too:
    t4.t3 = t3.clone();
    return t4;
  }
}

public class Compete {
  public static final int SIZE = 100000;
  public static void
  main(String[] args) throws Exception {
    Thing2[] a = new Thing2[SIZE];
    for(int i = 0; i < SIZE; i++)
      a[i] = new Thing2();
    Thing4[] b = new Thing4[SIZE];
    for(int i = 0; i < SIZE; i++)
      b[i] = new Thing4();
    Timer timer = new Timer();
    try(
      ByteArrayOutputStream buf =
        new ByteArrayOutputStream();
      ObjectOutputStream oos =
        new ObjectOutputStream(buf)
    ) {
      for(Thing2 a1 : a) {
        oos.writeObject(a1);
      }
      // Now get copies:
      try(
        ObjectInputStream in =
          new ObjectInputStream(
            new ByteArrayInputStream(
              buf.toByteArray()))
      ) {
```

```
        Thing2[] c = new Thing2[SIZE];
        for(int i = 0; i < SIZE; i++)
          c[i] = (Thing2)in.readObject();
      }
    }
    System.out.println(
      "Duplication via serialization: " +
      timer.duration() + " Milliseconds");

    // Now try cloning:
    timer = new Timer();
    Thing4[] d = new Thing4[SIZE];
    for(int i = 0; i < SIZE; i++)
      d[i] = b[i].clone();
    System.out.println(
      "Duplication via cloning: " +
      timer.duration() + " Milliseconds");
  }
}
/* Output:
Duplication via serialization: 385 Milliseconds
Duplication via cloning: 38 Milliseconds
*/
```

`Thing2` and `Thing4` contain member objects so there's some deep copying going on. `Serializable` classes are easy to set up, but there's much more work going on to duplicate them. On the other hand, cloning involves more work to set up the class, but the actual duplication of objects is relatively simple.

Notice that serialization is at least an order of magnitude slower than cloning.

Adding Cloneability Further Down a Hierarchy

If you create a new class, its base class defaults to `Object`, which defaults to non-cloneability. As long as you don't explicitly add cloneability, you won't get it. But you can add it in at any layer and it will then be cloneable from that layer downward, like this:

```
// references/HorrorFlick.java
// Insert Cloneability at any level of inheritance

class Person {}

class Hero extends Person {}

class Scientist extends Person implements Cloneable {
  @Override public Scientist clone() {
    try {
      return (Scientist)super.clone();
    } catch(CloneNotSupportedException e) {
      throw new RuntimeException(e);
    }
  }
}

class MadScientist extends Scientist {}

public class HorrorFlick {
  public static void main(String[] args) {
    Person p = new Person();
    Hero h = new Hero();
    Scientist s = new Scientist();
    MadScientist m = new MadScientist();
    //- p = (Person)p.clone(); // Compile error
    //- h = (Hero)h.clone(); // Compile error
    s = s.clone();
    m = (MadScientist)m.clone();
  }
}
```

Before cloneability was added in the hierarchy, the compiler stopped you from trying to clone things. When cloneability is added in `Scientist`, then `Scientist` and all its descendants are cloneable. Notice that `Scientist`'s `clone()` returns a `Scientist`, whereas cloning `MadScientist`, which inherits `Scientist`'s `clone()` rather than creating its own specific version, requires a cast.

Why This Strange Design?

If all this seems a strange scheme, that's because it is. You might wonder why it worked out this way. What is the meaning behind this design?

Originally, Java was designed as a language to control hardware boxes, and definitely not with the Internet in mind. In a general-purpose language like this, it makes sense for the programmer to clone any object. Thus, `clone()` was placed in the root class `Object`, *but* it was a `public` method so you can always clone any object. This seemed the most flexible approach, and after all, what could it hurt?

When Java was seen as an Internet programming language, things changed. Suddenly, there are security issues, and these issues are dealt with using objects, and you don't necessarily want anyone to clone your security objects. So what you're seeing is many patches applied on the original simple and straightforward scheme: `clone()` is now `protected` in `Object`. You must override it *and* `implement Cloneable` *and* deal with the exceptions.

It's worth noting you must implement the `Cloneable` interface *only* if you're going to call `Object`'s `clone()` method because that method checks at run time to make sure that your class implements `Cloneable`.

Controlling Cloneability

You might suggest that, to remove cloneability, the `clone()` method should simply be made `private`, but this won't work, because you cannot take a base-class method and make it less accessible in a derived class. And yet, it's necessary to control whether an object can be cloned. There are a number of attitudes you can take for your classes:

1. Indifference. You don't do anything about cloning, which means your class can't be cloned, but a class that inherits from you can add cloning if it wants. This works only if the default `Object.clone()` will do something reasonable with all the fields in your class.
2. Support `clone()`. Implement `Cloneable` and override `clone()`. In the overridden `clone()`, call `super.clone()` and catch all exceptions (so your overridden `clone()` doesn't throw any exceptions).

3. Support cloning conditionally. If your class holds references to other objects that might or might not be cloneable (a container class, for example), your `clone()` can try to clone all objects for which you have references, and if they throw exceptions, just pass those exceptions out to the programmer. For example, consider a special sort of `ArrayList` that tries to clone all the objects it holds. When you write such an `ArrayList`, you don't know what sort of objects the client programmer might put into your `ArrayList`, so you don't know whether they can be cloned.
4. Don't implement `Cloneable` but override `clone()` as `protected`, producing the correct copying behavior for any fields. This way, anyone inheriting from this class can override `clone()` and call `super.clone()` to produce the correct copying behavior. Note that your implementation can and should invoke `super.clone()` even though that method expects a `Cloneable` object (it will throw an exception otherwise), because no one will directly invoke it on an object of your type. It will get invoked only through a derived class, which, if it is to work successfully, implements `Cloneable`.
5. Try to prevent cloning by not implementing `Cloneable` and overriding `clone()` to throw an exception. This is successful only if any class derived from this calls `super.clone()` in its redefinition of `clone()`. Otherwise, a programmer might circumvent it.
6. Prevent cloning by making your class `final`. If `clone()` has not been overridden by any of your ancestor classes, then it can't be. If it has been overridden, then override it again and throw `CloneNotSupportedException`. Making the class `final` is the only way to guarantee that cloning is prevented. In addition, when dealing with security objects or other situations where you control the number of objects created, make all constructors `private` and provide one or more special methods for creating objects. That way, these methods can restrict the number of objects created and the conditions in which they're created.

Here's an example that shows the various ways cloning can be implemented and then, later in the hierarchy, "turned off":

```
// references/CheckCloneable.java
// Check to see if a reference can be cloned

// Can't clone this -- doesn't override clone():
class Ordinary {}

// Overrides clone, doesn't implement Cloneable:
class WrongClone extends Ordinary {
  @Override public Object clone()
  throws CloneNotSupportedException {
    return super.clone(); // Throws exception
  }
}

// Does all the right things for cloning:
class IsCloneable extends Ordinary
implements Cloneable {
  @Override public Object clone()
  throws CloneNotSupportedException {
    return super.clone();
  }
}

// Turn off cloning by throwing the exception:
class NoMore extends IsCloneable {
  @Override public Object clone()
  throws CloneNotSupportedException {
    throw new CloneNotSupportedException();
  }
}

class TryMore extends NoMore {
  @Override public Object clone()
  throws CloneNotSupportedException {
    // Calls NoMore.clone(), throws exception:
    return super.clone();
  }
}

class BackOn extends NoMore {
  private BackOn duplicate(BackOn b) {
    // Somehow make a copy of b and return that
```

```
    // copy. A dummy copy, just to make a point:
    return new BackOn();
  }
  @Override public Object clone() {
    // Doesn't call NoMore.clone():
    return duplicate(this);
  }
}

// You can't inherit from this, so you can't
// override clone() as you can in BackOn:
final class ReallyNoMore extends NoMore {}

public class CheckCloneable {
  public static
  Ordinary tryToClone(Ordinary ord) {
    String id = ord.getClass().getName();
    System.out.println("Attempting " + id);
    Ordinary x = null;
    if(ord instanceof Cloneable) {
      try {
        x = (Ordinary)((IsCloneable)ord).clone();
        System.out.println("Cloned " + id);
      } catch(CloneNotSupportedException e) {
        System.out.println(
          "Could not clone " + id);
      }
    } else {
      System.out.println("Doesn't implement Cloneable");
    }
    return x;
  }
  public static void main(String[] args) {
    // Upcasting:
    Ordinary[] ord = {
      new IsCloneable(),
      new WrongClone(),
      new NoMore(),
      new TryMore(),
      new BackOn(),
      new ReallyNoMore(),
    };
```

```
    Ordinary x = new Ordinary();
    // This won't compile because
    // clone() is protected in Object:
    //- x = (Ordinary)x.clone();
    // Checks first to see if the class
    // implements Cloneable:
    for(Ordinary ord1 : ord) {
      tryToClone(ord1);
    }
  }
}
/* Output:
Attempting IsCloneable
Cloned IsCloneable
Attempting WrongClone
Doesn't implement Cloneable
Attempting NoMore
Could not clone NoMore
Attempting TryMore
Could not clone TryMore
Attempting BackOn
Cloned BackOn
Attempting ReallyNoMore
Could not clone ReallyNoMore
*/
```

The first class, `Ordinary`, represents the kinds of classes we've seen throughout this book: no support for cloning, but as it turns out, no prevention of cloning either. But if you have a reference to an `Ordinary` object that might be upcast from a more derived class, you can't tell if it can be cloned or not.

The class `WrongClone` shows an incorrect way to implement cloning. It does override `Object.clone()` and makes that method `public`, but it doesn't implement `Cloneable`, so when `super.clone()` is called (which results in a call to `Object.clone()`), `CloneNotSupportedException` is thrown, so the cloning doesn't work.

`IsCloneable` performs all the right actions for cloning; `clone()` is overridden and `Cloneable` is implemented. However, this `clone()` method and several others that follow in this example *do not* catch `CloneNotSupportedException`, but instead pass it through to the caller, who must then put a try-catch block around it. In your own `clone()` methods you typically catch `CloneNotSupportedException` *inside*

`clone()` rather than passing it through. As you'll see, in this example it's more informative to pass the exceptions through.

Class `NoMore` attempts to "turn off" cloning in the way that the Java designers intended: in the derived class `clone()`, you throw `CloneNotSupportedException`. The `clone()` method in class `TryMore` properly calls `super.clone()`, and this resolves to `NoMore.clone()`, which throws an exception and prevents cloning.

But what if the programmer doesn't follow the "proper" path of calling `super.clone()` inside the overridden `clone()` method? In `BackOn`, you see how this can happen. This class uses a separate method `duplicate()` to make a copy of the current object and calls this method inside `clone()` *instead* of calling `super.clone()`. The exception is never thrown and the new class is cloneable. You can't rely on throwing an exception to prevent making a cloneable class. The only sure-fire solution is shown in `ReallyNoMore`, which is `final` and thus cannot be inherited. That means if `clone()` throws an exception in the `final` class, it cannot be modified with inheritance, and the prevention of cloning is assured. (You cannot explicitly call `Object.clone()` from a class that has an arbitrary level of inheritance; you are limited to calling `super.clone()`, which has access to only the direct base class.) Thus, if you make any objects that involve security issues, make those classes `final`.

The first method you see in class `CheckCloneable` is `tryToClone()`, which takes any `Ordinary` object and checks to see whether it's cloneable with `instanceof`. If so, it casts the object to an `IsCloneable`, calls `clone()`, and casts the result back to `Ordinary`, catching any exceptions. Notice the use of reflection (see the Reflection chapter) to display the class name and show what's happening.

In `main()`, different types of `Ordinary` objects are created and upcast to `Ordinary` in the array definition. The subsequent two lines of code create a plain `Ordinary` object and try to clone it. However, this code will not compile because `clone()` is a `protected` method in `Object`. The remainder of the code steps through the array and tries to clone each object, reporting the success or failure of each.

So to summarize, if you want a cloneable class:

1. Implement the `Cloneable` interface.
2. Override `clone()`.
3. Call `super.clone()` inside your `clone()`.
4. Capture exceptions inside your `clone()`.

This will produce the most convenient effects.

The Copy Constructor

Cloning can be a complicated process to set up. Is there an alternative? One (slow) approach is to use serialization, as shown earlier. You can also make a special constructor whose job it is to duplicate an object. In C++, this is called the *copy constructor*. A first attempt seems like it should work, but it doesn't:

```
// references/CopyConstructor.java
// A constructor to copy an object of the same
// type, as an attempt to create a local copy
import java.lang.reflect.*;

class FruitQualities {
  private int weight;
  private int color;
  private int firmness;
  private int ripeness;
  private int smell;
  // etc.
  // Zero-argument constructor:
  FruitQualities() {
    // Do something meaningful...
  }
  // Other constructors:
  // ...
  // Copy constructor:
  FruitQualities(FruitQualities f) {
    weight = f.weight;
    color = f.color;
    firmness = f.firmness;
    ripeness = f.ripeness;
    smell = f.smell;
    // etc.
  }
}

class Seed {
  // Members...
  Seed() { /* Zero-argument constructor */ }
  Seed(Seed s) { /* Copy constructor */ }
}
```

```
class Fruit {
  private FruitQualities fq;
  private int seeds;
  private Seed[] s;
  Fruit(FruitQualities q, int seedCount) {
    fq = q;
    seeds = seedCount;
    s = new Seed[seeds];
    for(int i = 0; i < seeds; i++)
      s[i] = new Seed();
  }
  // Other constructors:
  // ...
  // Copy constructor:
  Fruit(Fruit f) {
    fq = new FruitQualities(f.fq);
    seeds = f.seeds;
    s = new Seed[seeds];
    // Call all Seed copy-constructors:
    for(int i = 0; i < seeds; i++)
      s[i] = new Seed(f.s[i]);
    // Other copy-construction activities...
  }
  // This allows derived constructors (or other
  // methods) to put in different qualities:
  protected void addQualities(FruitQualities q) {
    fq = q;
  }
  protected FruitQualities getQualities() {
    return fq;
  }
}

class Tomato extends Fruit {
  Tomato() {
    super(new FruitQualities(), 100);
  }
  Tomato(Tomato t) { // Copy-constructor
    super(t); // Upcast to base copy-constructor
    // Other copy-construction activities...
  }
```

```
}

class ZebraQualities extends FruitQualities {
  private int stripedness;
  // Zero-argument constructor:
  ZebraQualities() {
    super();
    // do something meaningful...
  }
  ZebraQualities(ZebraQualities z) {
    super(z);
    stripedness = z.stripedness;
  }
}

class GreenZebra extends Tomato {
  GreenZebra() {
    addQualities(new ZebraQualities());
  }
  GreenZebra(GreenZebra g) {
    super(g); // Calls Tomato(Tomato)
    // Restore the right qualities:
    addQualities(new ZebraQualities());
  }
  public void evaluate() {
    ZebraQualities zq =
      (ZebraQualities)getQualities();
    // Do something with the qualities
    // ...
  }
}

public class CopyConstructor {
  public static void ripen(Tomato t) {
    // Use the "copy constructor":
    t = new Tomato(t);                          // [1]
    System.out.println("In ripen, t is a " +
      t.getClass().getName());
  }
  public static void slice(Fruit f) {
    f = new Fruit(f); // Hmm... will this work? // [2]
    System.out.println("In slice, f is a " +
```

```
      f.getClass().getName());
  }
  @SuppressWarnings("unchecked")
  public static void ripen2(Tomato t) {
    try {
      Class c = t.getClass();
      // Use the "copy constructor":
      Constructor ct =
        c.getConstructor(new Class[] { c });
      Object obj =
        ct.newInstance(new Object[] { t });
      System.out.println("In ripen2, t is a " +
        obj.getClass().getName());
    } catch(NoSuchMethodException |
            SecurityException |
            InstantiationException |
            IllegalAccessException |
            IllegalArgumentException |
            InvocationTargetException e) {
      System.out.println(e);
    }
  }
  @SuppressWarnings("unchecked")
  public static void slice2(Fruit f) {
    try {
      Class c = f.getClass();
      Constructor ct =
        c.getConstructor(new Class[] { c });
      Object obj =
        ct.newInstance(new Object[] { f });
      System.out.println("In slice2, f is a " +
        obj.getClass().getName());
    } catch(NoSuchMethodException |
            SecurityException |
            InstantiationException |
            IllegalAccessException |
            IllegalArgumentException |
            InvocationTargetException e) {
      System.out.println(e);
    }
  }
  public static void main(String[] args) {
```

```
    Tomato tomato = new Tomato();
    ripen(tomato); // OK
    slice(tomato); // OOPS!
    ripen2(tomato); // OK
    slice2(tomato); // OK
    GreenZebra g = new GreenZebra();
    ripen(g); // OOPS!
    slice(g); // OOPS!
    ripen2(g); // OK
    slice2(g); // OK
    g.evaluate();
  }
}
/* Output:
In ripen, t is a Tomato
In slice, f is a Fruit
java.lang.NoSuchMethodException: Tomato.<init>(Tomato)
java.lang.NoSuchMethodException: Tomato.<init>(Tomato)
In ripen, t is a Tomato
In slice, f is a Fruit
java.lang.NoSuchMethodException:
GreenZebra.<init>(GreenZebra)
java.lang.NoSuchMethodException:
GreenZebra.<init>(GreenZebra)
*/
```

This seems a bit strange at first. Sure, fruit has qualities, but why not just put fields representing those qualities directly into the `Fruit` class? There are two potential reasons.

1. To easily insert or change the qualities. Note that `Fruit` has a `protected addQualities()` method to allow derived classes to do this. (You might think the logical thing to do is include a `protected` constructor in `Fruit` that takes a `FruitQualities` argument, but constructors don't inherit, so it wouldn't be available in second or greater level classes.) By making the fruit qualities into a separate class and using composition, you have greater flexibility, including the ability to change the qualities midway through the lifetime of a particular `Fruit` object.
2. Making `FruitQualities` a separate object allows you to add new qualities or to change the behavior via inheritance and polymorphism. Note that for

`GreenZebra` (which *really is* a type of tomato), the constructor calls `addQualities()` and passes it a `ZebraQualities` object, which is derived from `FruitQualities`, so it can be attached to the `FruitQualities` reference in the base class. When `GreenZebra` uses the `FruitQualities`, it must downcast it to the correct type (as seen in `evaluate()`), but it always knows that type is `ZebraQualities`.

You'll also see there's a `Seed` class, and that `Fruit` (which by definition carries its own seeds) contains an array of `Seeds`.

Finally, notice that each class has a copy constructor, and that each copy constructor must take care to call the copy constructors for the base class and member objects to produce a deep copy. The copy constructor is tested inside the class `CopyConstructor`.

- **[1]** `ripen()` takes a `Tomato` argument and performs copy-construction on it to duplicate the object.
- **[2]** `slice()` takes a more generic `Fruit` object and also duplicates it.

These are tested with different kinds of `Fruit` in `main()`. The output shows the problem. After the copy-construction that happens to the `Tomato` inside `slice()`, the result is no longer a `Tomato` object, but just a `Fruit`. It has lost all of its tomatoness. Furthermore, when you take a `GreenZebra`, both `ripen()` and `slice()` turn it into a `Tomato` and a `Fruit`, respectively. Thus, unfortunately, the copy constructor scheme is no good to us in Java when attempting to make a local copy of an object.

Why Does It Work in C++ and Not Java?

The copy constructor is a fundamental part of C++ because it automatically makes a local copy of an object. Yet the preceding example proves it does not work for Java. Why? In Java, everything that we manipulate is a reference, but in C++, you can have reference-like entities and you can *also* pass objects around directly. That's what the C++ copy constructor is for: to take an object and pass it in by value, thus duplicating the object. So it works fine in C++, but keep in mind this scheme fails in Java, so don't use it.

Immutable Classes

Although the local copy produced by `clone()` gives the desired results in the appropriate cases, it is an example of forcing the programmer (the author of the method) to be responsible for preventing the ill effects of aliasing. What if you're making a library that's so general purpose and commonly used you cannot make the assumption it will always be cloned in the proper places? Or more likely, how do you allow aliasing for efficiency—to prevent the needless duplication of objects—without the negative side effects of aliasing?

One solution (used by pure functional programming languages) is to create *immutable objects* that belong to read-only classes. You can define a class such that no methods in the class change the internal state of the object. In such a class, aliasing has no impact because you can only read the internal state, so if many pieces of code are reading the same object, there's no problem.

As a simple example of immutable objects, Java's standard library contains "wrapper" classes for all the primitive types. You might have already discovered that, to store an `int` inside a container such as an `ArrayList` (which takes only `Object` references), you must wrap your `int` inside the standard library `Integer` class. Here, the wrapping occurs automatically, with autoboxing:

```
// references/ImmutableInteger.java
// The Integer class cannot be changed
import java.util.*;
import java.util.stream.*;

public class ImmutableInteger {
  public static void main(String[] args) {
    @SuppressWarnings("deprecation")
    List<Integer> v = IntStream.range(0, 10)
      .mapToObj(Integer::new)
      .collect(Collectors.toList());
    System.out.println(v);
    // But how do you change the int
    // inside the Integer?
  }
}
/* Output:
```

```
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
*/
```

The `Integer` class (as well as all the primitive "wrapper" classes) implements immutability in a simple fashion: It has no methods that allow you to change the object.

If you do need an object to hold a primitive type that can be modified, you must create it yourself. Fortunately, this is trivial:

```
// references/MutableInteger.java
// A changeable wrapper class
import java.util.*;
import java.util.stream.*;

class IntValue {
  private int n;
  IntValue(int x) { n = x; }
  public int getValue() { return n; }
  public void setValue(int n) { this.n = n; }
  public void increment() { n++; }
  @Override public String toString() {
    return Integer.toString(n);
  }
}

public class MutableInteger {
  public static void main(String[] args) {
    List<IntValue> v = IntStream.range(0, 10)
      .mapToObj(IntValue::new)
      .collect(Collectors.toList());
    System.out.println(v);
    v.forEach(IntValue::increment);
    System.out.println(v);
  }
}
/* Output:
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
[1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
*/
```

`IntValue` can be even simpler if privacy is not an issue:

```
// references/SimplerMutableInteger.java
// A trivial wrapper class
import java.util.*;
import java.util.stream.*;

class IntValue2 {
  public int n;
  IntValue2(int n) { this.n = n; }
}

public class SimplerMutableInteger {
  public static void main(String[] args) {
    List<IntValue2> v = IntStream.range(0, 10)
      .mapToObj(IntValue2::new)
      .collect(Collectors.toList());
    v.forEach(iv2 ->
      System.out.print(iv2.n + " "));
    System.out.println();
    v.forEach(iv2 -> iv2.n += 1);
    v.forEach(iv2 ->
      System.out.print(iv2.n + " "));
  }
}
/* Output:
0 1 2 3 4 5 6 7 8 9
1 2 3 4 5 6 7 8 9 10
*/
```

Directly selecting the `n` member is a bit awkward, however.

Creating Immutable Classes

Here's one way to create your own immutable class:

```
// references/Immutable1.java
// Immutable objects are immune to aliasing

public class Immutable1 {
  private int data;
  public Immutable1(int initVal) {
    data = initVal;
  }
  public int read() { return data; }
  public boolean nonzero() { return data != 0; }
  public Immutable1 multiply(int multiplier) {
    return new Immutable1(data * multiplier);
  }
  public static void f(Immutable1 i1) {
    Immutable1 quad = i1.multiply(4);
    System.out.println("i1 = " + i1.read());
    System.out.println("quad = " + quad.read());
  }
  public static void main(String[] args) {
    Immutable1 x = new Immutable1(47);
    System.out.println("x = " + x.read());
    f(x);
    System.out.println("x = " + x.read());
  }
}
/* Output:
x = 47
i1 = 47
quad = 188
x = 47
*/
```

All data is `private`, and you'll see that none of the `public` methods modify that data. Indeed, the method that does appear to modify an object is `multiply()`, but this creates a new `Immutable1` object and leaves the original untouched.

The method `f()` takes an `Immutable1` object and performs various operations on it, and the output of `main()` demonstrates there is no change to `x`. Thus, `x`s object could be aliased many times without harm, because the `Immutable1` class is designed to guarantee that objects cannot be changed.

The Drawback to Immutability

Creating an immutable class seems at first to provide an elegant solution. However, whenever you do need a modified object of that new type, you must suffer the overhead of a new object creation, as well as potentially causing more frequent garbage collections. For some classes this is not a problem (and functional programming languages rely on it), but for others (such as the `String` class) it can be expensive—but remember always the dictum against premature optimization. (Note that languages that only provide immutability do quite well, despite the perceived overhead).

The solution is to create a companion class that *can* be modified. Then, when you're doing many modifications, you can switch to using the modifiable companion class and switch back to the immutable class when you're done.

We can change `Immutable1.java` to show this:

```
// references/Immutable2.java
// A companion class to modify immutable objects

class Mutable {
  private int data;
  Mutable(int initVal) {
    data = initVal;
  }
  public Mutable add(int x) {
    data += x;
    return this;
  }
  public Mutable multiply(int x) {
    data *= x;
    return this;
  }
  public Immutable2 makeImmutable2() {
    return new Immutable2(data);
  }
}

public class Immutable2 {
  private int data;
  public Immutable2(int initVal) {
    data = initVal;
```

```
  }
  public int read() { return data; }
  public boolean nonzero() {
    return data != 0;
  }
  public Immutable2 add(int x) {
    return new Immutable2(data + x);
  }
  public Immutable2 multiply(int x) {
    return new Immutable2(data * x);
  }
  public Mutable makeMutable() {
    return new Mutable(data);
  }
  public static
  Immutable2 modify1(Immutable2 y) {
    Immutable2 val = y.add(12);
    val = val.multiply(3);
    val = val.add(11);
    val = val.multiply(2);
    return val;
  }
  // This produces the same result:
  public static
  Immutable2 modify2(Immutable2 y) {
    Mutable m = y.makeMutable();
    m.add(12).multiply(3).add(11).multiply(2);
    return m.makeImmutable2();
  }
  public static void main(String[] args) {
    Immutable2 i2 = new Immutable2(47);
    Immutable2 r1 = modify1(i2);
    Immutable2 r2 = modify2(i2);
    System.out.println("i2 = " + i2.read());
    System.out.println("r1 = " + r1.read());
    System.out.println("r2 = " + r2.read());
  }
}
/* Output:
i2 = 47
r1 = 376
r2 = 376
```

```
*/
```

`Immutable2` contains methods that, as before, preserve the immutability of the objects by producing new objects whenever a modification is desired. These are the `add()` and `multiply()` methods. The companion class is called `Mutable`, and it also has `add()` and `multiply()` methods, but these modify the `Mutable` object rather than making a new one. In addition, `Mutable` has a method to use its data to produce an `Immutable2` object and vice versa.

The two static methods `modify1()` and `modify2()` show two different approaches to producing the same result. In `modify1()`, everything is done within the `Immutable2` class and you see that four new `Immutable2` objects are created in the process. (And each time `val` is reassigned, the previous object becomes garbage.)

The first action in `modify2()` is to take the `Immutable2 y` and produce a `Mutable` from it. (This is just like calling `clone()` as you saw earlier, but this time a different type of object is created.) Then the `Mutable` object is used to perform many change operations *without* requiring the creation of many new objects. Finally, it's turned back into an `Immutable2`. Here, two new objects are created (the `Mutable` and the result `Immutable2`) instead of four.

This approach makes sense, then, when:

1. You need immutable objects and
2. You often need to make many modifications or
3. It's expensive to create new immutable objects.

Immutable `String`s

Consider the following code:

```
// references/Stringer.java

public class Stringer {
  public static String upcase(String s) {
    return s.toUpperCase();
  }
  public static void main(String[] args) {
    String q = new String("howdy");
    System.out.println(q); // howdy
    String qq = upcase(q);
    System.out.println(qq); // HOWDY
    System.out.println(q); // howdy
  }
}
/* Output:
howdy
HOWDY
howdy
*/
```

When `q` is passed in to `upcase()` it's actually a copy of the reference to `q`. The object this reference is connected to stays in a single physical location. The references are copied as they are passed around.

In the definition for `upcase()`, the reference that's passed in has the name `s`, and it exists for only as long as the body of `upcase()` is executed. When `upcase()` completes, the local reference `s` vanishes. `upcase()` returns the result, which is the original `String` with all the characters set to uppercase. Of course, it actually returns a reference to the result. But the reference is for a new object, and the original `q` is left alone. How does this happen?

Implicit Constants

If you say:

```
String s = "asdf";
String x = Stringer.upcase(s);
```

do you really want the `upcase()` method to *change* the argument? In general, you don't, because an argument usually looks to the reader of the code as a piece of information provided to the method, not something to be modified. This is an important guarantee because it makes code easier to write and understand.

Overloading + and StringBuilder

Objects of the String class are designed to be immutable, using the companion-class technique shown previously. If you examine the JDK documentation for the String class (which is summarized a little later in this appendix), you'll see that every method in the class that appears to modify a String really creates and returns a brand new String object containing the modification. The original String is left untouched. Thus, there's no feature in Java like C++'s const to make the compiler support the immutability of your objects. If you want it, you have to wire it in yourself, like String does.

String objects are immutable so you can alias to a particular String as many times as you want. Immutability means there's no chance one reference will change something that affects other references. So an immutable object solves the aliasing problem nicely.

It also seems possible to handle all the cases where you need a modified object by creating a brand new version of the object with the modifications, as String does. However, for some operations this isn't efficient. A case in point is the operator + that is overloaded for String objects. Overloading means it has extra meaning when used with a particular class. (The + and += for String are the only overloaded operators in Java, and Java does not allow the programmer to overload any others).[190]

When used with String objects, the + concatenates Strings together:

```
String s = "abc" + foo + "def" + Integer.toString(47);
```

You can imagine how this *might* work. The String "abc" could have a method append() that creates a new String object containing "abc" concatenated with the contents of foo. The new String object would then create another new String that added "def," and so on.

This would certainly work, but it requires the creation of many String objects just to put together this new String, then you have a bunch of intermediate String objects that need garbage-collecting. I suspect that the Java designers tried this approach

[190]C++ allows the programmer to overload operators at will. Because this can often be a complicated process (see Chapter 10 of *Thinking in C++, 2nd edition*, Prentice Hall, 2000), the Java designers deemed it a "bad" feature that shouldn't be included in Java. It wasn't so bad they didn't end up doing it themselves, and ironically enough, operator overloading would be much easier to use in Java than in C++. This can be seen in Python (see www.Python.org) which has garbage collection and straightforward operator overloading.

first (which is a lesson in software design—you don't really know anything about a system until you try it out in code and get something working). I also suspect they discovered it delivered unacceptable performance.

The solution is a mutable companion class similar to the one shown previously. For `String`, this companion class is called `StringBuilder`, and the compiler automatically creates a `StringBuilder` to evaluate certain expressions, in particular when the overloaded operators `+` and `+=` are used with `String` objects. Here's what it looks like:

```
// references/ImmutableStrings.java
// Demonstrating StringBuilder

public class ImmutableStrings {
  public static void main(String[] args) {
    String foo = "foo";
    String s = "abc" + foo + "def"
      + Integer.toString(47);
    System.out.println(s);
    // The "equivalent" using StringBuilder:
    StringBuilder sb =
      new StringBuilder("abc"); // Creates String
    sb.append(foo);
    sb.append("def"); // Creates String
    sb.append(Integer.toString(47));
    System.out.println(sb);
  }
}
/* Output:
abcfoodef47
abcfoodef47
*/
```

In the creation of `String s`, the compiler is doing the rough equivalent of the subsequent code that uses `sb`: a `StringBuilder` is created, and `append()` is used to add new characters directly into the `StringBuilder` object (rather than making new copies each time). While this is more efficient, it's worth noting that each time you create a quoted character string such as `"abc"` and `"def"`, the compiler turns those into `String` objects. This can create more objects than you expect, despite the efficiency afforded through `StringBuilder`.

You can find more details of `StringBuilder` in the Strings chapter.

`String`s are Special

If you review the Strings chapter, you'll see that every `String` method carefully returns a new `String` object when the contents change. If the contents don't need changing, the method just returns a reference to the original `String`. This saves storage and overhead.

The `String` class is not just another class in Java. There are many special cases in `String`, not the least of which is that it's a built-in class and fundamental to Java. Then there's the fact that a quoted character string is converted to a `String` by the compiler and the special overloaded operators + and +=. In this appendix you've seen the remaining special case: the carefully-built immutability using the companion `StringBuilder` and some extra magic in the compiler.

Summary

Because all object identifiers are references in Java, and because every object is created on the heap and garbage collected only when it is no longer used, the flavor of object manipulation changes, especially when passing and returning objects. To initialize some piece of storage in a method in C or C++, you might request that the user pass the address of that piece of storage into the method. Otherwise, you have to worry about who is responsible for destroying that storage. Thus, the interface and understanding of such methods is more complicated. But in Java, you don't worry about responsibility or whether an object will still exist when it is needed because that is always handled for you. You create an object when it is needed (and no sooner) and never worry about mechanics of passing around responsibility for that object; you simply pass the reference. Sometimes the simplification this provides is unnoticed. Other times it is staggering.

The downside to all this underlying magic is twofold:

1. You always take the efficiency hit for the extra memory management (although this is usually a non-issue), and there's always a slight amount of uncertainty about the time something can take to run (the garbage collector can be forced into action whenever you get low on memory). For most applications, the benefits outweigh the drawbacks, and the hotspot technologies in particular have sped things up to the point where it's not much of an issue.

2. Aliasing: Sometimes you end up with two references to the same object, which is a problem only if both references are assumed to point to a *distinct* object. This is where you must pay a little closer attention and, when necessary, `clone()` or otherwise duplicate an object to prevent the other reference from being surprised by an unexpected change. Alternatively, you can support aliasing for efficiency by creating immutable objects whose operations can return a new object of the same type or some different type, but never change the original object so anyone aliased to that object sees no change.

Some people say that cloning in Java is a botched design that shouldn't be used, so they implement their own version of cloning. Doug Lea, who was helpful in resolving this issue, suggested this to me, saying that he simply creates a function called `duplicate()` for each class. This way, you never call `Object.clone()`, eliminating the need to implement `Cloneable` and catch the `CloneNotSupportedException`. This is certainly a reasonable approach, and because `clone()` is supported so rarely within the standard Java library, it is apparently a safe one as well.

Rather than writing your own clone support, consider either the *Apache Commons Serialization Utility Classes* or the deep cloning library[191].

[191] https://github.com/kostaskougios/cloning

Appendix: I/O Streams

> Java 7 introduced a simple and clear approach for reading and writing files and using directory paths. Most of the time, the libraries and techniques shown in the Files chapter are all you need. If, however, you must deal with specific needs, lower-level operations or with legacy code, you must understand the information in this appendix.

Creating a good input/output (I/O) system is one of the more difficult tasks for a language designer. This is evidenced by the number of different approaches. The challenge seems to be in covering all possibilities. Not only are there different sources and sinks of I/O (files, the console, network connections, etc.), but you must talk to them in a wide variety of ways (sequential, random-access, buffered, binary, character, by lines, by words, etc.).

The Java library designers attacked this problem by creating lots of classes. In fact, there are so many classes in Java I/O streams it can be intimidating at first. There was also a significant change in the I/O library after Java 1.0, when the original byte-oriented library was supplemented with `char`-oriented, Unicode-based I/O classes. The `nio` classes (for "new I/O," a name used years after they were introduced in Java 1.4) were added for improved performance and functionality; these are covered in the Appendix: New I/O.

As a result, there are a fair number of classes to learn before you understand enough of Java I/O streams library to use it properly. It's also helpful to understand the evolution of the I/O library. The problem is that, without the historical perspective, you rapidly become confused with some of the classes and when you should and shouldn't use them.

Programming language I/O libraries often use the abstraction of a *stream*, which represents any data source or sink as an object capable of producing or receiving pieces of data.

> It's important to understand there is *no* connection between the Java 8 functional-style `Stream` classes and I/O Streams. This is another example

> where, if the designers could do it over again, they'd use different terminology.

An I/O stream hides details of what happens to the data inside the actual I/O device:

1. Byte Streams are for raw binary data.
2. Character Streams are for character data. These automatically handle translation to and from the local character set.
3. Buffered Streams improve performance. They optimize input and output by reducing the number of calls to the native API.

The Java library classes for I/O streams are divided by input and output, as shown in the class hierarchy in the JDK documentation. In Java 1.0, the library designers decided that all classes that had anything to do with input are inherited from `InputStream`, and all classes associated with output are inherited from `OutputStream`. Everything derived from the `InputStream` or `Reader` classes has basic methods called `read()` for reading a single `byte` or an array of `bytes`. Likewise, everything derived from `OutputStream` or `Writer` classes has basic methods called `write()` for writing a single `byte` or an array of `bytes`. However, you won't generally use these methods; they exist so other classes can use them—these other classes provide a more useful interface.

You'll rarely create your stream object by using a single class, but instead will layer multiple objects together to provide your desired functionality (this is the *Decorator* design pattern). The fact you create more than one object to produce a single stream is the primary reason that Java's I/O library is confusing.

I attempt to provide an overview of the classes here, but assume you will use the JDK documentation to determine all the details, such as the exhaustive list of methods for a particular class.

Types of `InputStream`

`InputStream` represents classes that produce input from different sources. These sources can be:

1. An array of bytes.

2. A `String` object.
3. A file.
4. A "pipe," which works like a physical pipe: You put things in at one end and they come out the other.
5. A sequence of other streams, so you can collect them together into a single stream.
6. Other sources, such as an Internet connection.

Each of these sources has an associated subclass of `InputStream`. In addition, the `FilterInputStream` is also a type of `InputStream`, to provide a base class for "decorator" classes that attach attributes or useful interfaces to input streams. This is discussed later.

Table I/O-1: Types of InputStream

Class	Function	Constructor arguments	How to Use It
`ByteArray-InputStream`	Allows a buffer in memory to act as an `InputStream`.	The buffer from which to extract the bytes.	As a source of data: Connect it to a `FilterInputStream` object to provide a useful interface.
`StringBuffer-InputStream`	Converts a `String` into an `InputStream`.	A `String`. The underlying implementation actually uses a `StringBuffer`.	As a source of data: Connect it to a `FilterInputStream` object to provide a useful interface.
`FileInputStream`	For reading information from a file.	A `String` representing the file name, or a `File` or `FileDescriptor` object.	As a source of data: Connect it to a `FilterInputStream` object to provide a useful interface.

Table I/O-1: Types of InputStream

Class	Function	Constructor arguments	How to Use It
`PipedInputStream`	Produces the data that's written to the associated `PipedOutputStream`. Implements the "piping" concept.	`PipedOutputStream`	As a source of data in multithreading: Connect it to a `FilterInputStream` object to provide a useful interface.
`SequenceInputStream`	Converts two or more `InputStream` objects into a single `InputStream`.	Two `InputStream` objects or an `Enumeration` for a container of `InputStream` objects.	As a source of data: Connect it to a `FilterInputStream` object to provide a useful interface.
`FilterInputStream`	Abstract class that is an interface for decorators that provide useful functionality to the other `InputStream` classes. See **Table I/O-3**.	See **Table I/O-3**.	See **Table I/O-3**.

Types of `OutputStream`

This category includes the classes that decide where your output will go: an array of bytes (but not a `String`—presumably, you can create one using the array of bytes), a file, or a "pipe."

In addition, the `FilterOutputStream` provides a base class for "decorator" classes that attach attributes or useful interfaces to output streams. This is discussed later.

Table I/O-2: Types of OutputStream

Class	Function	Constructor arguments	How to use it
`ByteArray-OutputStream`	Creates a buffer in memory. All the data you send to the stream is placed in this buffer.	Optional initial size of the buffer.	To designate the destination of your data: Connect it to a `FilterOutputStream` object to provide a useful interface.
`FileOutputStream`	For sending information to a file.	A `String` representing the file name, or a `File` or `FileDescriptor` object.	To designate the destination of your data: Connect it to a `FilterOutputStream` object to provide a useful interface.
`PipedOutputStream`	Any information you write to this automatically ends up as input for the associated `PipedInputStream`. Implements the "piping" concept.	`PipedInputStream`	To designate the destination of your data for multithreading: Connect it to a `FilterOutputStream` object to provide a useful interface.
`FilterOutputStream`	Abstract class that is an interface for decorators that provide useful functionality to the other `OutputStream` classes. See **Table I/O-4**.	See **Table I/O-4**.	See **Table I/O-4**.

Adding Attributes and Useful Interfaces

Decorators were introduced in the Generics chapter. The Java I/O library requires many different combinations of features, and this is the justification for using the Decorator design pattern.[192] The reason for the existence of the "filter" classes in the Java I/O library is that the abstract "filter" class is the base class for all the decorators. A decorator must have the same interface as the object it decorates, but the decorator can also extend the interface, which occurs in several of the "filter" classes.

There is a drawback to the Decorator pattern. Decorators provide more flexibility when writing a program (you can easily mix and match attributes), but they add complexity to your code. The reason that the Java I/O library is awkward to use is that you must create many classes—the "core" I/O type plus all the decorators—to get the single I/O object you want.

The classes that provide the decorator interface to control a particular `InputStream` or `OutputStream` are `FilterInputStream` and `FilterOutputStream`, which don't have very intuitive names. `FilterInputStream` and `FilterOutputStream` are derived from the base classes of the I/O library, `InputStream` and `OutputStream`, which is a key requirement of the decorator (so it provides the common interface to all decorated objects).

Reading from an `InputStream` with `FilterInputStream`

The `FilterInputStream` classes accomplish two notably different things. `DataInputStream` reads different types of primitive data as well as `String` objects. (All the methods start with "read," such as `readByte()`, `readFloat()`, etc.) This, along with its companion `DataOutputStream`, lets you move primitive data from one place to another via a stream. These "places" are determined by the classes in **Table I/O-1**.

The remaining `FilterInputStream` classes modify the way an `InputStream` behaves internally: whether it's buffered or unbuffered, whether it keeps track of the lines it's reading (allowing you to ask for line numbers or set the line number), and whether you can push back a single character. The last two classes look a lot like support for building a compiler (they were probably added to support the experiment

[192] It's not clear that this was a good design decision, especially compared to the simplicity of I/O libraries in other languages. But it's the justification for the decision.

of "building a Java compiler in Java"), so you probably won't use them in general programming.

You'll buffer your input almost every time, regardless of the connected I/O device, so it would have made more sense for the I/O library to provide a special case (or simply a method call) for unbuffered input rather than forcing you to add buffering almost every time.

Table I/O-3: Types of FilterInputStream

Class	Function	Constructor Arguments	How to use it
`DataInputStream`	Used in concert with `DataOutputStream`, to read primitives (`int`, `char`, `long`, etc.) from a stream in a portable fashion.	`InputStream`	Contains a full interface to allow you to read primitive types.
`Buffered-InputStream`	Use this to prevent a physical read every time you want more data. You're saying, "Use a buffer."	`InputStream`, with optional buffer size.	This doesn't provide an interface per se. It just adds buffering to the process. Attach an interface object.
`LineNumber-InputStream`	Keeps track of line numbers in the input stream; you can call `getLineNumber()` and `setLineNumber(int)`.	`InputStream`	This just adds line numbering, so you'll probably attach an interface object.

Table I/O-3: Types of FilterInputStream

Class	Function	Constructor Arguments	How to use it
`PushbackInputStream`	Has a one-byte push-back buffer to push back the last character read.	`InputStream`	Generally used in the scanner for a compiler. You probably won't use this.

Writing to an `OutputStream` with `FilterOutputStream`

The complement to `DataInputStream` is `DataOutputStream`, which formats each of the primitive types and `String` objects onto a stream in such a way that any `DataInputStream`, on any machine, can read them. All the methods start with "write," such as `writeByte()`, `writeFloat()`, etc.

The original intent of `PrintStream` was to print all primitive data types and `String` objects in a viewable format. This is different from `DataOutputStream`, whose goal is to put data elements on a stream in a way that `DataInputStream` can portably reconstruct them.

The two important methods in `PrintStream` are `print()` and `println()`, which are overloaded to print all the various types. The difference between `print()` and `println()` is that the latter adds a newline when it's done.

`PrintStream` can be problematic because it traps all `IOExceptions` (you must explicitly test the error status with `checkError()`, which returns `true` if an error has occurred). Also, `PrintStream` doesn't internationalize properly. These problems are solved with `PrintWriter`, described later.

`BufferedOutputStream` is a modifier and tells the stream to use buffering so you don't get a physical write every time you write to the stream. You'll probably always use this when doing output.

Table I/O-4: Types of FilterOutputStream

Class	Function	Constructor Arguments	How to use it
`DataOutputStream`	Used in concert with `DataInputStream` so you can write primitives (`int`, `char`, `long`, etc.) to a stream in a portable fashion.	`OutputStream`	Contains a full interface to allow you to write primitive types.
`PrintStream`	For producing formatted output. While `DataOutputStream` handles the *storage* of data, `PrintStream` handles *display*.	`OutputStream`, with optional `boolean` indicating that the buffer is flushed with every newline.	Should be the "final" wrapping for your `OutputStream` object. You'll probably use this a lot.
`Buffered-OutputStream`	Use this to prevent a physical write every time you send a piece of data. You're saying, "Use a buffer." You can call `flush()` to flush the buffer.	`OutputStream`, with optional buffer size.	This doesn't provide an interface per se. It just adds buffering to the process. Attach an interface object.

`Reader`s & `Writer`s

Java 1.1 made significant modifications to the fundamental I/O stream library. When you see the `Reader` and `Writer` classes, your first thought (like mine) might be that these were meant to replace the `InputStream` and `OutputStream` classes. But that's

not the case. Although some aspects of the original streams library are deprecated (if you use them you receive a warning from the compiler), the `InputStream` and `OutputStream` classes still provide valuable functionality in the form of byte-oriented I/O, whereas the `Reader` and `Writer` classes provide Unicode-compliant, character-based I/O. In addition:

1. Java 1.1 added new classes into the `InputStream` and `OutputStream` hierarchy, so it's obvious those hierarchies weren't replaced.
2. There are times when you must use classes from the "byte" hierarchy *in combination* with classes in the "character" hierarchy. To accomplish this, there are "adapter" classes: `InputStreamReader` converts an `InputStream` to a `Reader`, and `OutputStreamWriter` converts an `OutputStream` to a `Writer`.

The most important reason for the `Reader` and `Writer` hierarchies is for internationalization. The old I/O stream hierarchy supports only 8-bit byte streams and doesn't handle the 16-bit Unicode characters well. Unicode is used for internationalization (and Java's native `char` is 16-bit Unicode) so the `Reader` and `Writer` hierarchies were added to support Unicode in all I/O operations. In addition, the new libraries are designed for faster operations than the old.

Sources and Sinks of Data

Almost all the original Java I/O stream classes have corresponding `Reader` and `Writer` classes to provide native Unicode manipulation. However, there are some places where the byte-oriented `InputStreams` and `OutputStreams` are the correct solution; in particular, the `java.util.zip` libraries are byte-oriented rather than `char`-oriented. So the most sensible approach to take is to *try* to use the `Reader` and `Writer` classes whenever you can. You'll discover the problems when you use the byte-oriented libraries because your code won't compile.

This table shows the correspondence between the sources and sinks of information (that is, where the data physically comes from or goes to) in the two hierarchies.

Sources & sinks: Java 1.0 class	Corresponding Java 1.1 class
`InputStream`	`Reader` adapter: `InputStreamReader`
`OutputStream`	`Writer` adapter: `OutputStreamWriter`
`FileInputStream`	`FileReader`
`FileOutputStream`	`FileWriter`
`StringBufferInputStream` (deprecated)	`StringReader`
(No corresponding class)	`StringWriter`
`ByteArrayInputStream`	`CharArrayReader`
`ByteArrayOutputStream`	`CharArrayWriter`
`PipedInputStream`	`PipedReader`
`PipedOutputStream`	`PipedWriter`

In general, you'll find that the interfaces for the two different hierarchies are similar, if not identical.

Modifying Stream Behavior

For `InputStreams` and `OutputStreams`, streams were adapted for particular needs using "decorator" subclasses of `FilterInputStream` and `FilterOutputStream`. The `Reader` and `Writer` class hierarchies continue this idea—but not exactly.

In the following table, the correspondence is a rougher approximation than in the previous table. The difference is because of the class organization; although `BufferedOutputStream` is a subclass of `FilterOutputStream`, `BufferedWriter` is *not* a subclass of `FilterWriter` (which, even though it is `abstract`, has no subclasses and so appears be there as a placeholder or simply so you don't wonder where it is). However, the interfaces to the classes are a close match.

Filters: Java 1.0 class	Corresponding Java 1.1 class
`FilterInputStream`	`FilterReader`
`FilterOutputStream`	`FilterWriter` (abstract class with no subclasses)
`BufferedInputStream`	`BufferedReader` (also has `readLine()`)
`BufferedOutputStream`	`BufferedWriter`
`DataInputStream`	Use `DataInputStream` (except when you must use `readLine()`, when you should use a `BufferedReader`)
`PrintStream`	`PrintWriter`
`LineNumberInputStream` (deprecated)	`LineNumberReader`
`StreamTokenizer`	`StreamTokenizer` (Use the constructor that takes a `Reader` instead)
`PushbackInputStream`	`PushbackReader`

There's one clear limitation: Whenever you use `readLine()`, don't do it with a `DataInputStream` (this is met with a deprecation message at compile time), but instead use a `BufferedReader`. Other than this, `DataInputStream` is still a "preferred" member of the I/O library.

To make the transition to `PrintWriter` easier, it has constructors that take any `OutputStream` object as well as `Writer` objects. `PrintWriter`'s formatting interface is virtually the same as `PrintStream`.

In Java 5, `PrintWriter` constructors were added to simplify the creation of files when writing output, as you shall see shortly.

One `PrintWriter` constructor also has an option to perform automatic flushing, which happens after every `println()` if the constructor flag is set.

Unchanged Classes

Some classes were left unchanged between Java 1.0 and Java 1.1:

Java 1.0 classes without corresponding Java 1.1 classes
`DataOutputStream`
`File`
`RandomAccessFile`
`SequenceInputStream`

`DataOutputStream`, in particular, is used without change, so for storing and retrieving data in a transportable format, you use the `InputStream` and `OutputStream` hierarchies.

Off By Itself: `RandomAccessFile`

`RandomAccessFile` is used for files containing records of known size so you can move from one record to another using `seek()`, then read or change the records. The records don't have to be the same size; you just determine how big they are and where they are placed in the file.

At first it's a little bit hard to believe that `RandomAccessFile` is not part of the `InputStream` or `OutputStream` hierarchy. However, it has no association with those hierarchies other than it happens to implement the `DataInput` and `DataOutput` interfaces (which are also implemented by `DataInputStream` and `DataOutputStream`). It doesn't even use any of the functionality of the existing `InputStream` or `OutputStream` classes; it's a completely separate class, written from scratch, with all of its own (mostly native) methods. The reason for this might be that `RandomAccessFile` has essentially different behavior than the other I/O types because you can move forward and backward within a file. In any event, it stands alone, as a direct descendant of `Object`.

Essentially, a `RandomAccessFile` works like a `DataInputStream` combined with a `DataOutputStream`, along with the methods `getFilePointer()` to find out where you are in the file, `seek()` to move to a new point in the file, and `length()` to determine the maximum size of the file. In addition, the constructors require a second argument (identical to `fopen()` in C) indicating whether you are just randomly

reading (`"r"`) or reading and writing (`"rw"`). There's no support for write-only files, which could suggest that `RandomAccessFile` might have worked well if it were inherited from `DataInputStream`.

The seeking methods are available only in `RandomAccessFile`, which works for files only. `BufferedInputStream` does allow you to `mark()` a position (whose value is held in a single internal variable) and `reset()` to that position, but this is limited and not very useful.

Most, if not all, `RandomAccessFile` functionality is superseded as of Java 1.4 with the `nio` *memory-mapped files*, described in the Appendix: New I/O.

Typical Uses of I/O Streams

Although you can combine the I/O stream classes in many different ways, you'll probably just use a few combinations. The following examples can be used as a basic reference for typical I/O usage. (After making sure you can't do what you want using the libraries described in the Files chapter.)

In these examples, exception handing is simplified by passing exceptions out to the console, but this is appropriate only in small examples and utilities. In your code, consider more sophisticated error-handling approaches.

Buffered Input File

To open a file for character input, you use a `FileReader` with a `String` or a `File` object as the file name. For speed, you'll want that file to be buffered so you give the resulting reference to the constructor for a `BufferedReader`. `BufferedReader` provides `lines()`, which produces a `Stream<String>`:

```
// iostreams/BufferedInputFile.java
// {VisuallyInspectOutput}
import java.io.*;
import java.util.stream.*;

public class BufferedInputFile {
  public static String read(String filename) {
    try(BufferedReader in = new BufferedReader(
      new FileReader(filename))) {
      return in.lines()
        .collect(Collectors.joining("\n"));
    } catch(IOException e) {
      throw new RuntimeException(e);
    }
  }
  public static void main(String[] args) {
    System.out.print(
      read("BufferedInputFile.java"));
  }
}
```

`Collectors.joining()` uses a `StringBuilder` internally to accumulate its result. The file is automatically closed via the try-with-resources clause.

Input from Memory

Here, the `String` result from `BufferedInputFile.read()` is used to create a `StringReader`. Then `read()` produces each character which is displayed on the console:

```
// iostreams/MemoryInput.java
// {VisuallyInspectOutput}
import java.io.*;

public class MemoryInput {
  public static void
  main(String[] args) throws IOException {
    StringReader in = new StringReader(
      BufferedInputFile.read("MemoryInput.java"));
    int c;
```

```
    while((c = in.read()) != -1)
      System.out.print((char)c);
  }
}
```

`read()` returns the next character as an `int` and thus the return value must be cast to a `char` to display properly.

Formatted Memory Input

To read "formatted" data, you use a `DataInputStream`, a byte-oriented I/O class (rather than `char`-oriented). Thus you must use all `InputStream` classes rather than `Reader` classes. You can read anything (such as a file) as bytes using `InputStream` classes, but here a `String` is used:

```
// iostreams/FormattedMemoryInput.java
// {VisuallyInspectOutput}
import java.io.*;

public class FormattedMemoryInput {
  public static void main(String[] args) {
    try(
      DataInputStream in = new DataInputStream(
        new ByteArrayInputStream(
          BufferedInputFile.read(
            "FormattedMemoryInput.java")
              .getBytes())))
    ) {
      while(true)
        System.out.write((char)in.readByte());
    } catch(EOFException e) {
      System.out.println("\nEnd of stream");
    } catch(IOException e) {
      throw new RuntimeException(e);
    }
  }
}
```

A `ByteArrayInputStream` must receive an array of bytes, produced here with `String.getBytes()`. The resulting `ByteArrayInputStream` is an appropriate `InputStream` to hand to `DataInputStream`.

If you read the characters from a `DataInputStream` one `byte` at a time using `readByte()`, any `byte` value is a legitimate result, so the return value cannot be used to detect the end of input. Instead, use the `available()` method to find out how many more characters are available. This shows how to read a file one `byte` at a time:

```
// iostreams/TestEOF.java
// Testing for end of file
// {VisuallyInspectOutput}
import java.io.*;

public class TestEOF {
  public static void main(String[] args) {
    try(
      DataInputStream in = new DataInputStream(
        new BufferedInputStream(
          new FileInputStream("TestEOF.java")))
    ) {
      while(in.available() != 0)
        System.out.write(in.readByte());
    } catch(IOException e) {
      throw new RuntimeException(e);
    }
  }
}
```

Note that `available()` works differently depending on what sort of medium you're reading from; it's literally "the number of bytes that can be read *without blocking*." With a file, this means the whole file, but with a different kind of stream this might not be true, so use it thoughtfully.

You can also detect the end of input in cases like these by catching an exception. However, control flow is considered a misuse of exceptions.

Basic File Output

A `FileWriter` object writes data to a file. You'll virtually always buffer the output by wrapping it in a `BufferedWriter` (try removing this wrapping to see the impact on the performance—buffering tends to dramatically increase performance of I/O operations). Here, a `FileWriter` is decorated as a `PrintWriter` to provide formatting. The data file created this way is readable as an ordinary text file:

```
// iostreams/BasicFileOutput.java
// {VisuallyInspectOutput}
import java.io.*;

public class BasicFileOutput {
  static String file = "BasicFileOutput.dat";
  public static void main(String[] args) {
    try(
      BufferedReader in = new BufferedReader(
        new StringReader(
          BufferedInputFile.read(
            "BasicFileOutput.java")));
      PrintWriter out = new PrintWriter(
        new BufferedWriter(new FileWriter(file)))
    ) {
      in.lines().forEach(out::println);
    } catch(IOException e) {
      throw new RuntimeException(e);
    }
    // Show the stored file:
    System.out.println(BufferedInputFile.read(file));
  }
}
```

The try-with-resources flushes the buffers and closes the file.

Text File Output Shortcut

Java 5 added a helper constructor to `PrintWriter` so you don't have to decorate by hand every time you create a text file and write to it. Here's `BasicFileOutput.java` rewritten to use this shortcut:

```
// iostreams/FileOutputShortcut.java
// {VisuallyInspectOutput}
import java.io.*;

public class FileOutputShortcut {
  static String file = "FileOutputShortcut.dat";
  public static void main(String[] args) {
    try(
      BufferedReader in = new BufferedReader(
        new StringReader(BufferedInputFile.read(
          "FileOutputShortcut.java")));
      // Here's the shortcut:
      PrintWriter out = new PrintWriter(file)
    ) {
      in.lines().forEach(out::println);
    } catch(IOException e) {
      throw new RuntimeException(e);
    }
    System.out.println(BufferedInputFile.read(file));
  }
}
```

You still get buffering, you just don't do it yourself. Unfortunately, other commonly written tasks were not given shortcuts, so typical I/O streams still involve a lot of redundant text. The Files chapter shows how this and other tasks were greatly simplified by taking a different approach.

Storing and Recovering Data

A `PrintWriter` formats human-readable data. To output data for recovery by another stream, you use a `DataOutputStream` to write the data and a `DataInputStream` to recover the data. These streams can be anything, but the following example uses a file, buffered for both reading and writing. `DataOutputStream` and `DataInputStream` are byte-oriented and thus require `InputStreams` and `OutputStreams`:

```
// iostreams/StoringAndRecoveringData.java
import java.io.*;

public class StoringAndRecoveringData {
  public static void main(String[] args) {
    try(
      DataOutputStream out = new DataOutputStream(
        new BufferedOutputStream(
          new FileOutputStream("Data.txt")))
    ) {
      out.writeDouble(3.14159);
      out.writeUTF("That was pi");
      out.writeDouble(1.41413);
      out.writeUTF("Square root of 2");
    } catch(IOException e) {
      throw new RuntimeException(e);
    }
    try(
      DataInputStream in = new DataInputStream(
        new BufferedInputStream(
          new FileInputStream("Data.txt")))
    ) {
      System.out.println(in.readDouble());
      // Only readUTF() will recover the
      // Java-UTF String properly:
      System.out.println(in.readUTF());
      System.out.println(in.readDouble());
      System.out.println(in.readUTF());
    } catch(IOException e) {
      throw new RuntimeException(e);
    }
  }
}
/* Output:
3.14159
That was pi
1.41413
Square root of 2
*/
```

If you use a `DataOutputStream` to write the data, then Java guarantees you can accurately recover the data using a `DataInputStream`—regardless of what different

platforms write and read the data. This is valuable, as anyone knows who has spent time worrying about platform-specific data issues. That problem vanishes if you have Java on both platforms.[193]

The only reliable way to write a `String` using a `DataOutputStream` so it can be recovered by a `DataInputStream` is to use UTF-8 encoding, produced in this example with `writeUTF()` and `readUTF()`. UTF-8 is a multi-byte format, and the length of encoding varies according to the actual character set in use. If you're working with ASCII or mostly ASCII characters (which occupy only seven bits), Unicode wastes space and/or bandwidth, so UTF-8 encodes ASCII characters in a single byte, and non-ASCII characters in two or three bytes. In addition, the length of the string is stored in the first two bytes of the UTF-8 string. However, `writeUTF()` and `readUTF()` use a special variation of UTF-8 for Java (described in the JDK documentation for those methods), so if you read a string written with `writeUTF()` using a non-Java program, you must write special code to read the string properly.

With `writeUTF()` and `readUTF()`, you can intermingle `Strings` and other types of data in a `DataOutputStream`, with the knowledge that the `Strings` are properly stored as Unicode and are easily recoverable with a `DataInputStream`.

The `writeDouble()` method stores the `double` number to the stream, and the complementary `readDouble()` method recovers it (there are similar methods for reading and writing the other types). But for any of the reading methods to work correctly, you must know the exact placement of the data item in the stream because it is equally possible to read the stored `double` as a simple sequence of bytes, or as a `char`, etc. So you must either use a fixed format for the data in the file, or include extra information in the file you parse to determine where the data is located. Note that object serialization or XML (both described in the Appendix: Object Serialization) can be easier ways to store and retrieve complex data structures.

Reading and Writing Random-Access Files

Using a `RandomAccessFile` is like using a combined `DataInputStream` and `DataOutputStream` (because it implements the same interfaces: `DataInput` and `DataOutput`). In addition, you can use `seek()` to move about in the file and change the values.

[193] XML is another way to solve the problem of moving data across different computing platforms, and does not depend on having Java on all platforms. XML is introduced in the Appendix: Object Serialization.

With `RandomAccessFile`, you must know the layout of the file to manipulate it properly. `RandomAccessFile` has specific methods to read and write primitives and UTF-8 strings:

```
// iostreams/UsingRandomAccessFile.java
import java.io.*;

public class UsingRandomAccessFile {
  static String file = "rtest.dat";
  public static void display() {
    try(
      RandomAccessFile rf =
        new RandomAccessFile(file, "r")
    ) {
      for(int i = 0; i < 7; i++)
        System.out.println(
          "Value " + i + ": " + rf.readDouble());
      System.out.println(rf.readUTF());
    } catch(IOException e) {
      throw new RuntimeException(e);
    }
  }
  public static void main(String[] args) {
    try(
      RandomAccessFile rf =
        new RandomAccessFile(file, "rw")
    ) {
      for(int i = 0; i < 7; i++)
        rf.writeDouble(i*1.414);
      rf.writeUTF("The end of the file");
      rf.close();
      display();
    } catch(IOException e) {
      throw new RuntimeException(e);
    }
    try(
      RandomAccessFile rf =
        new RandomAccessFile(file, "rw")
    ) {
      rf.seek(5*8);
      rf.writeDouble(47.0001);
      rf.close();
```

```
      display();
    } catch(IOException e) {
      throw new RuntimeException(e);
    }
  }
}
/* Output:
Value 0: 0.0
Value 1: 1.414
Value 2: 2.828
Value 3: 4.242
Value 4: 5.656
Value 5: 7.069999999999999
Value 6: 8.484
The end of the file
Value 0: 0.0
Value 1: 1.414
Value 2: 2.828
Value 3: 4.242
Value 4: 5.656
Value 5: 47.0001
Value 6: 8.484
The end of the file
*/
```

The `display()` method opens a file and displays the elements within as `double` values. In `main()`, the file is created, opened and modified. A `double` is always eight bytes long, so to `seek()` to `double` element `5` you multiply `5*8` to produce the seek value.

As previously noted, `RandomAccessFile` is effectively separate from the rest of the I/O hierarchy, save for the fact it implements `DataInput` and `DataOutput`. It doesn't support decoration, so you cannot combine it with any of the aspects of the `InputStream` and `OutputStream` subclasses. You must assume that a `RandomAccessFile` is properly buffered because you cannot add buffering.

The one option you have is in the second constructor argument: You can open a `RandomAccessFile` to read (`"r"`) or read and write (`"rw"`).

Consider using `nio` memory-mapped files instead of `RandomAccessFile`. These are described in the Appendix: New I/O.

Summary

The Java I/O stream library does satisfy basic requirements: You can read and write with the console, a file, a block of memory, or even across the Internet. With inheritance, you can create new types of input and output objects. You can even add simple extensibility to the kinds of objects a stream will accept by redefining the `toString()` method that's automatically called when you pass an object to a method that's expecting a `String` (Java's limited "automatic type conversion").

There are questions left unanswered by the documentation and design of the I/O stream library. For example, it would be nice to say you want an exception thrown if you try to overwrite a file when opening it for output—some programming systems allow you to open an output file, but only if it doesn't already exist. In Java, it appears you are supposed to use a `File` object to determine whether a file exists, because if you open it as a `FileOutputStream` or `FileWriter`, it will always get overwritten.

The I/O stream library brings up mixed feelings; it does much of the job and it's portable. But if you don't already understand the *Decorator* design pattern, the design is not intuitive, so there's extra overhead in learning and teaching it. It's also incomplete; for example, in the past I had to write utilities to read text files with a reasonable amount of code—fortunately Java 7 nio eliminates the need for such things.

Once you *do* understand the *Decorator* pattern and begin using the library in situations that require its flexibility, you can begin to benefit from this design, at which point its cost in extra lines of code might not bother you as much. Always check, however, to make sure you can't instead solve your problem using the libraries and techniques shown in the Files chapter.

Appendix: Standard I/O

The term *standard I/O* refers to the Unix concept of a single stream of information that is used by a program (this idea is reproduced in some form in most operating systems). All the program's input can come from *standard input*, all of its output can go to *standard output*, and all of its error messages can be sent to *standard error*. The value of standard I/O is that programs can easily be chained together, and one program's standard output can become the standard input for another program. This is a powerful tool.

Reading from Standard Input

Following the standard I/O model, Java has `System.in`, `System.out`, and `System.err`. Throughout this book, you've seen how to write to standard output using `System.out`, which is already pre-wrapped as a `PrintStream` object. `System.err` is likewise a `PrintStream`, but `System.in` is a raw `InputStream` with no wrapping. This means that although you can use `System.out` and `System.err` right away, `System.in` must be wrapped before you can read from it.

You'll typically read input a line at a time. To do this, wrap `System.in` in a `BufferedReader`, which requires you to convert `System.in` to a `Reader` using `InputStreamReader`. Here's an example that echoes each line you type in:

```
// standardio/Echo.java
// How to read from standard input
import java.io.*;
import onjava.TimedAbort;

public class Echo {
  public static void main(String[] args) {
    TimedAbort abort = new TimedAbort(2);
    new BufferedReader(
      new InputStreamReader(System.in))
      .lines()
      .peek(ln -> abort.restart())
      .forEach(System.out::println);
    // Ctrl-Z or two seconds inactivity
    // terminates the program
  }
}
```

`BufferedReader` has a `lines()` method that returns a `Stream<String>`, and this shows the flexibility of the stream model: it works just fine with standard input. The `peek()` method restarts the `TimedAbort` to keep the program open as long as there's input at least every two seconds.

Changing `System.out` to a `PrintWriter`

`System.out` is a `PrintStream`, which is an `OutputStream`. `PrintWriter` has a constructor that takes an `OutputStream` as an argument. Thus, if you want, you can convert `System.out` into a `PrintWriter` using that constructor:

```
// standardio/ChangeSystemOut.java
// Turn System.out into a PrintWriter
import java.io.*;

public class ChangeSystemOut {
  public static void main(String[] args) {
    PrintWriter out =
      new PrintWriter(System.out, true);
    out.println("Hello, world");
  }
}
/* Output:
Hello, world
*/
```

It's important to use the two-argument version of the `PrintWriter` constructor and to set the second argument to `true` to enable automatic flushing. Otherwise, you might not see the output.

Redirecting Standard I/O

The Java `System` class can redirect the standard input, output, and error I/O streams using simple `static` method calls:

- `setIn(InputStream)`
- `setOut(PrintStream)`
- `setErr(PrintStream)`

Redirecting output is especially useful if you suddenly start creating a large amount of output on your screen, and it's scrolling past faster than you can read it. Redirecting input is valuable for a command-line program to test a particular user-input sequence repeatedly. Here's a simple example that shows these methods:

```
// standardio/Redirecting.java
// Demonstrates standard I/O redirection
import java.io.*;

public class Redirecting {
  public static void main(String[] args) {
    PrintStream console = System.out;
    try(
      BufferedInputStream in = new BufferedInputStream(
        new FileInputStream("Redirecting.java"));
      PrintStream out = new PrintStream(
        new BufferedOutputStream(
          new FileOutputStream("Redirecting.txt")))
    ) {
      System.setIn(in);
      System.setOut(out);
      System.setErr(out);
      new BufferedReader(
        new InputStreamReader(System.in))
        .lines()
        .forEach(System.out::println);
    } catch(IOException e) {
      throw new RuntimeException(e);
    } finally {
      System.setOut(console);
    }
  }
}
```

This program attaches standard input to a file and redirects standard output and standard error to another file. It stores a reference to the original `System.out` object at the beginning of the program, and restores the system output to that object at the end.

I/O redirection manipulates streams of bytes, not streams of characters; thus, `InputStreams` and `OutputStreams` are used rather than `Readers` and `Writers`.

Process Control

The Java library provides classes to execute operating system programs from inside Java, and control the input and output from such programs.

A common task is to run a program and send the resulting output to the console. This section contains a utility to simplify this task.

Two types of errors can occur with this utility: the normal errors that result in exceptions—for these we just rethrow a `RuntimeException`—and errors from the execution of the process itself. We report these errors with a separate exception:

```
// onjava/OSExecuteException.java
package onjava;

public class
OSExecuteException extends RuntimeException {
  public OSExecuteException(String why) {
    super(why);
  }
}
```

To run a program, you pass `OSExecute.command()` a `String command`, the same command you would type to run the program on the console. This command is passed to the `java.lang.ProcessBuilder` constructor (which requires it as a sequence of `String` objects), and the resulting `ProcessBuilder` object is started:

```
// onjava/OSExecute.java
// Run an operating system command
// and send the output to the console
package onjava;
import java.io.*;

public class OSExecute {
  public static void command(String command) {
    boolean err = false;
    try {
      Process process = new ProcessBuilder(
        command.split(" ")).start();
      try(
```

```
        BufferedReader results = new BufferedReader(
          new InputStreamReader(
            process.getInputStream()));
        BufferedReader errors = new BufferedReader(
          new InputStreamReader(
            process.getErrorStream()))
      ) {
        results.lines()
          .forEach(System.out::println);
        err = errors.lines()
          .peek(System.err::println)
          .count() > 0;
      }
    } catch(IOException e) {
      throw new RuntimeException(e);
    }
    if(err)
      throw new OSExecuteException(
        "Errors executing " + command);
  }
}
```

To capture the standard output stream from the program as it executes, you call `getInputStream()`. This is because an `InputStream` is something we can read from.

Here the lines are only displayed, but you might also capture and return them from `command()`.

The program's errors are sent to the standard error stream, and are captured by calling `getErrorStream()`. If there are any errors, they are displayed and an `OSExecuteException` is thrown so the calling program will handle the problem.

Here's an example that shows how to use `OSExecute`:

```
// standardio/OSExecuteDemo.java
// Demonstrates standard I/O redirection
// javap -cp build/classes/java/main OSExecuteDemo
// {ExcludeFromGradle}
import onjava.*;

public class OSExecuteDemo {}
/* Output:
Compiled from "OSExecuteDemo.java"
public class OSExecuteDemo {
  public OSExecuteDemo();
}
*/
```

This uses the `javap` decompiler (that comes with the JDK) to decompile the program.

Appendix: New I/O

> The Java "new" I/O library, introduced in Java 1.4 in the `java.nio.*` packages, has one goal: speed.

In fact, the "old" I/O packages were reimplemented using `nio` to take advantage of this speed increase, so you benefit even if you don't explicitly write code with `nio`. The speed increase occurs both in file I/O, explored here, and in network I/O, used for example with Internet programming.

The speed comes from using structures that are closer to the operating system's way of performing I/O: *channels* and *buffers*. Think of it as a coal mine; the channel is the mine containing the seam of coal (the data), and the buffer is the cart you send into the mine. The cart comes back full of coal, and you get the coal from the cart. That is, you don't interact directly with the channel; you interact with the buffer and send the buffer into the channel. The channel either pulls data from the buffer, or puts data into the buffer.

This appendix goes into some depth exploring the `nio` package. Higher-level libraries like I/O streams use `nio`, but most of the time you won't need to work with I/O at this level. With Java 7 & 8, you (ideally) not even have to bother with I/O streams except in special cases. Ideally, everything you'll regularly use is covered in the Files chapter. Understanding `nio` is only necessary when you're struggling with performance (when you might need, for example, memory-mapped files) or creating your own I/O library.

`ByteBuffer`s

The only type of buffer that communicates directly with a channel is a `ByteBuffer`—that is, a buffer that holds raw bytes. If you look at the JDK documentation for `java.nio.ByteBuffer`, you'll see it's fairly basic: You create one by telling it how much storage to allocate, and there are methods to put and get data, in either raw

byte form or as primitive data types. But there's no way to put or get an object, or even a `String`. It's fairly low-level, precisely because this makes a more efficient mapping with most operating systems.

Three of the classes in the "old" I/O were modified to produce a `FileChannel`: `FileInputStream`, `FileOutputStream`, and, for both reading and writing, `RandomAccessFile`. Notice that these are the byte manipulation streams, in keeping with the low-level nature of `nio`. The `Reader` and `Writer` character-mode classes do not produce channels, but the `java.nio.channels.Channels` class has utility methods to produce `Readers` and `Writers` from channels.

Here we exercise all three types of stream to produce channels that are writeable, read/writeable, and readable:

```
// newio/GetChannel.java
// Getting channels from streams
import java.nio.*;
import java.nio.channels.*;
import java.io.*;

public class GetChannel {
  private static String name = "data.txt";
  private static final int BSIZE = 1024;
  public static void main(String[] args) {
    // Write a file:
    try(
      FileChannel fc = new FileOutputStream(name)
        .getChannel()
    ) {
      fc.write(ByteBuffer
        .wrap("Some text ".getBytes()));
    } catch(IOException e) {
      throw new RuntimeException(e);
    }
    // Add to the end of the file:
    try(
      FileChannel fc = new RandomAccessFile(
        name, "rw").getChannel()
    ) {
      fc.position(fc.size()); // Move to the end
      fc.write(ByteBuffer
        .wrap("Some more".getBytes()));
```

```
    } catch(IOException e) {
      throw new RuntimeException(e);
    }
    // Read the file:
    try(
      FileChannel fc = new FileInputStream(name)
        .getChannel()
    ) {
      ByteBuffer buff = ByteBuffer.allocate(BSIZE);
      fc.read(buff);
      buff.flip();
      while(buff.hasRemaining())
        System.out.write(buff.get());
    } catch(IOException e) {
      throw new RuntimeException(e);
    }
    System.out.flush();
  }
}
/* Output:
Some text Some more
*/
```

For any of the stream classes shown here, `getChannel()` will produce a `FileChannel`. A channel is fairly basic: Hand it a `ByteBuffer` for reading or writing, and lock regions of the file for exclusive access (this is described later).

One way to put bytes into a `ByteBuffer` is to stuff them in directly using one of the `put()` methods, to put one or more bytes or values of primitive types. However, as seen here, you can also "wrap" an existing `byte` array in a `ByteBuffer` using the `wrap()` method. When you do this, the underlying array is not copied, but instead is used as the storage for the generated `ByteBuffer`. We say that the `ByteBuffer` is "backed by" the array.

The `data.txt` file is reopened using a `RandomAccessFile`. Notice you can move the `FileChannel` around in the file; here, it is moved to the end so additional writes are appended.

For read-only access, you must explicitly allocate a `ByteBuffer` using the `static allocate()` method. The goal of `nio` is to rapidly move large amounts of data, so the size of the `ByteBuffer` should be significant—in fact, the 1K used here is probably

quite a bit smaller than you'd normally use (you'll have to experiment with your working application to find the best size).

It's also possible to go for even more speed by using `allocateDirect()` instead of `allocate()` to produce a "direct" buffer that can have an even higher coupling with the operating system. However, the overhead in such an allocation is greater, and the actual implementation varies from one operating system to another, so again, you must experiment with your working application to discover whether direct buffers will buy you any advantage in speed.

Once you call `read()` to tell the `FileChannel` to store bytes into the `ByteBuffer`, you must call `flip()` on the buffer to tell it to get ready to have its bytes extracted (yes, this seems a bit crude, but remember it's very low-level and is done for maximum speed). And if we were to use the buffer for further `read()` operations, we'd also call `clear()` to prepare it for each `read()`. This simple file-copying program demonstrates:

```
// newio/ChannelCopy.java
// Copying a file using channels and buffers
// {java ChannelCopy ChannelCopy.java test.txt}
import java.nio.*;
import java.nio.channels.*;
import java.io.*;

public class ChannelCopy {
  private static final int BSIZE = 1024;
  public static void main(String[] args) {
    if(args.length != 2) {
      System.out.println(
        "arguments: sourcefile destfile");
      System.exit(1);
    }
    try(
      FileChannel in = new FileInputStream(
        args[0]).getChannel();
      FileChannel out = new FileOutputStream(
        args[1]).getChannel()
    ) {
      ByteBuffer buffer = ByteBuffer.allocate(BSIZE);
      while(in.read(buffer) != -1) {
        buffer.flip(); // Prepare for writing
```

```
        out.write(buffer);
        buffer.clear();  // Prepare for reading
      }
    } catch(IOException e) {
      throw new RuntimeException(e);
    }
  }
}
```

One `FileChannel` is opened for reading, and one for writing. A `ByteBuffer` is allocated, and when `FileChannel.read()` returns `-1` (a holdover, no doubt, from Unix and C), it means you've reached the end of the input. After each `read()`, which puts data into the buffer, `flip()` prepares the buffer so its information can be extracted by the `write()`. After the `write()`, the information is still in the buffer, and `clear()` resets all the internal pointers so it's ready to accept data during another `read()`.

The preceding program is not the ideal way to handle this kind of operation, however. Special methods `transferTo()` and `transferFrom()` allow you to connect one channel directly to another:

```
// newio/TransferTo.java
// Using transferTo() between channels
// {java TransferTo TransferTo.java TransferTo.txt}
import java.nio.channels.*;
import java.io.*;

public class TransferTo {
  public static void main(String[] args) {
    if(args.length != 2) {
      System.out.println(
        "arguments: sourcefile destfile");
      System.exit(1);
    }
    try(
      FileChannel in = new FileInputStream(
        args[0]).getChannel();
      FileChannel out = new FileOutputStream(
        args[1]).getChannel()
    ) {
      in.transferTo(0, in.size(), out);
```

```
      // Or:
      // out.transferFrom(in, 0, in.size());
    } catch(IOException e) {
      throw new RuntimeException(e);
    }
  }
}
```

You won't do this very often, but it's good to know about.

Converting Data

To print the information in the file in `GetChannel.java`, we pull the data out one `byte` at a time and cast each `byte` to a `char`. This seems primitive—if you look at the `java.nio.CharBuffer` class, you'll see it has a `toString()` method that says, "Returns a `String` containing the characters in this buffer." A `ByteBuffer` can be viewed as a `CharBuffer` with the `asCharBuffer()` method, so why not use that? As you see from the first line in the output statement below, this doesn't work out:

```
// newio/BufferToText.java
// Converting text to and from ByteBuffers
import java.nio.*;
import java.nio.channels.*;
import java.nio.charset.*;
import java.io.*;

public class BufferToText {
  private static final int BSIZE = 1024;
  public static void main(String[] args) {
    try(
      FileChannel fc = new FileOutputStream(
        "data2.txt").getChannel()
    ) {
      fc.write(ByteBuffer.wrap("Some text".getBytes()));
    } catch(IOException e) {
      throw new RuntimeException(e);
    }
    ByteBuffer buff = ByteBuffer.allocate(BSIZE);
    try(
```

```
    FileChannel fc = new FileInputStream(
      "data2.txt").getChannel()
  ) {
    fc.read(buff);
  } catch(IOException e) {
    throw new RuntimeException(e);
  }
  buff.flip();
  // Doesn't work:
  System.out.println(buff.asCharBuffer());
  // Decode using this system's default Charset:
  buff.rewind();
  String encoding =
    System.getProperty("file.encoding");
  System.out.println("Decoded using " +
    encoding + ": "
    + Charset.forName(encoding).decode(buff));
  // Encode with something that prints:
  try(
    FileChannel fc = new FileOutputStream(
      "data2.txt").getChannel()
  ) {
    fc.write(ByteBuffer.wrap(
      "Some text".getBytes("UTF-16BE")));
  } catch(IOException e) {
    throw new RuntimeException(e);
  }
  // Now try reading again:
  buff.clear();
  try(
    FileChannel fc = new FileInputStream(
      "data2.txt").getChannel()
  ) {
    fc.read(buff);
  } catch(IOException e) {
    throw new RuntimeException(e);
  }
  buff.flip();
  System.out.println(buff.asCharBuffer());
  // Use a CharBuffer to write through:
  buff = ByteBuffer.allocate(24);
  buff.asCharBuffer().put("Some text");
```

```
    try(
      FileChannel fc = new FileOutputStream(
        "data2.txt").getChannel()
    ) {
      fc.write(buff);
    } catch(IOException e) {
      throw new RuntimeException(e);
    }
    // Read and display:
    buff.clear();
    try(
      FileChannel fc = new FileInputStream(
        "data2.txt").getChannel()
    ) {
      fc.read(buff);
    } catch(IOException e) {
      throw new RuntimeException(e);
    }
    buff.flip();
    System.out.println(buff.asCharBuffer());
  }
}
/* Output:
????
Decoded using windows-1252: Some text
Some text
Some textNULNULNUL
*/
```

The buffer contains plain bytes, and to turn these into characters, we must either *encode* them as we put them in (so they are meaningful when they come out) or *decode* them as they come out of the buffer. This can be accomplished using the `java.nio.charset.Charset` class, which provides tools for encoding into many different types of character sets:

```
// newio/AvailableCharSets.java
// Displays Charsets and aliases
import java.nio.charset.*;
import java.util.*;

public class AvailableCharSets {
  public static void main(String[] args) {
    SortedMap<String,Charset> charSets =
      Charset.availableCharsets();
    for(String csName : charSets.keySet()) {
      System.out.print(csName);
      Iterator aliases = charSets.get(csName)
        .aliases().iterator();
      if(aliases.hasNext())
        System.out.print(": ");
      while(aliases.hasNext()) {
        System.out.print(aliases.next());
        if(aliases.hasNext())
          System.out.print(", ");
      }
      System.out.println();
    }
  }
}
/* Output: (First 7 Lines)
Big5: csBig5
Big5-HKSCS: big5-hkscs, big5hk, Big5_HKSCS, big5hkscs
CESU-8: CESU8, csCESU-8
EUC-JP: csEUCPkdFmtjapanese, x-euc-jp, eucjis,
Extended_UNIX_Code_Packed_Format_for_Japanese, euc_jp,
eucjp, x-eucjp
EUC-KR: ksc5601-1987, csEUCKR, ksc5601_1987, ksc5601,
5601, euc_kr, ksc_5601, ks_c_5601-1987, euckr
GB18030: gb18030-2000
GB2312: gb2312, euc-cn, x-EUC-CN, euccn, EUC_CN,
gb2312-80, gb2312-1980
                  ...
*/
```

So, returning to `BufferToText.java`, if you `rewind()` the buffer (to go back to the beginning of the data), then use that platform's default character set to `decode()` the data, the resulting `CharBuffer` will display on the console just fine. To discover the

default character set, use `System.getProperty("file.encoding")`, which produces the `String` that names the character set. Passing this to `Charset.forName()` produces the `Charset` object that decodes the `String`.

Another alternative is to `encode()` using a character set that produces something printable when the file is read, as you see in the third part of `BufferToText.java`. Here, UTF-16BE is used to write the text into the file, and when it is read, all you must do is convert it to a `CharBuffer`, and it produces the expected text.

Finally, you see what happens if you *write* to the `ByteBuffer` through a `CharBuffer` (you'll learn more about this later). Note that 24 bytes are allocated for the `ByteBuffer`. Each `char` requires two bytes so this is enough for 12 `char`s, but "Some text" only has 9. The remaining zero bytes still appear in the representation of the `CharBuffer` produced by its `toString()`, as shown in the output.

Fetching Primitives

Although a `ByteBuffer` only holds bytes, it contains methods to produce each different type of primitive value from the bytes it contains. This example shows the insertion and extraction of various values using these methods:

```
// newio/GetData.java
// Getting different representations from a ByteBuffer
import java.nio.*;

public class GetData {
  private static final int BSIZE = 1024;
  public static void main(String[] args) {
    ByteBuffer bb = ByteBuffer.allocate(BSIZE);
    // Allocation automatically zeroes the ByteBuffer:
    int i = 0;
    while(i++ < bb.limit())
      if(bb.get() != 0)
        System.out.println("nonzero");
    System.out.println("i = " + i);
    bb.rewind();
    // Store and read a char array:
    bb.asCharBuffer().put("Howdy!");
    char c;
```

```
    while((c = bb.getChar()) != 0)
      System.out.print(c + " ");
    System.out.println();
    bb.rewind();
    // Store and read a short:
    bb.asShortBuffer().put((short)471142);
    System.out.println(bb.getShort());
    bb.rewind();
    // Store and read an int:
    bb.asIntBuffer().put(99471142);
    System.out.println(bb.getInt());
    bb.rewind();
    // Store and read a long:
    bb.asLongBuffer().put(99471142);
    System.out.println(bb.getLong());
    bb.rewind();
    // Store and read a float:
    bb.asFloatBuffer().put(99471142);
    System.out.println(bb.getFloat());
    bb.rewind();
    // Store and read a double:
    bb.asDoubleBuffer().put(99471142);
    System.out.println(bb.getDouble());
    bb.rewind();
  }
}
/* Output:
i = 1025
H o w d y !
12390
99471142
99471142
9.9471144E7
9.9471142E7
*/
```

After a `ByteBuffer` is allocated, its values are checked to see whether buffer allocation automatically zeroes the contents—and it does. All 1,024 values are checked (up to the `limit()` of the buffer), and all are zero.

The easiest way to insert primitive values into a `ByteBuffer` is to get the appropriate "view" on that buffer using `asCharBuffer()`, `asShortBuffer()`, etc., then to use

that view's `put()` method. This is performed for each of the primitive data types. The only one of these that is slightly odd is the `put()` for the `ShortBuffer`, which requires a cast (the cast truncates and changes the resulting value). All the other view buffers do not require casting in their `put()` methods.

View Buffers

A "view buffer" looks at an underlying `ByteBuffer` through the window of a particular primitive type. The `ByteBuffer` is still the actual storage that's "backing" the view, so any changes you make to the view are reflected in modifications to the data in the `ByteBuffer`. As seen in the previous example, this conveniently inserts primitive types into a `ByteBuffer`. A view can also read primitive values from a `ByteBuffer`, either one at a time (as `ByteBuffer` allows) or in batches (into arrays). Here's an example that manipulates `int`s in a `ByteBuffer` via an `IntBuffer`:

```
// newio/IntBufferDemo.java
// Manipulating ints in a ByteBuffer with an IntBuffer
import java.nio.*;

public class IntBufferDemo {
  private static final int BSIZE = 1024;
  public static void main(String[] args) {
    ByteBuffer bb = ByteBuffer.allocate(BSIZE);
    IntBuffer ib = bb.asIntBuffer();
    // Store an array of int:
    ib.put(new int[]{ 11, 42, 47, 99, 143, 811, 1016 });
    // Absolute location read and write:
    System.out.println(ib.get(3));
    ib.put(3, 1811);
    // Setting a new limit before rewinding the buffer.
    ib.flip();
    while(ib.hasRemaining()) {
      int i = ib.get();
      System.out.println(i);
    }
  }
}
/* Output:
99
```

```
11
42
47
1811
143
811
1016
*/
```

The overloaded `put()` method is first used to store an array of `int`. The following `get()` and `put()` method calls directly access an `int` location in the underlying `ByteBuffer`. Note that these absolute location accesses are available for primitive types by talking directly to a `ByteBuffer`, as well.

Once the underlying `ByteBuffer` is filled with `int`s or some other primitive type via a view buffer, then that `ByteBuffer` can be written directly to a channel. You can just as easily read from a channel and use a view buffer to convert everything to a particular type of primitive. Here's an example that interprets the same sequence of bytes as `short`, `int`, `float`, `long`, and `double` by producing different view buffers on the same `ByteBuffer`:

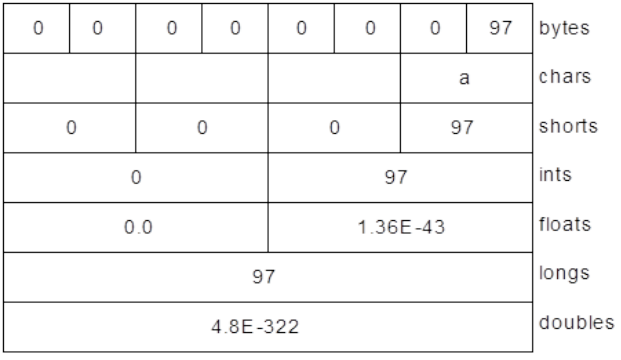
```
// newio/ViewBuffers.java
import java.nio.*;

public class ViewBuffers {
  public static void main(String[] args) {
    ByteBuffer bb = ByteBuffer.wrap(
      new byte[]{ 0, 0, 0, 0, 0, 0, 0, 'a' });
    bb.rewind();
    System.out.print("Byte Buffer ");
    while(bb.hasRemaining())
      System.out.print(
        bb.position()+ " -> " + bb.get() + ", ");
    System.out.println();
    CharBuffer cb =
      ((ByteBuffer)bb.rewind()).asCharBuffer();
    System.out.print("Char Buffer ");
    while(cb.hasRemaining())
      System.out.print(
        cb.position() + " -> " + cb.get() + ", ");
    System.out.println();
```

```
    FloatBuffer fb =
      ((ByteBuffer)bb.rewind()).asFloatBuffer();
    System.out.print("Float Buffer ");
    while(fb.hasRemaining())
      System.out.print(
        fb.position()+ " -> " + fb.get() + ", ");
    System.out.println();
    IntBuffer ib =
      ((ByteBuffer)bb.rewind()).asIntBuffer();
    System.out.print("Int Buffer ");
    while(ib.hasRemaining())
      System.out.print(
        ib.position()+ " -> " + ib.get() + ", ");
    System.out.println();
    LongBuffer lb =
      ((ByteBuffer)bb.rewind()).asLongBuffer();
    System.out.print("Long Buffer ");
    while(lb.hasRemaining())
      System.out.print(
        lb.position()+ " -> " + lb.get() + ", ");
    System.out.println();
    ShortBuffer sb =
      ((ByteBuffer)bb.rewind()).asShortBuffer();
    System.out.print("Short Buffer ");
    while(sb.hasRemaining())
      System.out.print(
        sb.position()+ " -> " + sb.get() + ", ");
    System.out.println();
    DoubleBuffer db =
      ((ByteBuffer)bb.rewind()).asDoubleBuffer();
    System.out.print("Double Buffer ");
    while(db.hasRemaining())
      System.out.print(
        db.position()+ " -> " + db.get() + ", ");
  }
}
/* Output:
Byte Buffer 0 -> 0, 1 -> 0, 2 -> 0, 3 -> 0, 4 -> 0, 5
-> 0, 6 -> 0, 7 -> 97,
Char Buffer 0 -> NUL, 1 -> NUL, 2 -> NUL, 3 -> a,
Float Buffer 0 -> 0.0, 1 -> 1.36E-43,
Int Buffer 0 -> 0, 1 -> 97,
```

```
Long Buffer 0 -> 97,
Short Buffer 0 -> 0, 1 -> 0, 2 -> 0, 3 -> 97,
Double Buffer 0 -> 4.8E-322,
*/
```

The `ByteBuffer` is produced by "wrapping" an eight-`byte` array, which is then displayed via view buffers of all the different primitive types. The following diagram shows how the data appears differently when read from the different types of buffers:

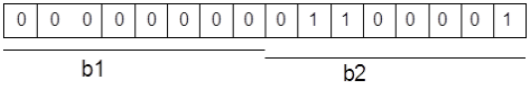


This corresponds to the output from the program.

Endians

Different machines can use different byte-ordering approaches to store data. "Big endian" places the most significant byte in the lowest memory address, and "little endian" places the most significant byte in the highest memory address. When storing a quantity that is greater than one `byte`, like `int`, `float`, etc., you might need to consider byte ordering. A `ByteBuffer` stores data in big endian form, and data sent over a network always uses big endian order. You can change the endianness of a `ByteBuffer` using `order()` with an argument of `ByteOrder.BIG_ENDIAN` or `ByteOrder.LITTLE_ENDIAN`.

Consider a `ByteBuffer` containing the following two bytes:



Reading the data as a `short` (`ByteBuffer.asShortBuffer()`) produces the number 97 (00000000 01100001). Changing to little endian produces the number 24832 (01100001 00000000).

This shows byte ordering changing depending on the endian setting:

```
// newio/Endians.java
// Endian differences and data storage
import java.nio.*;
import java.util.*;

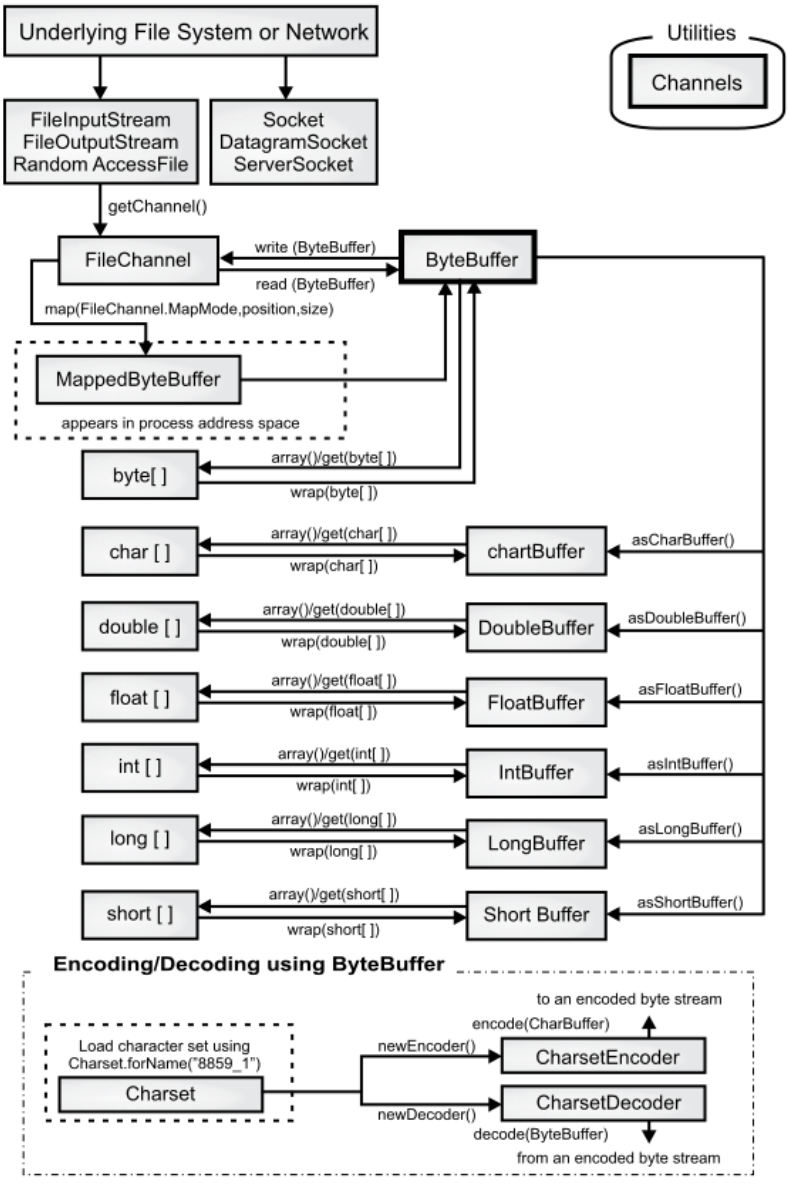
public class Endians {
  public static void main(String[] args) {
    ByteBuffer bb = ByteBuffer.wrap(new byte[12]);
    bb.asCharBuffer().put("abcdef");
    System.out.println(Arrays.toString(bb.array()));
    bb.rewind();
    bb.order(ByteOrder.BIG_ENDIAN);
    bb.asCharBuffer().put("abcdef");
    System.out.println(Arrays.toString(bb.array()));
    bb.rewind();
    bb.order(ByteOrder.LITTLE_ENDIAN);
    bb.asCharBuffer().put("abcdef");
    System.out.println(Arrays.toString(bb.array()));
  }
}
/* Output:
[0, 97, 0, 98, 0, 99, 0, 100, 0, 101, 0, 102]
[0, 97, 0, 98, 0, 99, 0, 100, 0, 101, 0, 102]
[97, 0, 98, 0, 99, 0, 100, 0, 101, 0, 102, 0]
*/
```

The `ByteBuffer` allocates space to hold all the bytes in `charArray` as an external buffer so the `array()` method can be called to display the underlying bytes. The `array()` method is "optional," and you can only call it on a buffer backed by an array, otherwise you'll get an `UnsupportedOperationException`.

`charArray` is inserted into the `ByteBuffer` via a `CharBuffer` view. When the underlying bytes are displayed, the default ordering is the same as the subsequent big endian order, whereas the little endian order swaps the bytes.

Data Manipulation with Buffers

The following diagram illustrates the relationships between the `nio` classes, showing how to move and convert data. For example, to write a `byte` array to a file, wrap the `byte` array using the `ByteBuffer.wrap()` method, open a channel on the `FileOutputStream` using `getChannel()`, then write data into `FileChannel` from the `ByteBuffer`.



`ByteBuffer` is the only way to move data into and out of channels, and you can only create a standalone primitive-typed buffer, or get one from a `ByteBuffer` using an "as" method. That is, you cannot convert a primitive-typed buffer *to* a `ByteBuffer`. Because you are able to move primitive data into and out of a `ByteBuffer` via a view

buffer, this is not really a restriction.

Buffer Details

A `Buffer` consists of data and four indexes to access and manipulate this data efficiently: *mark*, *position*, *limit* and *capacity*. There are methods to set and reset these indexes and to query their value.

Method	Effect
`capacity()`	Returns the buffer's *capacity*.
`clear()`	Clears the buffer, sets the *position* to zero, and *limit* to *capacity*. You call this method to overwrite an existing buffer.
`flip()`	Sets *limit* to *position* and *position* to zero. This method is used to prepare the buffer for a read after data has been written into it.
`limit()`	Returns the value of *limit*.
`limit(int lim)`	Sets the value of *limit*.
`mark()`	Sets *mark* at *position*.
`position()`	Returns the value of *position*.
`position(int pos)`	Sets the value of *position*.
`remaining()`	Returns *limit* - *position*.
`hasRemaining()`	Returns `true` if there are any elements between *position* and *limit*.

Methods that insert and extract data from the buffer update these indexes to reflect the changes.

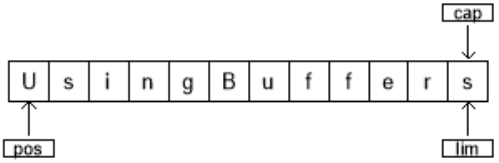
This example uses a very simple algorithm (swapping adjacent characters) to scramble and unscramble characters in a `CharBuffer`:

```
// newio/UsingBuffers.java
import java.nio.*;

public class UsingBuffers {
  private static
  void symmetricScramble(CharBuffer buffer) {
    while(buffer.hasRemaining()) {
      buffer.mark();
      char c1 = buffer.get();
      char c2 = buffer.get();
      buffer.reset();
      buffer.put(c2).put(c1);
    }
  }
  public static void main(String[] args) {
    char[] data = "UsingBuffers".toCharArray();
    ByteBuffer bb =
      ByteBuffer.allocate(data.length * 2);
    CharBuffer cb = bb.asCharBuffer();
    cb.put(data);
    System.out.println(cb.rewind());
    symmetricScramble(cb);
    System.out.println(cb.rewind());
    symmetricScramble(cb);
    System.out.println(cb.rewind());
  }
}
/* Output:
UsingBuffers
sUniBgfuefsr
UsingBuffers
*/
```

Although you can produce a `CharBuffer` directly by calling `wrap()` with a `char` array, an underlying `ByteBuffer` is allocated instead, and a `CharBuffer` is produced as a view on the `ByteBuffer`. This emphasizes that the goal is always to manipulate a `ByteBuffer` because that interacts with a channel.

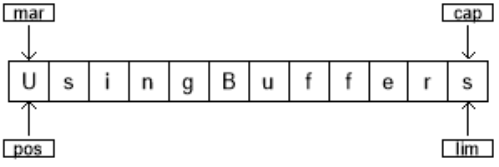
Here's what the buffer looks like at the entrance of the `symmetricScramble()` method:



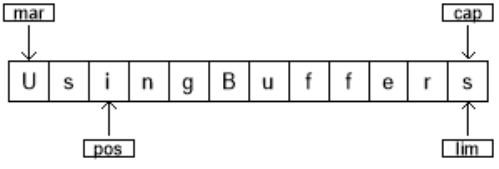
The *position* points to the first element in the buffer, and the *capacity* and *limit* point immediately after the last element.

In `symmetricScramble()`, the `while` loop iterates until *position* is equivalent to *limit*. The *position* of the buffer changes when a relative `get()` or `put()` function is called. You can also call absolute `get()` and `put()` methods that include an index argument: the location where the `get()` or `put()` takes place. These methods do not modify the value of the buffer's *position*.

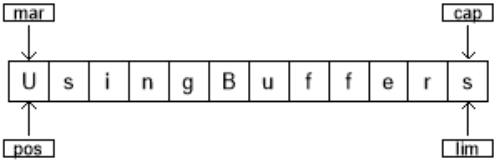
When the control enters the `while` loop, the value of *mark* is set using a `mark()` call. The state of the buffer is then:



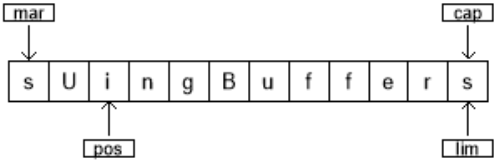
The two relative `get()` calls save the value of the first two characters in variables `c1` and `c2`. After these two calls, the buffer looks like this:



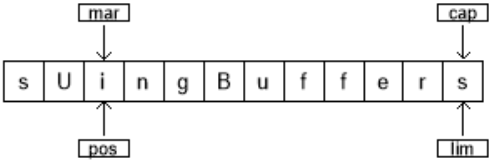
To perform the swap, we write `c2` at *position* `0` and `c1` at *position* `1`. We can either use the absolute `put()` method to achieve this, or set the value of *position* to *mark*, which `reset()` does:



The two `put()` methods write `c2` and then `c1`:



During the next iteration of the loop, *mark* is set to the current value of *position*:



The process continues until the entire buffer is traversed. At the end of the `while` loop, *position* is at the end of the buffer. If you display the buffer, only the characters between the *position* and *limit* are displayed. Thus, to show the entire contents of the buffer, you must set *position* to the start of the buffer using `rewind()`.

When the function `symmetricScramble()` is called again, the `CharBuffer` undergoes the same process and is restored to its original state.

Memory-Mapped Files

Memory-mapped files allow you to create and modify files that are too big to bring into memory. With a memory-mapped file, you can pretend the entire file is in memory and you can access it by treating it as a very large array. This approach greatly simplifies the code you write to modify the file:

```
// newio/LargeMappedFiles.java
// Creating a very large file using mapping
import java.nio.*;
import java.nio.channels.*;
import java.io.*;

public class LargeMappedFiles {
  static int length = 0x8000000; // 128 MB
  public static void
  main(String[] args) throws Exception {
    try(
      RandomAccessFile tdat =
        new RandomAccessFile("test.dat", "rw")
    ) {
      MappedByteBuffer out = tdat.getChannel().map(
        FileChannel.MapMode.READ_WRITE, 0, length);
      for(int i = 0; i < length; i++)
        out.put((byte)'x');
      System.out.println("Finished writing");
      for(int i = length/2; i < length/2 + 6; i++)
        System.out.print((char)out.get(i));
    }
  }
}
/* Output:
Finished writing
xxxxxx
*/
```

To write and read, we start with a `RandomAccessFile`, get a channel for that file, then call `map()` to produce a `MappedByteBuffer`, a particular kind of direct buffer. You must specify the starting point and the length of the region to map in the file—this means you have the option to map smaller regions of a large file.

`MappedByteBuffer` inherits `ByteBuffer`, so it has all of `ByteBuffer`'s methods. Only the simplest uses of `put()` and `get()` are shown here, but you can also use methods like `asCharBuffer()`, etc.

The file created with the preceding program is 128 MB long, probably larger than your OS will allow in memory at one time. The file appears to be accessible all at once because only portions of it are brought into memory, and other parts are

swapped out. This way a very large file (up to 2 GB) can easily be modified. Note that the file-mapping facilities of the underlying operating system are used to maximize performance.

Performance

Although the performance of "old" stream I/O is improved by implementing it with `nio`, mapped file access tends to be dramatically faster. This program does a simple performance comparison:

```
// newio/MappedIO.java
// {ExcludeFromGradle} Runs too long under WSL2
import java.util.*;
import java.nio.*;
import java.nio.channels.*;
import java.io.*;

public class MappedIO {
  private static int numOfInts =      4_000_000;
  private static int numOfUbuffInts = 100_000;
  private abstract static class Tester {
    private String name;
    Tester(String name) {
      this.name = name;
    }
    public void runTest() {
      System.out.print(name + ": ");
      long start = System.nanoTime();
      test();
      double duration = System.nanoTime() - start;
      System.out.format("%.3f%n", duration/1.0e9);
    }
    public abstract void test();
  }
  private static Tester[] tests = {
    new Tester("Stream Write") {
      @Override public void test() {
        try(
          DataOutputStream dos =
            new DataOutputStream(
```

```
            new BufferedOutputStream(
              new FileOutputStream(
                new File("temp.tmp"))))
      ) {
        for(int i = 0; i < numOfInts; i++)
          dos.writeInt(i);
      } catch(IOException e) {
        throw new RuntimeException(e);
      }
    }
  },
  new Tester("Mapped Write") {
    @Override public void test() {
      try(
        FileChannel fc =
          new RandomAccessFile("temp.tmp", "rw")
            .getChannel()
      ) {
        IntBuffer ib =
          fc.map(FileChannel.MapMode.READ_WRITE,
            0, fc.size()).asIntBuffer();
        for(int i = 0; i < numOfInts; i++)
          ib.put(i);
      } catch(IOException e) {
        throw new RuntimeException(e);
      }
    }
  },
  new Tester("Stream Read") {
    @Override public void test() {
      try(
        DataInputStream dis =
          new DataInputStream(
            new BufferedInputStream(
              new FileInputStream("temp.tmp")))
      ) {
        for(int i = 0; i < numOfInts; i++)
          dis.readInt();
      } catch(IOException e) {
        throw new RuntimeException(e);
      }
    }
```

```
  },
  new Tester("Mapped Read") {
    @Override public void test() {
      try(
        FileChannel fc = new FileInputStream(
          new File("temp.tmp")).getChannel()
      ) {
        IntBuffer ib =
          fc.map(FileChannel.MapMode.READ_ONLY,
            0, fc.size()).asIntBuffer();
        while(ib.hasRemaining())
          ib.get();
      } catch(IOException e) {
        throw new RuntimeException(e);
      }
    }
  },
  new Tester("Stream Read/Write") {
    @Override public void test() {
      try(
        RandomAccessFile raf =
          new RandomAccessFile(
            new File("temp.tmp"), "rw")
      ) {
        raf.writeInt(1);
        for(int i = 0; i < numOfUbuffInts; i++) {
          raf.seek(raf.length() - 4);
          raf.writeInt(raf.readInt());
        }
      } catch(IOException e) {
        throw new RuntimeException(e);
      }
    }
  },
  new Tester("Mapped Read/Write") {
    @Override public void test() {
      try(
        FileChannel fc = new RandomAccessFile(
          new File("temp.tmp"), "rw").getChannel()
      ) {
        IntBuffer ib =
          fc.map(FileChannel.MapMode.READ_WRITE,
```

```
              0, fc.size()).asIntBuffer();
          ib.put(0);
          for(int i = 1; i < numOfUbuffInts; i++)
            ib.put(ib.get(i - 1));
        } catch(IOException e) {
          throw new RuntimeException(e);
        }
      }
    }
  };
  public static void main(String[] args) {
    Arrays.stream(tests).forEach(Tester::runTest);
  }
}
/* Output:
Stream Write: 0.615
Mapped Write: 0.050
Stream Read: 0.577
Mapped Read: 0.015
Stream Read/Write: 4.069
Mapped Read/Write: 0.013
*/
```

`Tester` is a *Template Method* pattern that creates a test framework for various implementations of `test()` defined in anonymous inner subclasses. Each of these subclasses performs one kind of test, so the `test()` methods also give you a prototype for performing the various I/O activities.

Although a mapped write would seem to use a `FileOutputStream`, all output in file mapping must use a `RandomAccessFile`, just as read/write does in the preceding code.

Note that the `test()` methods include the time for initialization of the various I/O objects, so even though the setup for mapped files can be expensive, the overall gain compared to stream I/O is dramatic.

File Locking

File locking synchronizes access so a file can be a shared resource. However, two threads that contend for the same file might be in different JVMs, or one might be a

Java thread and the other some native thread in the operating system. The file locks are visible to other operating system processes because Java file locking maps directly to the native operating system locking facility.

This shows basic file locking:

```
// newio/FileLocking.java
import java.nio.channels.*;
import java.util.concurrent.*;
import java.io.*;

public class FileLocking {
  public static void main(String[] args) {
    try(
      FileOutputStream fos =
        new FileOutputStream("file.txt");
      FileLock fl = fos.getChannel().tryLock()
    ) {
      if(fl != null) {
        System.out.println("Locked File");
        TimeUnit.MILLISECONDS.sleep(100);
        fl.release();
        System.out.println("Released Lock");
      }
    } catch(IOException | InterruptedException e) {
      throw new RuntimeException(e);
    }
  }
}
/* Output:
Locked File
Released Lock
*/
```

You get a `FileLock` on the entire file by calling either `tryLock()` or `lock()` on a `FileChannel`. (`SocketChannel`, `DatagramChannel`, and `ServerSocketChannel` do not need locking because they are inherently single-process entities; you don't generally share a network socket between two processes.)

- `tryLock()` is non-blocking. It tries to grab the lock, but if it cannot (when some other process already holds the same lock and it is not shared), it simply returns from the method call.

- `lock()` blocks until the lock is acquired, or the thread that invoked `lock()` is interrupted, or the channel on which the `lock()` method is called is closed. A lock is released using `FileLock.release()`.

It is also possible to lock a part of the file by using

```
tryLock(long position, long size, boolean shared)
```

or

```
lock(long position, long size, boolean shared)
```

which locks the region (`size-position`). The third argument specifies whether this lock is shared.

Although the zero-argument locking methods adapt to changes in the size of a file, locks with a fixed size do not change if the file size changes. If a lock is acquired for a region from `position` to `position + size` and the file increases beyond `position + size`, then the section beyond `position + size` is not locked. The zero-argument locking methods lock the entire file, even if it grows.

Support for exclusive or shared locks must be provided by the underlying operating system. If the operating system does not support shared locks and a request is made for one, an exclusive lock is used instead. The type of lock (shared or exclusive) can be queried using `FileLock.isShared()`.

Locking Portions of a Mapped File

File mapping is typically used for very large files. You might need to lock portions of such a file so other processes can modify unlocked parts. A database, for example, must be available to many users at once.

Here you see two threads, each of which locks a distinct portion of a file:

```
// newio/LockingMappedFiles.java
// Locking portions of a mapped file
import java.nio.*;
import java.nio.channels.*;
import java.io.*;

public class LockingMappedFiles {
  static final int LENGTH = 0x8FFFFFF; // 128 MB
  static FileChannel fc;
  public static void
  main(String[] args) throws Exception {
    fc = new RandomAccessFile("test.dat", "rw")
      .getChannel();
    MappedByteBuffer out = fc.map(
      FileChannel.MapMode.READ_WRITE, 0, LENGTH);
    for(int i = 0; i < LENGTH; i++)
      out.put((byte)'x');
    new LockAndModify(out, 0, 0 + LENGTH/3);
    new LockAndModify(
      out, LENGTH/2, LENGTH/2 + LENGTH/4);
  }
  private static class LockAndModify extends Thread {
    private ByteBuffer buff;
    private int start, end;
    LockAndModify(ByteBuffer mbb, int start, int end) {
      this.start = start;
      this.end = end;
      mbb.limit(end);
      mbb.position(start);
      buff = mbb.slice();
      start();
    }
    @Override public void run() {
      try {
        // Exclusive lock with no overlap:
        FileLock fl = fc.lock(start, end, false);
        System.out.println(
          "Locked: "+ start +" to "+ end);
        // Perform modification:
        while(buff.position() < buff.limit() - 1)
          buff.put((byte)(buff.get() + 1));
        fl.release();
```

```
        System.out.println(
          "Released: " + start + " to " + end);
      } catch(IOException e) {
        throw new RuntimeException(e);
      }
    }
  }
}
/* Output:
Locked: 75497471 to 113246206
Locked: 0 to 50331647
Released: 75497471 to 113246206
Released: 0 to 50331647
*/
```

The `LockAndModify` thread class sets up the buffer region and creates a `slice()` to be modified, and in `run()`, the lock is acquired on the file channel (you can't acquire a lock on the buffer—only the channel). The call to `lock()` is very similar to acquiring a threading lock on an object—you now have a "critical section" with exclusive access to that portion of the file.[194]

The locks are automatically released when the JVM exits, or the channel on which it was acquired is closed, but you can also explicitly call `release()` on the `FileLock` object, as shown here.

[194] You can find more details about threads in the Appendix: Low-Level Concurrency.

Appendix: Understanding `equals()` and `hashCode()`

When you create a class for use in any container that utilizes *hashing*, you must not only define the `hashCode()` method (which we shall explore later in this appendix), but also the `equals()` method. The two methods are used together to perform a lookup into a hashed container.

A Canonical `equals()`

When you create a new class, it automatically inherits class `Object`. If you don't override `equals()`, you'll get `Object`'s `equals()` method. By default this compares addresses, so only if you are comparing the *exact same* objects will you get `true`. The default case is the "most discriminating."

```
// equalshashcode/DefaultComparison.java

class DefaultComparison {
  private int i, j, k;
  DefaultComparison(int i, int j, int k) {
    this.i = i;
    this.j = j;
    this.k = k;
  }
  public static void main(String[] args) {
    DefaultComparison
      a = new DefaultComparison(1, 2, 3),
      b = new DefaultComparison(1, 2, 3);
    System.out.println(a == a);
    System.out.println(a == b);
  }
}
/* Output:
```

```
true
false
*/
```

Normally you'll want to relax this restriction. Typically, if two objects are the same type and have fields with identical values, you'll consider those objects equal, but there may also be fields that you don't want to include in the `equals()` comparison. This is part of the class design process.

A proper `equals()` must satisfy the following five conditions:

1. Reflexive: For any `x`, `x.equals(x)` should return `true`.
2. Symmetric: For any `x` and `y`, `x.equals(y)` should return `true` if and only if `y.equals(x)` returns `true`.
3. Transitive: For any `x`, `y`, and `z`, if `x.equals(y)` returns `true` and `y.equals(z)` returns `true`, then `x.equals(z)` should return `true`.
4. Consistent: For any `x` and `y`, multiple invocations of `x.equals(y)` consistently return `true` or consistently return `false`, provided no information used in equals comparisons on the object is modified.
5. For any non-`null` `x`, `x.equals(null)` should return `false`.

Here are the tests that satisfy those conditions and determine whether the object you're comparing yourself to (which we'll call here the **rval**) is equal to this object:

1. If the **rval** is `null`, it's not equal.
2. If the **rval** is `this` (you're comparing yourself to yourself), the two objects are equal.
3. If the **rval** is not the same class or subclass, the two objects are not equal.
4. If all the above checks pass, then you must decide which fields in the **rval** are important (and consistent), and compare those.

Java 7 introduced the `Objects` class to help with this process, which we use to write a better `equals()`.

The following examples compare different versions of the `Equality` class. To prevent duplicate code we'll build the examples using the *Factory Method* (see Factories: Encapsulating Object Creation). The `EqualityFactory` interface simply provides a `make()` method to produce an `Equality` object, so a different `EqualityFactory` can produce a different subtype of `Equality`:

```
// equalshashcode/EqualityFactory.java
import java.util.*;

interface EqualityFactory {
  Equality make(int i, String s, double d);
}
```

Now we'll define `Equality` containing three fields (all of which we consider important during comparison) and an `equals()` method that fulfills the four checks described above. The constructor displays its type name to ensure we are performing the tests we think we are:

```
// equalshashcode/Equality.java
import java.util.*;

public class Equality {
  protected int i;
  protected String s;
  protected double d;
  public Equality(int i, String s, double d) {
    this.i = i;
    this.s = s;
    this.d = d;
    System.out.println("made 'Equality'");
  }
  @Override public boolean equals(Object rval) {
    if(rval == null)
      return false;
    if(rval == this)
      return true;
    if(!(rval instanceof Equality))
      return false;
    Equality other = (Equality)rval;
    if(!Objects.equals(i, other.i))
      return false;
    if(!Objects.equals(s, other.s))
      return false;
    if(!Objects.equals(d, other.d))
      return false;
    return true;
  }
  public void
```

```
  test(String descr, String expected, Object rval) {
    System.out.format("-- Testing %s --%n" +
      "%s instanceof Equality: %s%n" +
      "Expected %s, got %s%n",
      descr, descr, rval instanceof Equality,
      expected, equals(rval));
  }
  public static void testAll(EqualityFactory eqf) {
    Equality
      e = eqf.make(1, "Monty", 3.14),
      eq = eqf.make(1, "Monty", 3.14),
      neq = eqf.make(99, "Bob", 1.618);
    e.test("null", "false", null);
    e.test("same object", "true", e);
    e.test("different type",
           "false", Integer.valueOf(99));
    e.test("same values", "true", eq);
    e.test("different values", "false", neq);
  }
  public static void main(String[] args) {
    testAll( (i, s, d) -> new Equality(i, s, d));
  }
}
/* Output:
made 'Equality'
made 'Equality'
made 'Equality'
-- Testing null --
null instanceof Equality: false
Expected false, got false
-- Testing same object --
same object instanceof Equality: true
Expected true, got true
-- Testing different type --
different type instanceof Equality: false
Expected false, got false
-- Testing same values --
same values instanceof Equality: true
Expected true, got true
-- Testing different values --
different values instanceof Equality: true
Expected false, got false
```

```
*/
```

`testAll()` performs comparisons with all different types of objects we ever expect to encounter. It creates `Equality` objects using the factory.

In `main()`, notice the simplicity of the call to `testAll()`. Because `EqualityFactory` has a single method, it can be used with a lambda expression as the `make()` method.

The above `equals()` method is annoyingly verbose, and it turns out we can simplify it into a canonical form. Observe:

1. The `instanceof` check eliminates the need to test for `null`
2. The comparison to `this` is redundant. A correctly-written `equals()` will work properly with self comparison.

Because `&&` is a short-circuiting comparison, it quits and produces `false` the first time it encounters a failure. So, by chaining the checks together with `&&`, we can write `equals()` much more succinctly:

```
// equalshashcode/SuccinctEquality.java
import java.util.*;

public class SuccinctEquality extends Equality {
  public SuccinctEquality(int i, String s, double d) {
    super(i, s, d);
    System.out.println("made 'SuccinctEquality'");
  }
  @Override public boolean equals(Object rval) {
    return rval instanceof SuccinctEquality &&
      Objects.equals(i, ((SuccinctEquality)rval).i) &&
      Objects.equals(s, ((SuccinctEquality)rval).s) &&
      Objects.equals(d, ((SuccinctEquality)rval).d);
  }
  public static void main(String[] args) {
    Equality.testAll( (i, s, d) ->
      new SuccinctEquality(i, s, d));
  }
}
/* Output:
made 'Equality'
made 'SuccinctEquality'
made 'Equality'
```

```
made 'SuccinctEquality'
made 'Equality'
made 'SuccinctEquality'
-- Testing null --
null instanceof Equality: false
Expected false, got false
-- Testing same object --
same object instanceof Equality: true
Expected true, got true
-- Testing different type --
different type instanceof Equality: false
Expected false, got false
-- Testing same values --
same values instanceof Equality: true
Expected true, got true
-- Testing different values --
different values instanceof Equality: true
Expected false, got false
*/
```

For each `SuccinctEquality`, the base-class constructor is called before the derived-class constructor. The output shows that we still get the correct result. You can tell that short-circuiting happens because both the `null` test and the "different type" test would otherwise throw exceptions during the casts that occur further down the list of comparisons in `equals()`.

`Objects.equals()` shines when you compose your new class using another class:

```
// equalshashcode/ComposedEquality.java
import java.util.*;

class Part {
  String ss;
  double dd;
  Part(String ss, double dd) {
    this.ss = ss;
    this.dd = dd;
  }
  @Override public boolean equals(Object rval) {
    return rval instanceof Part &&
      Objects.equals(ss, ((Part)rval).ss) &&
      Objects.equals(dd, ((Part)rval).dd);
```

```
  }
}

public class ComposedEquality extends SuccinctEquality {
  Part part;
  public ComposedEquality(int i, String s, double d) {
    super(i, s, d);
    part = new Part(s, d);
    System.out.println("made 'ComposedEquality'");
  }
  @Override public boolean equals(Object rval) {
    return rval instanceof ComposedEquality &&
      super.equals(rval) &&
      Objects.equals(part,
        ((ComposedEquality)rval).part);
  }
  public static void main(String[] args) {
    Equality.testAll( (i, s, d) ->
      new ComposedEquality(i, s, d));
  }
}
/* Output:
made 'Equality'
made 'SuccinctEquality'
made 'ComposedEquality'
made 'Equality'
made 'SuccinctEquality'
made 'ComposedEquality'
made 'Equality'
made 'SuccinctEquality'
made 'ComposedEquality'
-- Testing null --
null instanceof Equality: false
Expected false, got false
-- Testing same object --
same object instanceof Equality: true
Expected true, got true
-- Testing different type --
different type instanceof Equality: false
Expected false, got false
-- Testing same values --
same values instanceof Equality: true
```

```
Expected true, got true
-- Testing different values --
different values instanceof Equality: true
Expected false, got false
*/
```

Notice the call to `super.equals()`—no need to reinvent it (plus you don't always have access to all necessary parts of a base class).

Equality Across Subtypes

Inheritance suggests that objects of two different subtypes can be "the same" when they are upcast. Suppose you have a collection of `Animal` objects. This collection naturally accepts subtypes of `Animal`—in this example, `Dogs` and `Pigs`. Each `Animal` has a `name` and a `size`, as well as a unique internal `id` number.

We define `equals()` and `hashCode()` using the canonical form via the `Objects` class, but we only define them in the base class `Animal`, and we do not include the unique `id` number in either one. From the standpoint of `equals()`, this means we only care if something is a `Animal`, not whether it is a specific type of `Animal`:

```
// equalshashcode/SubtypeEquality.java
import java.util.*;

enum Size { SMALL, MEDIUM, LARGE }

class Animal {
  private static int counter = 0;
  private final int id = counter++;
  private final String name;
  private final Size size;
  Animal(String name, Size size) {
    this.name = name;
    this.size = size;
  }
  @Override public boolean equals(Object rval) {
    return rval instanceof Animal &&
      // Objects.equals(id, ((Animal)rval).id) && // [1]
      Objects.equals(name, ((Animal)rval).name) &&
      Objects.equals(size, ((Animal)rval).size);
```

```
  }
  @Override public int hashCode() {
    return Objects.hash(name, size);
    // return Objects.hash(name, size, id);       // [2]
  }
  @Override public String toString() {
    return String.format("%s[%d]: %s %s %x",
      getClass().getSimpleName(), id,
      name, size, hashCode());
  }
}

class Dog extends Animal {
  Dog(String name, Size size) {
    super(name, size);
  }
}

class Pig extends Animal {
  Pig(String name, Size size) {
    super(name, size);
  }
}

public class SubtypeEquality {
  public static void main(String[] args) {
    Set<Animal> pets = new HashSet<>();
    pets.add(new Dog("Ralph", Size.MEDIUM));
    pets.add(new Pig("Ralph", Size.MEDIUM));
    pets.forEach(System.out::println);
  }
}
/* Output:
Dog[0]: Ralph MEDIUM 931523a9
*/
```

If we are just thinking about types, it does make sense—sometimes—to only consider the classes from the standpoint of their base type, which is the foundation of the *Liskov Substitution Principle*. This code fits nicely with that principle because the derived types don't add any extra functionality (methods) that isn't in the base class. The derived types only differ in behavior, not in interface (which of course is not the

general case).

But when we provide two different object types with identical data and place them in a HashSet<Animal>, only one of these objects survives. This emphasizes that equals() is not a perfectly mathematical concept but (at least partially) a mechanical one. hashCode() and equals() must be defined hand-in-hand to allow types to work properly in a hashed data structure.

In the example, both the Dog and Pig hash to the same bucket in the HashSet. At this point, the HashSet falls back to equals() to differentiate the objects, but equals() also declares the objects to be the same. The HashSet doesn't add the Pig because it's already got an identical object.

We can still make the example work by forcing uniqueness on otherwise identical objects. Here, each Animal already has a unique id so you can either uncomment line **[1]** in equals() or switch to line **[2]** in hashCode(). In the canonical form you would do both, to involve all "unchanging" fields in both operations ("unchanging" so that the equals() and hashCode() don't produce different values between storing and retrieving in a hashed data structure. I put "unchanging" in quotes because you must evaluate whether modification might happen).

Side note: in hashCode(), if you are only working with a single field, use Objects.hashCode() and if you are using multiple fields use Objects.hash().

We can also solve the issue by following the standard form and defining equals() in the subclasses (but still not including the unique id):

```
// equalshashcode/SubtypeEquality2.java
import java.util.*;

class Dog2 extends Animal {
  Dog2(String name, Size size) {
    super(name, size);
  }
  @Override public boolean equals(Object rval) {
    return rval instanceof Dog2 &&
      super.equals(rval);
  }
}

class Pig2 extends Animal {
```

```
  Pig2(String name, Size size) {
    super(name, size);
  }
  @Override public boolean equals(Object rval) {
    return rval instanceof Pig2 &&
      super.equals(rval);
  }
}

public class SubtypeEquality2 {
  public static void main(String[] args) {
    Set<Animal> pets = new HashSet<>();
    pets.add(new Dog2("Ralph", Size.MEDIUM));
    pets.add(new Pig2("Ralph", Size.MEDIUM));
    pets.forEach(System.out::println);
  }
}
/* Output:
Dog2[0]: Ralph MEDIUM 931523a9
Pig2[1]: Ralph MEDIUM 931523a9
*/
```

Notice that the `hashCode()`s are identical, but because the objects are no longer `equals()`, both now appear in the `HashSet`. Also, `super.equals()` means we don't need access to the `private` fields in the base class.

One way to look at this is to say that Java separates substitutability from the definition of `equals()` and `hashCode()`. We can still place `Dogs` and `Pigs` into a `Set<Animal>` regardless of how `equals()` and `hashCode()` are defined, but the objects won't behave correctly in hashed data structures unless those methods are defined with hashed structures in mind. Unfortunately, `equals()` is not only used in conjunction with `hashCode()`. This complicates things when you try to avoid defining it for specific classes, and it's why it's worth following the canonical form. However, this is further complicated because there are times when you don't need to define either method.

Hashing and Hash Codes

The examples in the Collections chapter used predefined classes as HashMap keys. These examples worked because the predefined classes had all the necessary wiring to make them behave correctly as keys.

A common pitfall occurs when you create your own classes as keys for HashMaps, and forget to put in the necessary wiring. For example, consider a weather predicting system that matches Groundhog objects to Prediction objects. This seems fairly straightforward: use Groundhog as the key and Prediction as the value:

```
// equalshashcode/Groundhog.java
// Looks plausible, but doesn't work as a HashMap key

public class Groundhog {
  protected int number;
  public Groundhog(int n) { number = n; }
  @Override public String toString() {
    return "Groundhog #" + number;
  }
}


// equalshashcode/Prediction.java
// Predicting the weather
import java.util.*;

public class Prediction {
  private static Random rand = new Random(47);
  @Override public String toString() {
    return rand.nextBoolean() ?
      "Six more weeks of Winter!" : "Early Spring!";
  }
}
```

```
// equalshashcode/SpringDetector.java
// What will the weather be?
import java.util.*;
import java.util.stream.*;
import java.util.function.*;
import java.lang.reflect.*;

public class SpringDetector {
  public static <T extends Groundhog>
  void detectSpring(Class<T> type) {
    try {
      Constructor<T> ghog =
        type.getConstructor(int.class);
      Map<Groundhog, Prediction> map =
        IntStream.range(0, 10)
        .mapToObj(i -> {
          try {
            return ghog.newInstance(i);
          } catch(Exception e) {
            throw new RuntimeException(e);
          }
        })
        .collect(Collectors.toMap(
          Function.identity(),
          gh -> new Prediction()));
      map.forEach((k, v) ->
        System.out.println(k + ": " + v));
      Groundhog gh = ghog.newInstance(3);
      System.out.println(
        "Looking up prediction for " + gh);
      if(map.containsKey(gh))
        System.out.println(map.get(gh));
      else
        System.out.println("Key not found: " + gh);
    } catch(NoSuchMethodException |
            IllegalAccessException |
            InvocationTargetException |
            InstantiationException e) {
      throw new RuntimeException(e);
    }
  }
  public static void main(String[] args) {
```

```
    detectSpring(Groundhog.class);
  }
}
/* Output:
Groundhog #5: Six more weeks of Winter!
Groundhog #2: Early Spring!
Groundhog #1: Six more weeks of Winter!
Groundhog #0: Early Spring!
Groundhog #8: Early Spring!
Groundhog #6: Six more weeks of Winter!
Groundhog #4: Early Spring!
Groundhog #3: Early Spring!
Groundhog #7: Six more weeks of Winter!
Groundhog #9: Six more weeks of Winter!
Looking up prediction for Groundhog #3
Key not found: Groundhog #3
*/
```

Each `Groundhog` is given an identity number, so you can look up a `Prediction` in the `HashMap` by saying, "Give me the `Prediction` associated with `Groundhog #3`." The `Prediction` chooses the weather by generating a random `boolean`. The `detectSpring()` method uses reflection to instantiate and use the `class Groundhog` or any class derived from `Groundhog`. This comes in handy later, when we inherit a new type of `Groundhog` to solve the problem demonstrated here.

A `HashMap` is filled with `Groundhogs` and their associated `Predictions`. Displaying `HashMap` shows it was filled. Then a `Groundhog` with an identity number of `3` is used as a key to find the prediction for `Groundhog #3` (which therefore must be in the `Map`).

It seems simple enough, but it doesn't work—it can't find the key for `#3`. The problem is that `Groundhog` automatically inherits class `Object`, and it is `Object`'s `hashCode()` method that is used to generate the hash code for each object. By default this just uses the address of its object. Thus, the first instance of `Groundhog(3)` does *not* produce a hash code equal to the hash code for the second instance of `Groundhog(3)` that we tried to use as a lookup.

We need an appropriate override for `hashCode()`. But that still won't work until you've done one more thing: override the `equals()` that is also part of `Object`. `equals()` is used by the `HashMap` when trying to determine if your key is equal to any of the keys in the table.

Because the default Object.equals() compares object addresses, one Groundhog(3) is not equal to another Groundhog(3). Thus, to use your own classes as keys in a HashMap, you must override both hashCode() and equals(), as shown in the following solution to the groundhog problem:

```
// equalshashcode/Groundhog2.java
// A class that's used as a key in a HashMap
// must override hashCode() and equals()
import java.util.*;

public class Groundhog2 extends Groundhog {
  public Groundhog2(int n) { super(n); }
  @Override
  public int hashCode() { return number; }
  @Override public boolean equals(Object o) {
    return o instanceof Groundhog2 &&
      Objects.equals(
        number, ((Groundhog2)o).number);
  }
}


// equalshashcode/SpringDetector2.java
// A working key

public class SpringDetector2 {
  public static void main(String[] args) {
    SpringDetector.detectSpring(Groundhog2.class);
  }
}
/* Output:
Groundhog #0: Six more weeks of Winter!
Groundhog #1: Early Spring!
Groundhog #2: Six more weeks of Winter!
Groundhog #3: Early Spring!
Groundhog #4: Early Spring!
Groundhog #5: Six more weeks of Winter!
Groundhog #6: Early Spring!
Groundhog #7: Early Spring!
Groundhog #8: Six more weeks of Winter!
Groundhog #9: Six more weeks of Winter!
Looking up prediction for Groundhog #3
```

```
Early Spring!
*/
```

`Groundhog2.hashCode()` returns the groundhog number as a hash value. In this example, the programmer is responsible for ensuring that no two groundhogs exist with the same ID number. The `hashCode()` is not required to return a unique identifier (something you'll understand later in this appendix), but the `equals()` method must strictly determine whether two objects are equivalent. Here, `equals()` is based on the groundhog number, so if two `Groundhog2` objects exist as keys in the `HashMap` with the same groundhog number, it will fail.

Defining `equals()` was covered in A Canonical `equals()`. The output shows that the behavior is now correct.

Understanding `hashCode()`

The preceding example is only a start toward solving the problem correctly. It shows that if you do not override `hashCode()` and `equals()` for your key, the hashed data structure (for example: `HashSet`, `HashMap`, `LinkedHashSet`, or `LinkedHashMap`) probably won't deal with your key properly. For a *good* solution to the problem, however, you must understand what's going on inside the hashed data structure.

First, consider the motivation behind hashing: To look up an object using another object. But you can also accomplish this with a `TreeMap`, or you can even implement your own `Map`. In contrast to a hashed implementation, the following example implements a `Map` using a pair of `ArrayLists`. Unlike `AssociativeArray.java`, this includes a full implementation of the `Map` interface, which accounts for the `entrySet()` method:

```
// equalshashcode/SlowMap.java
// A Map implemented with ArrayLists
import java.util.*;
import onjava.*;

public class SlowMap<K, V> extends AbstractMap<K, V> {
  private List<K> keys = new ArrayList<>();
  private List<V> values = new ArrayList<>();
  @Override public V put(K key, V value) {
    V oldValue = get(key); // The old value or null
    if(!keys.contains(key)) {
      keys.add(key);
      values.add(value);
    } else
      values.set(keys.indexOf(key), value);
    return oldValue;
  }
  @Override
  public V get(Object key) { // key: type Object, not K
    if(!keys.contains(key))
      return null;
    return values.get(keys.indexOf(key));
  }
  @Override public Set<Map.Entry<K, V>> entrySet() {
    Set<Map.Entry<K, V>> set= new HashSet<>();
    Iterator<K> ki = keys.iterator();
    Iterator<V> vi = values.iterator();
    while(ki.hasNext())
      set.add(new MapEntry<>(ki.next(), vi.next()));
    return set;
  }
  public static void main(String[] args) {
    SlowMap<String,String> m= new SlowMap<>();
    m.putAll(Countries.capitals(8));
    m.forEach((k, v) ->
      System.out.println(k + "=" + v));
    System.out.println(m.get("BENIN"));
    m.entrySet().forEach(System.out::println);
  }
}
/* Output:
CAMEROON=Yaounde
```

```
ANGOLA=Luanda
BURKINA FASO=Ouagadougou
BURUNDI=Bujumbura
ALGERIA=Algiers
BENIN=Porto-Novo
CAPE VERDE=Praia
BOTSWANA=Gaberone
Porto-Novo
CAMEROON=Yaounde
ANGOLA=Luanda
BURKINA FASO=Ouagadougou
BURUNDI=Bujumbura
ALGERIA=Algiers
BENIN=Porto-Novo
CAPE VERDE=Praia
BOTSWANA=Gaberone
*/
```

The `put()` method places the keys and values in corresponding `ArrayLists`. In accordance with the `Map` interface, it must return the old key or `null` if there was no old key.

Following the specifications for `Map`, `get()` produces `null` if the key is not in the `SlowMap`. If the key exists, it is used to look up the numerical index indicating its location in the `keys List`, and this number is used as an index to produce the associated value from the `values List`. Notice that the type of `key` is `Object` in `get()`, rather than the parameterized type `K` as you might expect (and which was indeed used in `AssociativeArray.java`). This is a result of the injection of generics into the Java language at such a late date—if generics had been an original feature in the language, `get()` could have specified the type of its parameter.

The `String` representation of the contents of `SlowMap` is automatically produced by the `toString()` method defined in `AbstractMap`.

In `SlowMap.main()`, a `SlowMap` is loaded, then the contents are displayed. A call to `get()` shows it works.

`Map.entrySet()` produces a set of `Map.Entry` objects. However, `Map.Entry` is an interface describing an implementation-dependent structure, so to make your own type of `Map`, you must also define an implementation of `Map.Entry`:

```
// equalshashcode/MapEntry.java
// A simple Map.Entry for sample Map implementations
import java.util.*;

public class MapEntry<K, V> implements Map.Entry<K, V> {
  private K key;
  private V value;
  public MapEntry(K key, V value) {
    this.key = key;
    this.value = value;
  }
  @Override public K getKey() { return key; }
  @Override public V getValue() { return value; }
  @Override public V setValue(V v) {
    V result = value;
    value = v;
    return result;
  }
  @Override public int hashCode() {
    return Objects.hash(key, value);
  }
  @SuppressWarnings("unchecked")
  @Override public boolean equals(Object rval) {
    return rval instanceof MapEntry &&
      Objects.equals(key,
        ((MapEntry<K, V>)rval).getKey()) &&
      Objects.equals(value,
        ((MapEntry<K, V>)rval).getValue());
  }
  @Override public String toString() {
    return key + "=" + value;
  }
}
```

The `equals()` method follows A Canonical `equals()`. There is a similar method in the `Objects` class for creating a `hashCode()`: `Objects.hash()`. Use this when you are defining a `hashCode()` involving *more than one field*. If you are only using a single field, use `Objects.hashCode()` instead.

This simple solution appears to work for the trivial test in `SlowMap.main()`, but it is not a correct implementation because it makes a copy of the keys and values. A

proper implementation of entrySet() provides a *view* into the Map, rather than a copy, and this view allows modification of the original map (which a copy doesn't).

Hashing for Speed

SlowMap.java shows it's not that hard to produce a new type of Map. But as the name suggests, a SlowMap isn't very fast, so you won't use it if you have an alternative. The problem happens during lookup: the keys are not kept in any particular order, so a simple linear search is used. A linear search is the slowest way to find something.

The whole point of hashing is speed, because hashing looks up values very quickly. The bottleneck is in the speed of the key lookup so one solution is to keep the keys sorted, then use Collections.binarySearch() to perform the lookup.

Hashing goes further by only storing the key *somewhere* in a way it can be found quickly. The fastest structure for storing a group of elements is an array, so that is used for representing the key information (note I say "key information," and not the key itself). But because an array cannot be resized, we have a problem: We want to store an indeterminate number of values in the Map, but if the number of keys is fixed by the array size, how can this be?

The array does not hold the keys. From the key object, a number is derived to index into the array. This number is the *hash code*, produced by the hashCode() method (using a *hash function*) defined in Object and presumably overridden by your class.

To solve the problem of the fixed-size array, more than one key can produce the same index. That is, there can be *collisions*. Because of this, it doesn't matter how big the array is; any key object's hash code will land somewhere in that array.

So the process of looking up a value starts by computing the hash code and using it to index into the array. If you could guarantee there were no collisions (possible with a fixed number of values), you'd have a *perfect hashing function*, but that's a special case.[195] In all other cases, collisions are handled with *external chaining*: The array doesn't point directly to a value, but instead to a list of values. These values are searched in a linear fashion using the equals() method (thus, equals()

[195]Perfect hashing functions are implemented in EnumMap and EnumSet, because an enum defines a fixed number of instances. See the Enumerations chapter.

is also essential for hashing). This aspect of the search is much slower, but if the hash function is good, there will only be a few values in each slot. So instead of searching through the entire list, you quickly jump to a slot where you only compare a few entries to find the value. This is much faster, which is why a HashMap is so quick.

Knowing the basics of hashing, you can implement a simple hashed Map:

```
// equalshashcode/SimpleHashMap.java
// A demonstration hashed Map
import java.util.*;
import onjava.*;

public
class SimpleHashMap<K, V> extends AbstractMap<K, V> {
  // Choose a prime number for the hash table
  // size, to achieve a uniform distribution:
  static final int SIZE = 997;
  // You can't have a physical array of generics,
  // but you can upcast to one:
  @SuppressWarnings("unchecked")
  LinkedList<MapEntry<K, V>>[] buckets =
    new LinkedList[SIZE];
  @Override public V put(K key, V value) {
    V oldValue = null;
    int index = Math.abs(key.hashCode()) % SIZE;
    if(buckets[index] == null)
      buckets[index] = new LinkedList<>();
    LinkedList<MapEntry<K, V>> bucket = buckets[index];
    MapEntry<K, V> pair = new MapEntry<>(key, value);
    boolean found = false;
    ListIterator<MapEntry<K, V>> it =
      bucket.listIterator();
    while(it.hasNext()) {
      MapEntry<K, V> iPair = it.next();
      if(iPair.getKey().equals(key)) {
        oldValue = iPair.getValue();
        it.set(pair); // Replace old with new
        found = true;
        break;
      }
    }
    if(!found)
```

```
      buckets[index].add(pair);
    return oldValue;
  }
  @Override public V get(Object key) {
    int index = Math.abs(key.hashCode()) % SIZE;
    if(buckets[index] == null) return null;
    for(MapEntry<K, V> iPair : buckets[index])
      if(iPair.getKey().equals(key))
        return iPair.getValue();
    return null;
  }
  @Override public Set<Map.Entry<K, V>> entrySet() {
    Set<Map.Entry<K, V>> set= new HashSet<>();
    for(LinkedList<MapEntry<K, V>> bucket : buckets) {
      if(bucket == null) continue;
      for(MapEntry<K, V> mpair : bucket)
        set.add(mpair);
    }
    return set;
  }
  public static void main(String[] args) {
    SimpleHashMap<String,String> m =
      new SimpleHashMap<>();
    m.putAll(Countries.capitals(8));
    m.forEach((k, v) ->
      System.out.println(k + "=" + v));
    System.out.println(m.get("BENIN"));
    m.entrySet().forEach(System.out::println);
  }
}
/* Output:
CAMEROON=Yaounde
ANGOLA=Luanda
BURKINA FASO=Ouagadougou
BURUNDI=Bujumbura
ALGERIA=Algiers
BENIN=Porto-Novo
CAPE VERDE=Praia
BOTSWANA=Gaberone
Porto-Novo
CAMEROON=Yaounde
ANGOLA=Luanda
```

```
BURKINA FASO=Ouagadougou
BURUNDI=Bujumbura
ALGERIA=Algiers
BENIN=Porto-Novo
CAPE VERDE=Praia
BOTSWANA=Gaberone
*/
```

Because the "slots" in a hash table are often called *buckets*, the array that represents the actual table is called `buckets`. To promote even distribution, the number of buckets is typically a prime number.[196] Notice it is an array of `LinkedList`, which automatically provides for collisions—each new item is added to the end of the list in a particular bucket. Even though Java will not let you create an array of generics, it is possible to make a *reference* to such an array. Here, it is convenient to upcast to such an array, to prevent extra casting later in the code.

For a `put()`, `hashCode()` is called for the key and the result is forced to a positive number. To fit the resulting number into the `buckets` array, the modulus operator is used with the size of that array. If that location is `null`, it means there are no elements that hash to that location, so a new `LinkedList` holds the object that did just hash to that location. However, the normal process is to look through the list to see if there are duplicates, and if there are, the old value is put into `oldValue` and the new value replaces the old. The `found` flag keeps track of whether an old key-value pair was found and, if not, the new pair is appended to the end of the list.

`get()` calculates the index into the `buckets` array in the same fashion as `put()` (this is important to guarantee that you end up in the same spot). If a `LinkedList` exists, it is searched for a match.

This particular implementation is not meant to be tuned for performance; it is only intended to show the operations performed by a hash map. If you look at the source code for `java.util.HashMap`, you'll see a tuned implementation. Also, for simplicity, `SimpleHashMap` uses the same approach to `entrySet()` as did `SlowMap`, which is oversimplified and will not work for a general-purpose `Map`.

[196] A prime number is not actually the ideal size for hash buckets, and hashed implementations in Java use a power-of-two size (after extensive testing). Division or remainder is the slowest operation on a modern processor. With a power-of-two hash table length, masking can be used instead of division. `get()` is by far the most common operation so the `%` calculation is a large part of the cost, and the power-of-two approach eliminates this (but may also affect some `hashCode()` methods).

Overriding hashCode()

Now you understand how hashing works, writing a proper hashCode() method will make more sense.

First of all, you don't control the creation of the actual value that's used to index into the array of buckets. That is dependent on the capacity of the particular HashMap object, and that capacity changes depending on how full the collection is, along with the *load factor* (this term is described later). Thus, the value produced by your hashCode() is further processed to create the bucket index (in SimpleHashMap, the calculation is just a modulo by the size of the bucket array).

The most important factor in creating a hashCode() is that, regardless of when hashCode() is called, it produces the same value for a particular object every time. If you end up with an object that produces one hashCode() value when it is put() into a HashMap and another during a get(), you can't retrieve the objects. So if your hashCode() depends on mutable data in the object, the user must be made aware that changing the data produces a different key because it generates a different hashCode().

In addition, you probably *won't* want to generate a hashCode() based on unique object information—in particular, the value of this makes a bad hashCode() because then you can't generate a new key identical to the one used to put() the original key-value pair. This was the problem that occurred in SpringDetector.java, because the default implementation of hashCode() *does* use the object address. So use information in the object that identifies the object in a meaningful way.

One example can be seen in the String class. Strings have the special characteristic that if a program has several String objects that contain identical character sequences, those String objects all map to the same memory. So it makes sense that the hashCode() produced by two separate instances of the String "hello" should be identical, as demonstrated here:

```
// equalshashcode/StringHashCode.java

public class StringHashCode {
  public static void main(String[] args) {
    String[] hellos = "Hello Hello".split(" ");
    System.out.println(hellos[0].hashCode());
    System.out.println(hellos[1].hashCode());
  }
}
/* Output:
69609650
69609650
*/
```

The hashCode() for String is clearly based on the contents of the String.

So, for a hashCode() to be effective, it must be both fast and meaningful; that is, it must generate a value based on the contents of the object. Remember this value doesn't have to be unique—you should lean toward speed rather than uniqueness—but between hashCode() and equals(), the identity of the object must be completely resolved.

Because the hashCode() is further processed before the bucket index is produced, the range of values is not important; it must only generate an int.

There's one other factor: A good hashCode() should result in an even distribution of values. If the values tend to cluster, then the HashMap or HashSet is more heavily loaded in some areas and is not as fast as it can be with an evenly distributed hashing function.

In *Effective Java Programming Language Guide* (Addison-Wesley, 2001), Joshua Bloch gives a basic recipe for generating a decent hashCode():

1. Store some constant nonzero value, say 17, in an int variable called result.
2. For each significant field f in your object (that is, each field taken into account by the equals() method), calculate an int hash code c for the field as shown in the subsequent table.
3. Combine the hash code(s) computed above: result = 37 * result + c;
4. Return result.
5. Look at the resulting hashCode() and make sure that equal instances have equal hash codes.

Field type	Calculation
boolean	c = (f ? 0 : 1)
byte, char, short, or int	c = (int)f
long	c = (int)(f ^ (f>>>32))
float	c = Float.floatToIntBits(f);
double	long l = Double.doubleToLongBits(f); c = (int)(l ^ (l >>> 32))
Object, where equals() calls equals() for this field	c = f.hashCode()
Array	Apply above rules to each element

Here's an example that follows these guidelines. Note that you shouldn't actually write code like this—instead, use Objects.hash() for hashing together multiple fields (as in this case) and Objects.hashCode() for hashing a single field.

```
// equalshashcode/CountedString.java
// Creating a good hashCode()
import java.util.*;

public class CountedString {
  private static List<String> created =
    new ArrayList<>();
  private String s;
  private int id = 0;
  public CountedString(String str) {
    s = str;
    created.add(s);
    // id is the total number of instances
    // of this String used by CountedString:
    for(String s2 : created)
      if(s2.equals(s))
        id++;
  }
  @Override public String toString() {
```

```
    return "String: " + s + " id: " + id +
      " hashCode(): " + hashCode();
  }
  @Override public int hashCode() {
    // The very simple approach:
    // return s.hashCode() * id;
    // Using Joshua Bloch's recipe:
    int result = 17;
    result = 37 * result + s.hashCode();
    result = 37 * result + id;
    return result;
  }
  @Override public boolean equals(Object o) {
    return o instanceof CountedString &&
      Objects.equals(s, ((CountedString)o).s) &&
      Objects.equals(id, ((CountedString)o).id);
  }
  public static void main(String[] args) {
    Map<CountedString,Integer> map = new HashMap<>();
    CountedString[] cs = new CountedString[5];
    for(int i = 0; i < cs.length; i++) {
      cs[i] = new CountedString("hi");
      map.put(cs[i], i); // Autobox int to Integer
    }
    System.out.println(map);
    for(CountedString cstring : cs) {
      System.out.println("Looking up " + cstring);
      System.out.println(map.get(cstring));
    }
  }
}
/* Output:
{String: hi id: 4 hashCode(): 146450=3, String: hi id:
5 hashCode(): 146451=4, String: hi id: 2 hashCode():
146448=1, String: hi id: 3 hashCode(): 146449=2,
String: hi id: 1 hashCode(): 146447=0}
Looking up String: hi id: 1 hashCode(): 146447
0
Looking up String: hi id: 2 hashCode(): 146448
1
Looking up String: hi id: 3 hashCode(): 146449
2
```

```
Looking up String: hi id: 4 hashCode(): 146450
3
Looking up String: hi id: 5 hashCode(): 146451
4
*/
```

`CountedString` includes a `String` and an `id` that represents the number of `CountedString` objects containing an identical `String`. The counting is accomplished in the constructor by iterating through the `static ArrayList` where all the `Strings` are stored.

Both `hashCode()` and `equals()` produce results based on both fields; if they were just based on the `String` alone or the `id` alone, there would be duplicate matches for distinct values.

In `main()`, several `CountedString` objects are created using the same `String`, to show that the duplicates create unique values because of the count `id`. The `HashMap` is displayed so you see how it is stored internally (no discernible orders). Each key is looked up individually to demonstrate that the lookup mechanism is working properly.

As a second example, consider the `Individual` class that was used as the base class for the `reflection.pet` library defined in the Reflection chapter. The `Individual` class was used in that chapter but the definition is delayed until this appendix so you can properly understand the implementation.

Here, instead of calculating the `hashCode()` by hand, we'll use the proper approach with `Objects.hash()`:

```
// reflection/pets/Individual.java
package reflection.pets;
import java.util.*;

public class
Individual implements Comparable<Individual> {
  private static long counter = 0;
  private final long id = counter++;
  private String name;
  public Individual(String name) { this.name = name; }
  // 'name' is optional:
  public Individual() {}
```

```
  @Override public String toString() {
    return getClass().getSimpleName() +
      (name == null ? "" : " " + name);
  }
  public long id() { return id; }
  @Override public boolean equals(Object o) {
    return o instanceof Individual &&
      Objects.equals(id, ((Individual)o).id);
  }
  @Override public int hashCode() {
    return Objects.hash(name, id);
  }
  @Override public int compareTo(Individual arg) {
    // Compare by class name first:
    String first = getClass().getSimpleName();
    String argFirst = arg.getClass().getSimpleName();
    int firstCompare = first.compareTo(argFirst);
    if(firstCompare != 0)
      return firstCompare;
    if(name != null && arg.name != null) {
      int secondCompare = name.compareTo(arg.name);
      if(secondCompare != 0)
        return secondCompare;
    }
    return (arg.id < id ? -1 : (arg.id == id ? 0 : 1));
  }
}
```

The compareTo() method has a hierarchy of comparisons, so it will produce a sequence that is sorted first by actual type, then by name if there is one, and finally falls back to creation order. Here's an example that shows how it works:

```
// equalshashcode/IndividualTest.java
import collections.MapOfList;
import reflection.pets.*;
import java.util.*;

public class IndividualTest {
  public static void main(String[] args) {
    Set<Individual> pets = new TreeSet<>();
    for(List<? extends Pet> lp :
        MapOfList.petPeople.values())
      for(Pet p : lp)
        pets.add(p);
    pets.forEach(System.out::println);
  }
}
/* Output:
Cat Elsie May
Cat Pinkola
Cat Shackleton
Cat Stanford
Cymric Molly
Dog Margrett
Mutt Spot
Pug Louie aka Louis Snorkelstein Dupree
Rat Fizzy
Rat Freckly
Rat Fuzzy
*/
```

All pets have names so they are sorted first by type, then by name within their type.

Tuning a HashMap

It's possible to hand-tune a HashMap to increase its performance for your particular application. To understand performance issues when tuning a HashMap, some terminology is necessary:

Capacity: The number of buckets in the table.

Initial capacity: The number of buckets when the table is created. HashMap and HashSet have constructors that allow you to specify the initial capacity.

Size: The number of entries currently in the table.

Load factor: Size/capacity. A load factor of 0 is an empty table, 0.5 is a half-full table, etc. A lightly loaded table has few collisions and so is optimal for insertions and lookups (but will slow down the process of traversing with an iterator). HashMap and HashSet have constructors that allow you to specify the load factor. When this load factor is reached, the collection automatically increases the capacity (the number of buckets) by roughly doubling it and will redistribute the existing objects into the new set of buckets (this is called *rehashing*).

The default load factor used by HashMap is 0.75 (it doesn't rehash until the table is three-fourths full). This seems a good trade-off between time and space costs. A higher load factor decreases the space required by the table but increases the lookup cost, important because lookup is what you do most of the time (including both get() and put()).

If you know that you'll be storing many entries in a HashMap, creating it with an appropriately large initial capacity will prevent the overhead of automatic rehashing.[197]

[197] In a private message, Joshua Bloch wrote: "… I believe that we erred by allowing implementation details (such as hash table size and load factor) into our APIs. The client should perhaps tell us the maximum expected size of a collection, and we should take it from there. Clients can easily do more harm than good by choosing values for these parameters. As an extreme example, consider Vector's capacityIncrement. No one should ever set this, and we shouldn't have provided it. If you set it to any nonzero value, the asymptotic cost of a sequence of appends goes from linear to quadratic. In other words, it destroys your performance. Over time, we're beginning to wise up about this sort of thing. If you look at IdentityHashMap, you'll see it has no low-level tuning parameters."

Appendix: Collection Topics

> This appendix is a selection of somewhat more advanced topics than were introduced in Collections.

Sample Data

Let's create some sample data to use with our collection examples. The following data associates color names with RGB values of HTML colors. Notice that each of the keys and values are unique:

```
// onjava/HTMLColors.java
// Sample data for collection examples
package onjava;
import java.util.*;
import java.util.stream.*;
import java.util.concurrent.*;

public class HTMLColors {
  public static final Object[][] ARRAY = {
    { 0xF0F8FF, "AliceBlue" },
    { 0xFAEBD7, "AntiqueWhite" },
    { 0x7FFFD4, "Aquamarine" },
    { 0xF0FFFF, "Azure" },
    { 0xF5F5DC, "Beige" },
    { 0xFFE4C4, "Bisque" },
    { 0x000000, "Black" },
    { 0xFFEBCD, "BlanchedAlmond" },
    { 0x0000FF, "Blue" },
    { 0x8A2BE2, "BlueViolet" },
    { 0xA52A2A, "Brown" },
    { 0xDEB887, "BurlyWood" },
    { 0x5F9EA0, "CadetBlue" },
    { 0x7FFF00, "Chartreuse" },
    { 0xD2691E, "Chocolate" },
```

```
    { 0xFF7F50, "Coral" },
    { 0x6495ED, "CornflowerBlue" },
    { 0xFFF8DC, "Cornsilk" },
    { 0xDC143C, "Crimson" },
    { 0x00FFFF, "Cyan" },
    { 0x00008B, "DarkBlue" },
    { 0x008B8B, "DarkCyan" },
    { 0xB8860B, "DarkGoldenRod" },
    { 0xA9A9A9, "DarkGray" },
    { 0x006400, "DarkGreen" },
    { 0xBDB76B, "DarkKhaki" },
    { 0x8B008B, "DarkMagenta" },
    { 0x556B2F, "DarkOliveGreen" },
    { 0xFF8C00, "DarkOrange" },
    { 0x9932CC, "DarkOrchid" },
    { 0x8B0000, "DarkRed" },
    { 0xE9967A, "DarkSalmon" },
    { 0x8FBC8F, "DarkSeaGreen" },
    { 0x483D8B, "DarkSlateBlue" },
    { 0x2F4F4F, "DarkSlateGray" },
    { 0x00CED1, "DarkTurquoise" },
    { 0x9400D3, "DarkViolet" },
    { 0xFF1493, "DeepPink" },
    { 0x00BFFF, "DeepSkyBlue" },
    { 0x696969, "DimGray" },
    { 0x1E90FF, "DodgerBlue" },
    { 0xB22222, "FireBrick" },
    { 0xFFFAF0, "FloralWhite" },
    { 0x228B22, "ForestGreen" },
    { 0xDCDCDC, "Gainsboro" },
    { 0xF8F8FF, "GhostWhite" },
    { 0xFFD700, "Gold" },
    { 0xDAA520, "GoldenRod" },
    { 0x808080, "Gray" },
    { 0x008000, "Green" },
    { 0xADFF2F, "GreenYellow" },
    { 0xF0FFF0, "HoneyDew" },
    { 0xFF69B4, "HotPink" },
    { 0xCD5C5C, "IndianRed" },
    { 0x4B0082, "Indigo" },
    { 0xFFFFF0, "Ivory" },
    { 0xF0E68C, "Khaki" },
```

```
    { 0xE6E6FA, "Lavender" },
    { 0xFFF0F5, "LavenderBlush" },
    { 0x7CFC00, "LawnGreen" },
    { 0xFFFACD, "LemonChiffon" },
    { 0xADD8E6, "LightBlue" },
    { 0xF08080, "LightCoral" },
    { 0xE0FFFF, "LightCyan" },
    { 0xFAFAD2, "LightGoldenRodYellow" },
    { 0xD3D3D3, "LightGray" },
    { 0x90EE90, "LightGreen" },
    { 0xFFB6C1, "LightPink" },
    { 0xFFA07A, "LightSalmon" },
    { 0x20B2AA, "LightSeaGreen" },
    { 0x87CEFA, "LightSkyBlue" },
    { 0x778899, "LightSlateGray" },
    { 0xB0C4DE, "LightSteelBlue" },
    { 0xFFFFE0, "LightYellow" },
    { 0x00FF00, "Lime" },
    { 0x32CD32, "LimeGreen" },
    { 0xFAF0E6, "Linen" },
    { 0xFF00FF, "Magenta" },
    { 0x800000, "Maroon" },
    { 0x66CDAA, "MediumAquaMarine" },
    { 0x0000CD, "MediumBlue" },
    { 0xBA55D3, "MediumOrchid" },
    { 0x9370DB, "MediumPurple" },
    { 0x3CB371, "MediumSeaGreen" },
    { 0x7B68EE, "MediumSlateBlue" },
    { 0x00FA9A, "MediumSpringGreen" },
    { 0x48D1CC, "MediumTurquoise" },
    { 0xC71585, "MediumVioletRed" },
    { 0x191970, "MidnightBlue" },
    { 0xF5FFFA, "MintCream" },
    { 0xFFE4E1, "MistyRose" },
    { 0xFFE4B5, "Moccasin" },
    { 0xFFDEAD, "NavajoWhite" },
    { 0x000080, "Navy" },
    { 0xFDF5E6, "OldLace" },
    { 0x808000, "Olive" },
    { 0x6B8E23, "OliveDrab" },
    { 0xFFA500, "Orange" },
    { 0xFF4500, "OrangeRed" },
```

```
    { 0xDA70D6, "Orchid" },
    { 0xEEE8AA, "PaleGoldenRod" },
    { 0x98FB98, "PaleGreen" },
    { 0xAFEEEE, "PaleTurquoise" },
    { 0xDB7093, "PaleVioletRed" },
    { 0xFFEFD5, "PapayaWhip" },
    { 0xFFDAB9, "PeachPuff" },
    { 0xCD853F, "Peru" },
    { 0xFFC0CB, "Pink" },
    { 0xDDA0DD, "Plum" },
    { 0xB0E0E6, "PowderBlue" },
    { 0x800080, "Purple" },
    { 0xFF0000, "Red" },
    { 0xBC8F8F, "RosyBrown" },
    { 0x4169E1, "RoyalBlue" },
    { 0x8B4513, "SaddleBrown" },
    { 0xFA8072, "Salmon" },
    { 0xF4A460, "SandyBrown" },
    { 0x2E8B57, "SeaGreen" },
    { 0xFFF5EE, "SeaShell" },
    { 0xA0522D, "Sienna" },
    { 0xC0C0C0, "Silver" },
    { 0x87CEEB, "SkyBlue" },
    { 0x6A5ACD, "SlateBlue" },
    { 0x708090, "SlateGray" },
    { 0xFFFAFA, "Snow" },
    { 0x00FF7F, "SpringGreen" },
    { 0x4682B4, "SteelBlue" },
    { 0xD2B48C, "Tan" },
    { 0x008080, "Teal" },
    { 0xD8BFD8, "Thistle" },
    { 0xFF6347, "Tomato" },
    { 0x40E0D0, "Turquoise" },
    { 0xEE82EE, "Violet" },
    { 0xF5DEB3, "Wheat" },
    { 0xFFFFFF, "White" },
    { 0xF5F5F5, "WhiteSmoke" },
    { 0xFFFF00, "Yellow" },
    { 0x9ACD32, "YellowGreen" },
  };
  public static final Map<Integer,String> MAP =
    Arrays.stream(ARRAY)
```

```
      .collect(Collectors.toMap(
        element -> (Integer)element[0],
        element -> (String)element[1],
        (v1, v2) -> { // Merge function
          throw new IllegalStateException();
        },
        LinkedHashMap::new
      ));
  // Inversion only works if values are unique:
  public static <V, K> Map<V, K>
  invert(Map<K, V> map) {
    return map.entrySet().stream()
      .collect(Collectors.toMap(
        Map.Entry::getValue,
        Map.Entry::getKey,
        (v1, v2) -> {
          throw new IllegalStateException();
        },
        LinkedHashMap::new
      ));
  }
  public static final Map<String,Integer>
    INVMAP = invert(MAP);
  // Look up RGB value given a name:
  public static Integer rgb(String colorName) {
    return INVMAP.get(colorName);
  }
  public static final List<String> LIST =
    Arrays.stream(ARRAY)
      .map(item -> (String)item[1])
      .collect(Collectors.toList());
  public static final List<Integer> RGBLIST =
    Arrays.stream(ARRAY)
      .map(item -> (Integer)item[0])
      .collect(Collectors.toList());
  public static
  void show(Map.Entry<Integer,String> e) {
    System.out.format(
      "0x%06X: %s%n", e.getKey(), e.getValue());
  }
  public static void
  show(Map<Integer,String> m, int count) {
```

```
    m.entrySet().stream()
      .limit(count)
      .forEach(e -> show(e));
  }
  public static void show(Map<Integer,String> m) {
    show(m, m.size());
  }
  public static
  void show(Collection<String> lst, int count) {
    lst.stream()
      .limit(count)
      .forEach(System.out::println);
  }
  public static void show(Collection<String> lst) {
    show(lst, lst.size());
  }
  public static
  void showrgb(Collection<Integer> lst, int count) {
    lst.stream()
      .limit(count)
      .forEach(n -> System.out.format("0x%06X%n", n));
  }
  public static void showrgb(Collection<Integer> lst) {
    showrgb(lst, lst.size());
  }
  public static
  void showInv(Map<String,Integer> m, int count) {
    m.entrySet().stream()
      .limit(count)
      .forEach(e ->
        System.out.format(
          "%-20s  0x%06X%n", e.getKey(), e.getValue()));
  }
  public static void showInv(Map<String,Integer> m) {
    showInv(m, m.size());
  }
  public static void border() {
    System.out.println(
      "*******************************");
  }
}
```

`MAP` is created using Streams. The two-dimensional `ARRAY` is streamed into a `Map`, but notice we are not just using the simple version of `Collectors.toMap()`. That version produces a `HashMap`, which scrambles the order of the keys using a hashing function. To preserve the order, we place the key-value pairs directly into a `TreeMap`, which means we use the more complex version of `Collectors.toMap()`. This takes two functions that extract the key and value from each streamed element, just like the simple `Collectors.toMap()`. Then it requires a *merge function*, which resolves collisions between two values associated with the same key. Our data is pre-vetted so this should never happen and we throw an exception if it does. Finally we pass the function that produces an empty map of our desired type, which is then filled by the stream.

The `rgb()` method is a convenience function that takes a color name `String` and produces its numerical RGB value. To achieve this, we need an *inverted* version of `COLORS` which takes a `String` key and looks up an RGB `Integer` value. This is achieved through the `invert()` method, which throws an exception if any of the `COLORS` values are not unique.

We also create `LIST`, containing all the names, and `RGBLIST`, containing the RGB values in hex notation.

The first `show()` method takes a single `Map.Entry` and displays the key in hex notation to enable easy double-checking against the original `ARRAY`. Each of the methods with names that start with `show` are overloaded with one version taking a `count` argument to indicate how many elements you want to display, and the second version displaying all the elements in the sequence.

Here's a basic test:

```
// collectiontopics/HTMLColorTest.java
import static onjava.HTMLColors.*;

public class HTMLColorTest {
  static final int DISPLAY_SIZE = 20;
  public static void main(String[] args) {
    show(MAP, DISPLAY_SIZE);
    border();
    showInv(INVMAP, DISPLAY_SIZE);
    border();
    show(LIST, DISPLAY_SIZE);
```

```
    border();
    showrgb(RGBLIST, DISPLAY_SIZE);
  }
}
/* Output:
0xF0F8FF: AliceBlue
0xFAEBD7: AntiqueWhite
0x7FFFD4: Aquamarine
0xF0FFFF: Azure
0xF5F5DC: Beige
0xFFE4C4: Bisque
0x000000: Black
0xFFEBCD: BlanchedAlmond
0x0000FF: Blue
0x8A2BE2: BlueViolet
0xA52A2A: Brown
0xDEB887: BurlyWood
0x5F9EA0: CadetBlue
0x7FFF00: Chartreuse
0xD2691E: Chocolate
0xFF7F50: Coral
0x6495ED: CornflowerBlue
0xFFF8DC: Cornsilk
0xDC143C: Crimson
0x00FFFF: Cyan
*****************************
AliceBlue             0xF0F8FF
AntiqueWhite          0xFAEBD7
Aquamarine            0x7FFFD4
Azure                 0xF0FFFF
Beige                 0xF5F5DC
Bisque                0xFFE4C4
Black                 0x000000
BlanchedAlmond        0xFFEBCD
Blue                  0x0000FF
BlueViolet            0x8A2BE2
Brown                 0xA52A2A
BurlyWood             0xDEB887
CadetBlue             0x5F9EA0
Chartreuse            0x7FFF00
Chocolate             0xD2691E
Coral                 0xFF7F50
```

```
CornflowerBlue          0x6495ED
Cornsilk                0xFFF8DC
Crimson                 0xDC143C
Cyan                    0x00FFFF
*****************************
AliceBlue
AntiqueWhite
Aquamarine
Azure
Beige
Bisque
Black
BlanchedAlmond
Blue
BlueViolet
Brown
BurlyWood
CadetBlue
Chartreuse
Chocolate
Coral
CornflowerBlue
Cornsilk
Crimson
Cyan
*****************************
0xF0F8FF
0xFAEBD7
0x7FFFD4
0xF0FFFF
0xF5F5DC
0xFFE4C4
0x000000
0xFFEBCD
0x0000FF
0x8A2BE2
0xA52A2A
0xDEB887
0x5F9EA0
0x7FFF00
0xD2691E
0xFF7F50
```

```
0x6495ED
0xFFF8DC
0xDC143C
0x00FFFF
*/
```

Using a `LinkedHashMap`, we are indeed able to preserve the order of `HTMLColors.ARRAY`.

`List` Behavior

`Lists` are the most fundamental way to store and retrieve objects (after arrays). Basic List operations include:

- `add()` to insert elements
- `get()` for random-access selection of elements
- `iterator()` to get an `Iterator` over the sequence
- `stream()` to produce a `Stream` of elements

List constructors always preserve the order the elements are added.

The methods in the following example each cover a different group of activities: things that every `List` can do (`basicTest()`), moving around with an `Iterator` (`iterMotion()`) versus changing things with an `Iterator` (`iterManipulation()`), seeing the effects of `List` manipulation (`testVisual()`), and operations available only to `LinkedLists`:

```
// collectiontopics/ListOps.java
// Things you can do with Lists
import java.util.*;
import onjava.HTMLColors;

public class ListOps {
  // Create a short list for testing:
  static final List<String> LIST =
    HTMLColors.LIST.subList(0, 10);
  private static boolean b;
  private static String s;
  private static int i;
```

```
  private static Iterator<String> it;
  private static ListIterator<String> lit;
  public static void basicTest(List<String> a) {
    a.add(1, "x"); // Add at location 1
    a.add("x"); // Add at end
    // Add a collection:
    a.addAll(LIST);
    // Add a collection starting at location 3:
    a.addAll(3, LIST);
    b = a.contains("1"); // Is it in there?
    // Is the entire collection in there?
    b = a.containsAll(LIST);
    // Lists allow random access, which is cheap
    // for ArrayList, expensive for LinkedList:
    s = a.get(1); // Get (typed) object at location 1
    i = a.indexOf("1"); // Tell index of object
    b = a.isEmpty(); // Any elements inside?
    it = a.iterator(); // Ordinary Iterator
    lit = a.listIterator(); // ListIterator
    lit = a.listIterator(3); // Start at location 3
    i = a.lastIndexOf("1"); // Last match
    a.remove(1); // Remove location 1
    a.remove("3"); // Remove this object
    a.set(1, "y"); // Set location 1 to "y"
    // Keep everything that's in the argument
    // (the intersection of the two sets):
    a.retainAll(LIST);
    // Remove everything that's in the argument:
    a.removeAll(LIST);
    i = a.size(); // How big is it?
    a.clear(); // Remove all elements
  }
  public static void iterMotion(List<String> a) {
    ListIterator<String> it = a.listIterator();
    b = it.hasNext();
    b = it.hasPrevious();
    s = it.next();
    i = it.nextIndex();
    s = it.previous();
    i = it.previousIndex();
  }
  public static void iterManipulation(List<String> a) {
```

```
    ListIterator<String> it = a.listIterator();
    it.add("47");
    // Must move to an element after add():
    it.next();
    // Remove the element after the new one:
    it.remove();
    // Must move to an element after remove():
    it.next();
    // Change the element after the deleted one:
    it.set("47");
  }
  public static void testVisual(List<String> a) {
    System.out.println(a);
    List<String> b = LIST;
    System.out.println("b = " + b);
    a.addAll(b);
    a.addAll(b);
    System.out.println(a);
    // Insert, remove, and replace elements
    // using a ListIterator:
    ListIterator<String> x =
      a.listIterator(a.size()/2);
    x.add("one");
    System.out.println(a);
    System.out.println(x.next());
    x.remove();
    System.out.println(x.next());
    x.set("47");
    System.out.println(a);
    // Traverse the list backwards:
    x = a.listIterator(a.size());
    while(x.hasPrevious())
      System.out.print(x.previous() + " ");
    System.out.println();
    System.out.println("testVisual finished");
  }
  // There are some things that only LinkedLists can do:
  public static void testLinkedList() {
    LinkedList<String> ll = new LinkedList<>();
    ll.addAll(LIST);
    System.out.println(ll);
    // Treat it like a stack, pushing:
```

```
    ll.addFirst("one");
    ll.addFirst("two");
    System.out.println(ll);
    // Like "peeking" at the top of a stack:
    System.out.println(ll.getFirst());
    // Like popping a stack:
    System.out.println(ll.removeFirst());
    System.out.println(ll.removeFirst());
    // Treat it like a queue, pulling elements
    // off the tail end:
    System.out.println(ll.removeLast());
    System.out.println(ll);
  }
  public static void main(String[] args) {
    // Make and fill a new list each time:
    basicTest(new LinkedList<>(LIST));
    basicTest(new ArrayList<>(LIST));
    iterMotion(new LinkedList<>(LIST));
    iterMotion(new ArrayList<>(LIST));
    iterManipulation(new LinkedList<>(LIST));
    iterManipulation(new ArrayList<>(LIST));
    testVisual(new LinkedList<>(LIST));
    testLinkedList();
  }
}
/* Output:
[AliceBlue, AntiqueWhite, Aquamarine, Azure, Beige,
Bisque, Black, BlanchedAlmond, Blue, BlueViolet]
b = [AliceBlue, AntiqueWhite, Aquamarine, Azure, Beige,
Bisque, Black, BlanchedAlmond, Blue, BlueViolet]
[AliceBlue, AntiqueWhite, Aquamarine, Azure, Beige,
Bisque, Black, BlanchedAlmond, Blue, BlueViolet,
AliceBlue, AntiqueWhite, Aquamarine, Azure, Beige,
Bisque, Black, BlanchedAlmond, Blue, BlueViolet,
AliceBlue, AntiqueWhite, Aquamarine, Azure, Beige,
Bisque, Black, BlanchedAlmond, Blue, BlueViolet]
[AliceBlue, AntiqueWhite, Aquamarine, Azure, Beige,
Bisque, Black, BlanchedAlmond, Blue, BlueViolet,
AliceBlue, AntiqueWhite, Aquamarine, Azure, Beige, one,
Bisque, Black, BlanchedAlmond, Blue, BlueViolet,
AliceBlue, AntiqueWhite, Aquamarine, Azure, Beige,
Bisque, Black, BlanchedAlmond, Blue, BlueViolet]
```

```
Bisque
Black
[AliceBlue, AntiqueWhite, Aquamarine, Azure, Beige,
Bisque, Black, BlanchedAlmond, Blue, BlueViolet,
AliceBlue, AntiqueWhite, Aquamarine, Azure, Beige, one,
47, BlanchedAlmond, Blue, BlueViolet, AliceBlue,
AntiqueWhite, Aquamarine, Azure, Beige, Bisque, Black,
BlanchedAlmond, Blue, BlueViolet]
BlueViolet Blue BlanchedAlmond Black Bisque Beige Azure
Aquamarine AntiqueWhite AliceBlue BlueViolet Blue
BlanchedAlmond 47 one Beige Azure Aquamarine
AntiqueWhite AliceBlue BlueViolet Blue BlanchedAlmond
Black Bisque Beige Azure Aquamarine AntiqueWhite
AliceBlue
testVisual finished
[AliceBlue, AntiqueWhite, Aquamarine, Azure, Beige,
Bisque, Black, BlanchedAlmond, Blue, BlueViolet]
[two, one, AliceBlue, AntiqueWhite, Aquamarine, Azure,
Beige, Bisque, Black, BlanchedAlmond, Blue, BlueViolet]
two
two
one
BlueViolet
[AliceBlue, AntiqueWhite, Aquamarine, Azure, Beige,
Bisque, Black, BlanchedAlmond, Blue]
*/
```

In `basicTest()` and `iterMotion()` the calls are made to show the proper syntax, and although the return value is captured, it is not used. In some cases, the return value isn't captured at all. Look up the full usage of each of these methods in the JDK documentation before you use them.

`Set` Behavior

The point of a `Set` is to test for membership, although you can also use it as a tool to remove duplicate elements. If you don't care about element order or concurrency, `HashSet` is always your best bet because it is designed for the fastest possible lookup (using the *hashing function* we explored in the Appendix: Understanding `equals()` and `hashCode()`).

Additional `Set` implementations produce different ordering behavior:

```
// collectiontopics/SetOrder.java
import java.util.*;
import onjava.HTMLColors;

public class SetOrder {
  static String[] sets = {
    "java.util.HashSet",
    "java.util.TreeSet",
    "java.util.concurrent.ConcurrentSkipListSet",
    "java.util.LinkedHashSet",
    "java.util.concurrent.CopyOnWriteArraySet",
  };
  static final List<String> RLIST =
    new ArrayList<>(HTMLColors.LIST);
  static {
    Collections.reverse(RLIST);
  }
  public static void
  main(String[] args) throws Exception {
    for(String type: sets) {
      System.out.format("[-> %s <-]%n",
        type.substring(type.lastIndexOf('.') + 1));
      @SuppressWarnings("unchecked")
      Set<String> set = (Set<String>)
        Class.forName(type).getConstructor().newInstance();
      set.addAll(RLIST);
      set.stream()
        .limit(10)
        .forEach(System.out::println);
    }
  }
}
/* Output:
[-> HashSet <-]
MediumOrchid
PaleGoldenRod
Sienna
LightSlateGray
DarkSeaGreen
Black
```

```
Gainsboro
Orange
LightCoral
DodgerBlue
[-> TreeSet <-]
AliceBlue
AntiqueWhite
Aquamarine
Azure
Beige
Bisque
Black
BlanchedAlmond
Blue
BlueViolet
[-> ConcurrentSkipListSet <-]
AliceBlue
AntiqueWhite
Aquamarine
Azure
Beige
Bisque
Black
BlanchedAlmond
Blue
BlueViolet
[-> LinkedHashSet <-]
YellowGreen
Yellow
WhiteSmoke
White
Wheat
Violet
Turquoise
Tomato
Thistle
Teal
[-> CopyOnWriteArraySet <-]
YellowGreen
Yellow
WhiteSmoke
White
```

```
Wheat
Violet
Turquoise
Tomato
Thistle
Teal
*/
```

We need `@SuppressWarnings("unchecked")` here because we pass a `String`, which could be anything, to `Class.forName(type).newInstance()`. The compiler can't guarantee this is a successful operation.

`RLIST` is a reversed version of `HTMLColors.LIST`. Because `Collections.reverse()` performs the reverse by modifying the argument (instead of returning a new `List` containing reversed elements), that call is performed inside a `static` clause. `RLIST` prevents us from accidentally thinking that a `Set` sorts its results.

The `HashSet` output appears to have no discernible order (because it's based on the hash function). Both `TreeSet` and `ConcurrentSkipListSet` have sorted their elements, and they implement the `SortedSet` interface to indicate this; that interface also has further operations available because the `Set` is in sorted order. `LinkedHashSet` and `CopyOnWriteArraySet` preserve the order of the elements as they are added, although there is no interface to indicate this.

`ConcurrentSkipListSet` and `CopyOnWriteArraySet` are thread safe.

At the end of the appendix we'll find out the performance cost of the additional ordering imposed on non-`HashSet` implementations, along with the cost of any other functionality in the different implementations.

Using Functional Operations with any `Map`

Just as with the `Collection` interface, `forEach()` is built into the `Map` interface. But what if you want to perform any of the other basic functional operations: `map()`, `flatMap()`, `reduce()` or `filter()`? When you look at the `Map` interface, there's no hint of these.

You connect to these methods through `entrySet()`, which produces a `Set` of `Map.Entry` objects. This `Set`, in turn, contains `stream()` and `parallelStream()`

methods. The only thing you must remember is that you're working with `Map.Entry` objects:

```
// collectiontopics/FunctionalMap.java
// Functional operations on a Map
import java.util.*;
import java.util.stream.*;
import java.util.concurrent.*;
import static onjava.HTMLColors.*;

public class FunctionalMap {
  public static void main(String[] args) {
    MAP.entrySet().stream()
      .map(Map.Entry::getValue)
      .filter(v -> v.startsWith("Dark"))
      .map(v -> v.replaceFirst("Dark", "Hot"))
      .forEach(System.out::println);
  }
}
/* Output:
HotBlue
HotCyan
HotGoldenRod
HotGray
HotGreen
HotKhaki
HotMagenta
HotOliveGreen
HotOrange
HotOrchid
HotRed
HotSalmon
HotSeaGreen
HotSlateBlue
HotSlateGray
HotTurquoise
HotViolet
*/
```

Once you produce a `Stream`, all the basic functional methods (and more) become available.

Selecting Parts of a `Map`

The `NavigableMap` interface, implemented by `TreeMap` and `ConcurrentSkipListMap`, solves problems that require selecting portions of a `Map`. Here's an example utilizing `HTMLColors`:

```
// collectiontopics/NavMap.java
// NavigableMap produces pieces of a Map
import java.util.*;
import java.util.concurrent.*;
import static onjava.HTMLColors.*;

public class NavMap {
  public static final
  NavigableMap<Integer,String> COLORS =
    new ConcurrentSkipListMap<>(MAP);
  public static void main(String[] args) {
    show(COLORS.firstEntry());
    border();
    show(COLORS.lastEntry());
    border();
    NavigableMap<Integer, String> toLime =
      COLORS.headMap(rgb("Lime"), true);
    show(toLime);
    border();
    show(COLORS.ceilingEntry(rgb("DeepSkyBlue") - 1));
    border();
    show(COLORS.floorEntry(rgb("DeepSkyBlue") - 1));
    border();
    show(toLime.descendingMap());
    border();
    show(COLORS.tailMap(rgb("MistyRose"), true));
    border();
    show(COLORS.subMap(
      rgb("Orchid"), true,
      rgb("DarkSalmon"), false));
  }
}
/* Output:
0x000000: Black
*****************************
```

```
0xFFFFFF: White
*****************************
0x000000: Black
0x000080: Navy
0x00008B: DarkBlue
0x0000CD: MediumBlue
0x0000FF: Blue
0x006400: DarkGreen
0x008000: Green
0x008080: Teal
0x008B8B: DarkCyan
0x00BFFF: DeepSkyBlue
0x00CED1: DarkTurquoise
0x00FA9A: MediumSpringGreen
0x00FF00: Lime
*****************************
0x00BFFF: DeepSkyBlue
*****************************
0x008B8B: DarkCyan
*****************************
0x00FF00: Lime
0x00FA9A: MediumSpringGreen
0x00CED1: DarkTurquoise
0x00BFFF: DeepSkyBlue
0x008B8B: DarkCyan
0x008080: Teal
0x008000: Green
0x006400: DarkGreen
0x0000FF: Blue
0x0000CD: MediumBlue
0x00008B: DarkBlue
0x000080: Navy
0x000000: Black
*****************************
0xFFE4E1: MistyRose
0xFFEBCD: BlanchedAlmond
0xFFEFD5: PapayaWhip
0xFFF0F5: LavenderBlush
0xFFF5EE: SeaShell
0xFFF8DC: Cornsilk
0xFFFACD: LemonChiffon
0xFFFAF0: FloralWhite
```

```
0xFFFAFA: Snow
0xFFFF00: Yellow
0xFFFFE0: LightYellow
0xFFFFF0: Ivory
0xFFFFFF: White
***************************
0xDA70D6: Orchid
0xDAA520: GoldenRod
0xDB7093: PaleVioletRed
0xDC143C: Crimson
0xDCDCDC: Gainsboro
0xDDA0DD: Plum
0xDEB887: BurlyWood
0xE0FFFF: LightCyan
0xE6E6FA: Lavender
*/
```

In `main()`, you see various features of a `NavigableMap`. Because a `NavigableMap` has a key order, it enables the concept of a `firstEntry()` and `lastEntry()`. Calling `headMap()` produces a `NavigableMap` containing the elements starting from the beginning of the `Map` up to the argument to `headMap()`, with the `boolean` value indicating whether to include the argument. Calling `tailMap()` performs a similar operation, but at the end of the `Map`. `subMap()` allows you to produce a section of the `Map` from the middle.

`ceilingEntry()` searches upward from the key value to the next entry, and `floorEntry()` searches downward. `descendingMap()` reverses the order of the `NavigableMap`.

If the problem you're solving is simplified by slicing up a `Map`, `NavigableMap` does the trick. Other collection implementations have similar features that can also help your problem-solving process.

Filling Collections

Just as with `Arrays`, there is a companion class called `Collections` containing `static` utility methods, including one called `fill()`. `fill()` just duplicates a single object reference throughout the collection. In addition, it only works for `List` objects, but the resulting list can be passed to a constructor or to an `addAll()` method:

```
// collectiontopics/FillingLists.java
// Collections.fill() & Collections.nCopies()
import java.util.*;

class StringAddress {
  private String s;
  StringAddress(String s) { this.s = s; }
  @Override public String toString() {
    return super.toString() + " " + s;
  }
}

public class FillingLists {
  public static void main(String[] args) {
    List<StringAddress> list = new ArrayList<>(
      Collections.nCopies(4,
        new StringAddress("Hello")));
    System.out.println(list);
    Collections.fill(list,
      new StringAddress("World!"));
    System.out.println(list);
  }
}
/* Output:
[StringAddress@19e0bfd Hello, StringAddress@19e0bfd
Hello, StringAddress@19e0bfd Hello,
StringAddress@19e0bfd Hello]
[StringAddress@139a55 World!, StringAddress@139a55
World!, StringAddress@139a55 World!,
StringAddress@139a55 World!]
*/
```

This example shows two ways to fill a `Collection` with references to a single object. The first, `Collections.nCopies()`, creates a `List` which is passed to the constructor; this fills the `ArrayList`.

The `toString()` method in `StringAddress` calls `Object.toString()`, which produces the class name followed by the unsigned hexadecimal representation of the hash code of the object (generated by the `hashCode()` method). The output shows that all references are set to the same object, and this is also true after calling the second method, `Collections.fill()`. The `fill()` method is made even less useful

by the fact it can only replace elements already in the `List`, and will not add new elements.

Using `Suppliers` to Fill a `Collection`

The `onjava.Suppliers` class introduced in the Generics chapter provides a universal solution for filling `Collections`. Here's an example that initializes several different types of `Collection` using `Suppliers`:

```
// collectiontopics/SuppliersCollectionTest.java
import java.util.*;
import java.util.function.*;
import java.util.stream.*;
import onjava.*;

class Government implements Supplier<String> {
  static String[] foundation = (
    "strange women lying in ponds " +
    "distributing swords is no basis " +
    "for a system of government").split(" ");
  private int index;
  @Override public String get() {
    return foundation[index++];
  }
}

public class SuppliersCollectionTest {
  public static void main(String[] args) {
    // Suppliers class from the Generics chapter:
    Set<String> set = Suppliers.create(
      LinkedHashSet::new, new Government(), 15);
    System.out.println(set);
    List<String> list = Suppliers.create(
      LinkedList::new, new Government(), 15);
    System.out.println(list);
    list = new ArrayList<>();
    Suppliers.fill(list, new Government(), 15);
    System.out.println(list);

    // Or we can use Streams:
    set = Arrays.stream(Government.foundation)
```

```
      .collect(Collectors.toSet());
    System.out.println(set);
    list = Arrays.stream(Government.foundation)
      .collect(Collectors.toList());
    System.out.println(list);
    list = Arrays.stream(Government.foundation)
      .collect(Collectors
        .toCollection(LinkedList::new));
    System.out.println(list);
    set = Arrays.stream(Government.foundation)
      .collect(Collectors
        .toCollection(LinkedHashSet::new));
    System.out.println(set);
  }
}
/* Output:
[strange, women, lying, in, ponds, distributing,
swords, is, no, basis, for, a, system, of, government]
[strange, women, lying, in, ponds, distributing,
swords, is, no, basis, for, a, system, of, government]
[strange, women, lying, in, ponds, distributing,
swords, is, no, basis, for, a, system, of, government]
[ponds, no, a, in, swords, for, is, basis, strange,
system, government, distributing, of, women, lying]
[strange, women, lying, in, ponds, distributing,
swords, is, no, basis, for, a, system, of, government]
[strange, women, lying, in, ponds, distributing,
swords, is, no, basis, for, a, system, of, government]
[strange, women, lying, in, ponds, distributing,
swords, is, no, basis, for, a, system, of, government]
*/
```

The elements of the `LinkedHashSet` are in insertion order because it maintains a linked list to hold that order.

Notice, however, the second part of the example: much of the time you can just use `Streams` to create and fill a `Collection`. And here, the `Stream` version doesn't require us to state the number of elements we want created as the `Supplier` version does; it just absorbs all the `Stream` elements.

The `Stream` solution is preferred whenever possible.

`Map` Suppliers

Populating a `Map` with data using a `Supplier` requires a `Pair` class because a pair of objects (one key and one value) must be produced by each call to a `Supplier`'s `get()`:

```
// onjava/Pair.java
package onjava;

public class Pair<K, V> {
  public final K key;
  public final V value;
  public Pair(K k, V v) {
    key = k;
    value = v;
  }
  public K key() { return key; }
  public V value() { return value; }
  public static <K,V> Pair<K, V> make(K k, V v) {
    return new Pair<K,V>(k, v);
  }
}
```

`Pair` is a read-only *Data Transfer Object* or *Messenger*. This is basically the same as `Tuple2` from the Generics chapter, but the name is more appropriate for `Map` initialization. I've also added the `static make()` method to provide a simpler shorthand for creating `Pair` objects.

Java 8 `Streams` provide convenient ways to produce filled `Maps`:

```
// collectiontopics/StreamFillMaps.java
import java.util.*;
import java.util.function.*;
import java.util.stream.*;
import onjava.*;

class Letters
implements Supplier<Pair<Integer,String>> {
  private int number = 1;
  private char letter = 'A';
  @Override public Pair<Integer,String> get() {
```

```
    return new Pair<>(number++, "" + letter++);
  }
}

public class StreamFillMaps {
  public static void main(String[] args) {
    Map<Integer,String> m =
      Stream.generate(new Letters())
      .limit(11)
      .collect(Collectors
        .toMap(Pair::key, Pair::value));
    System.out.println(m);

    // Two separate Suppliers:
    Rand.String rs = new Rand.String(3);
    Count.Character cc = new Count.Character();
    Map<Character,String> mcs = Stream.generate(
      () -> Pair.make(cc.get(), rs.get()))
      .limit(8)
      .collect(Collectors
        .toMap(Pair::key, Pair::value));
    System.out.println(mcs);

    // A key Supplier and a single value:
    Map<Character,String> mcs2 = Stream.generate(
      () -> Pair.make(cc.get(), "Val"))
      .limit(8)
      .collect(Collectors
        .toMap(Pair::key, Pair::value));
    System.out.println(mcs2);
  }
}
/* Output:
{1=A, 2=B, 3=C, 4=D, 5=E, 6=F, 7=G, 8=H, 9=I, 10=J,
11=K}
{b=btp, c=enp, d=ccu, e=xsz, f=gvg, g=mei, h=nne,
i=elo}
{p=Val, q=Val, j=Val, k=Val, l=Val, m=Val, n=Val,
o=Val}
*/
```

A pattern emerges in the above example, which we can use to create a tool that

automatically creates and fills `Maps`:

```
// onjava/FillMap.java
package onjava;
import java.util.*;
import java.util.function.*;
import java.util.stream.*;

public class FillMap {
  public static <K, V> Map<K,V>
  basic(Supplier<Pair<K,V>> pairGen, int size) {
    return Stream.generate(pairGen)
      .limit(size)
      .collect(Collectors
        .toMap(Pair::key, Pair::value));
  }
  public static <K, V> Map<K,V>
  basic(Supplier<K> keyGen,
        Supplier<V> valueGen, int size) {
    return Stream.generate(
      () -> Pair.make(keyGen.get(), valueGen.get()))
      .limit(size)
      .collect(Collectors
        .toMap(Pair::key, Pair::value));
  }
  public static <K, V, M extends Map<K,V>>
  M create(Supplier<K> keyGen,
           Supplier<V> valueGen,
           Supplier<M> mapSupplier, int size) {
    return Stream.generate( () ->
      Pair.make(keyGen.get(), valueGen.get()))
        .limit(size)
        .collect(Collectors
          .toMap(Pair::key, Pair::value,
                 (k, v) -> k, mapSupplier));
  }
}
```

The `basic()` method produces a default `Map`, while `create()` allows you to specify an exact type of map, and it returns that exact type.

Here's a test:

```
// collectiontopics/FillMapTest.java
import java.util.*;
import java.util.function.*;
import java.util.stream.*;
import onjava.*;

public class FillMapTest {
  public static void main(String[] args) {
    Map<String,Integer> mcs = FillMap.basic(
      new Rand.String(4), new Count.Integer(), 7);
    System.out.println(mcs);
    HashMap<String,Integer> hashm =
      FillMap.create(new Rand.String(4),
        new Count.Integer(), HashMap::new, 7);
    System.out.println(hashm);
    LinkedHashMap<String,Integer> linkm =
      FillMap.create(new Rand.String(4),
        new Count.Integer(), LinkedHashMap::new, 7);
    System.out.println(linkm);
  }
}
/* Output:
{npcc=1, ztdv=6, gvgm=3, btpe=0, einn=4, eelo=5,
uxsz=2}
{npcc=1, ztdv=6, gvgm=3, btpe=0, einn=4, eelo=5,
uxsz=2}
{btpe=0, npcc=1, uxsz=2, gvgm=3, einn=4, eelo=5,
ztdv=6}
*/
```

Custom `Collection` and `Map` using *Flyweight*

This section shows how to create custom `Collection` and `Map` implementations. Each `java.util` collection has its own `Abstract` class providing a partial implementation of that collection, so you need only implement the necessary methods to produce the desired collection. You'll see how relatively simple it is to create a custom `Map` and `Collection` by inheriting from the `java.util.Abstract` classes. For example, to create a read-only `Set`, you inherit from `AbstractSet` and implement `iterator()` and `size()`. The last example is an alternate way to produce test

data. The resulting collection is typically read-only, and the number of methods you provide is minimized.

This solution also demonstrates the *Flyweight* design pattern. You use a flyweight when the ordinary solution requires too many objects, or when producing normal objects takes up too much space. The Flyweight pattern externalizes part of the object. Instead of everything in the object being contained within the object, some or all of the object is looked up in a more efficient external table (or produced through some other calculation that saves space).

Here is a `List` that can be any size, and is (effectively) pre-initialized with `Integer` data. To create a read-only `List` from an `AbstractList`, you must implement `get()` and `size()`:

```
// onjava/CountingIntegerList.java
// List of any length, containing sample data
// {java onjava.CountingIntegerList}
package onjava;
import java.util.*;

public class CountingIntegerList
extends AbstractList<Integer> {
  private int size;
  public CountingIntegerList() { size = 0; }
  public CountingIntegerList(int size) {
    this.size = size < 0 ? 0 : size;
  }
  @Override public Integer get(int index) {
    return index;
  }
  @Override public int size() { return size; }
  public static void main(String[] args) {
    List<Integer> cil =
      new CountingIntegerList(30);
    System.out.println(cil);
    System.out.println(cil.get(500));
  }
}
/* Output:
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15,
16, 17, 18, 19, 20, 21, 22, 23, 24, 25, 26, 27, 28, 29]
```

```
500
*/
```

The value of `size` is only important if you want to "limit the length" of the `List`, as we do in `main()`. Even in that case, `get()` will produce any value.

This class is a clear example of flyweight. `get()` "calculates" the value when you ask for it, so there's no actual underlying `List` structure that requires storage and initialization.

The storage saved here would never make a difference in most programs. However, it allows you to call `List.get()` using a very large `index` without requiring a `List` populated with all those values. Also, you can use a very large number of `CountingIntegerLists` in your program without worrying about storage. Indeed, one of the benefits of flyweight is it allows you to use nicer abstractions without concern for resources.

We can use flyweight to implement other "initialized" custom collections with a data set of any size. Here is a `Map` that produces a unique value for every `Integer` key:

```
// onjava/CountMap.java
// Unlimited-length Map containing sample data
// {java onjava.CountMap}
package onjava;
import java.util.*;
import java.util.stream.*;

public class CountMap
extends AbstractMap<Integer,String> {
  private int size;
  private static char[] chars =
    "ABCDEFGHIJKLMNOPQRSTUVWXYZ".toCharArray();
  private static String value(int key) {
    return
      chars[key % chars.length] +
      Integer.toString(key / chars.length);
  }
  public CountMap(int size) {
    this.size = size < 0 ? 0 : size;
  }
  @Override public String get(Object key) {
    return value((Integer)key);
```

```
  }
  private static class Entry
  implements Map.Entry<Integer,String> {
    int index;
    Entry(int index) { this.index = index; }
    @Override   public boolean equals(Object o) {
      return o instanceof Entry &&
        Objects.equals(index, ((Entry)o).index);
    }
    @Override public Integer getKey() { return index; }
    @Override public String getValue() {
      return value(index);
    }
    @Override public String setValue(String value) {
      throw new UnsupportedOperationException();
    }
    @Override public int hashCode() {
      return Objects.hashCode(index);
    }
  }
  @Override
  public Set<Map.Entry<Integer,String>> entrySet() {
    // LinkedHashSet retains initialization order:
    return IntStream.range(0, size)
      .mapToObj(Entry::new)
      .collect(Collectors
        .toCollection(LinkedHashSet::new));
  }
  public static void main(String[] args) {
    final int size = 6;
    CountMap cm = new CountMap(60);
    System.out.println(cm);
    System.out.println(cm.get(500));
    cm.values().stream()
      .limit(size)
      .forEach(System.out::println);
    System.out.println();
    new Random(47).ints(size, 0, 1000)
      .mapToObj(cm::get)
      .forEach(System.out::println);
  }
}
```

```
/* Output:
{0=A0, 1=B0, 2=C0, 3=D0, 4=E0, 5=F0, 6=G0, 7=H0, 8=I0,
9=J0, 10=K0, 11=L0, 12=M0, 13=N0, 14=O0, 15=P0, 16=Q0,
17=R0, 18=S0, 19=T0, 20=U0, 21=V0, 22=W0, 23=X0, 24=Y0,
25=Z0, 26=A1, 27=B1, 28=C1, 29=D1, 30=E1, 31=F1, 32=G1,
33=H1, 34=I1, 35=J1, 36=K1, 37=L1, 38=M1, 39=N1, 40=O1,
41=P1, 42=Q1, 43=R1, 44=S1, 45=T1, 46=U1, 47=V1, 48=W1,
49=X1, 50=Y1, 51=Z1, 52=A2, 53=B2, 54=C2, 55=D2, 56=E2,
57=F2, 58=G2, 59=H2}
G19
A0
B0
C0
D0
E0
F0

Y9
J21
R26
D33
Z36
N16
*/
```

To create a read-only `Map`, you inherit from `AbstractMap` and implement `entrySet()`. The `private value()` method performs the calculation of the value for any key, and is used within `get()` and `Entry.getValue()`. The size of a `CountMap` is negligible.

A `LinkedHashSet` is used instead of creating a custom `Set` class, so the flyweight is not fully implemented. This object is only produced if you call `entrySet()`.

Now we'll create a more complex flyweight. The data set in this example is a `Map` of the countries of the world and their capitals. The `capitals()` method produces a `Map` of countries and capitals. The `names()` method produces a `List` of the country names. Both methods provide a partial listing when given an `int` argument indicating the desired size:

```
// onjava/Countries.java
// "Flyweight" Maps and Lists of sample data
// {java onjava.Countries}
package onjava;
import java.util.*;

public class Countries {
  public static final String[][] DATA = {
    // Africa
    {"ALGERIA","Algiers"},
    {"ANGOLA","Luanda"},
    {"BENIN","Porto-Novo"},
    {"BOTSWANA","Gaberone"},
    {"BURKINA FASO","Ouagadougou"},
    {"BURUNDI","Bujumbura"},
    {"CAMEROON","Yaounde"},
    {"CAPE VERDE","Praia"},
    {"CENTRAL AFRICAN REPUBLIC","Bangui"},
    {"CHAD","N'djamena"},
    {"COMOROS","Moroni"},
    {"CONGO","Brazzaville"},
    {"DJIBOUTI","Dijibouti"},
    {"EGYPT","Cairo"},
    {"EQUATORIAL GUINEA","Malabo"},
    {"ERITREA","Asmara"},
    {"ETHIOPIA","Addis Ababa"},
    {"GABON","Libreville"},
    {"THE GAMBIA","Banjul"},
    {"GHANA","Accra"},
    {"GUINEA","Conakry"},
    {"BISSAU","Bissau"},
    {"COTE D'IVOIR (IVORY COAST)","Yamoussoukro"},
    {"KENYA","Nairobi"},
    {"LESOTHO","Maseru"},
    {"LIBERIA","Monrovia"},
    {"LIBYA","Tripoli"},
    {"MADAGASCAR","Antananarivo"},
    {"MALAWI","Lilongwe"},
    {"MALI","Bamako"},
    {"MAURITANIA","Nouakchott"},
    {"MAURITIUS","Port Louis"},
    {"MOROCCO","Rabat"},
```

```
    {"MOZAMBIQUE","Maputo"},
    {"NAMIBIA","Windhoek"},
    {"NIGER","Niamey"},
    {"NIGERIA","Abuja"},
    {"RWANDA","Kigali"},
    {"SAO TOME E PRINCIPE","Sao Tome"},
    {"SENEGAL","Dakar"},
    {"SEYCHELLES","Victoria"},
    {"SIERRA LEONE","Freetown"},
    {"SOMALIA","Mogadishu"},
    {"SOUTH AFRICA","Pretoria/Cape Town"},
    {"SUDAN","Khartoum"},
    {"SWAZILAND","Mbabane"},
    {"TANZANIA","Dodoma"},
    {"TOGO","Lome"},
    {"TUNISIA","Tunis"},
    {"UGANDA","Kampala"},
    {"DEMOCRATIC REPUBLIC OF THE CONGO (ZAIRE)",
     "Kinshasa"},
    {"ZAMBIA","Lusaka"},
    {"ZIMBABWE","Harare"},
    // Asia
    {"AFGHANISTAN","Kabul"},
    {"BAHRAIN","Manama"},
    {"BANGLADESH","Dhaka"},
    {"BHUTAN","Thimphu"},
    {"BRUNEI","Bandar Seri Begawan"},
    {"CAMBODIA","Phnom Penh"},
    {"CHINA","Beijing"},
    {"CYPRUS","Nicosia"},
    {"INDIA","New Delhi"},
    {"INDONESIA","Jakarta"},
    {"IRAN","Tehran"},
    {"IRAQ","Baghdad"},
    {"ISRAEL","Jerusalem"},
    {"JAPAN","Tokyo"},
    {"JORDAN","Amman"},
    {"KUWAIT","Kuwait City"},
    {"LAOS","Vientiane"},
    {"LEBANON","Beirut"},
    {"MALAYSIA","Kuala Lumpur"},
    {"THE MALDIVES","Male"},
```

```
    {"MONGOLIA","Ulan Bator"},
    {"MYANMAR (BURMA)","Rangoon"},
    {"NEPAL","Katmandu"},
    {"NORTH KOREA","P'yongyang"},
    {"OMAN","Muscat"},
    {"PAKISTAN","Islamabad"},
    {"PHILIPPINES","Manila"},
    {"QATAR","Doha"},
    {"SAUDI ARABIA","Riyadh"},
    {"SINGAPORE","Singapore"},
    {"SOUTH KOREA","Seoul"},
    {"SRI LANKA","Colombo"},
    {"SYRIA","Damascus"},
    {"TAIWAN (REPUBLIC OF CHINA)","Taipei"},
    {"THAILAND","Bangkok"},
    {"TURKEY","Ankara"},
    {"UNITED ARAB EMIRATES","Abu Dhabi"},
    {"VIETNAM","Hanoi"},
    {"YEMEN","Sana'a"},
    // Australia and Oceania
    {"AUSTRALIA","Canberra"},
    {"FIJI","Suva"},
    {"KIRIBATI","Bairiki"},
    {"MARSHALL ISLANDS","Dalap-Uliga-Darrit"},
    {"MICRONESIA","Palikir"},
    {"NAURU","Yaren"},
    {"NEW ZEALAND","Wellington"},
    {"PALAU","Koror"},
    {"PAPUA NEW GUINEA","Port Moresby"},
    {"SOLOMON ISLANDS","Honaira"},
    {"TONGA","Nuku'alofa"},
    {"TUVALU","Fongafale"},
    {"VANUATU","Port Vila"},
    {"WESTERN SAMOA","Apia"},
    // Eastern Europe and former USSR
    {"ARMENIA","Yerevan"},
    {"AZERBAIJAN","Baku"},
    {"BELARUS (BYELORUSSIA)","Minsk"},
    {"BULGARIA","Sofia"},
    {"GEORGIA","Tbilisi"},
    {"KAZAKSTAN","Almaty"},
    {"KYRGYZSTAN","Alma-Ata"},
```

```
    {"MOLDOVA","Chisinau"},
    {"RUSSIA","Moscow"},
    {"TAJIKISTAN","Dushanbe"},
    {"TURKMENISTAN","Ashkhabad"},
    {"UKRAINE","Kyiv"},
    {"UZBEKISTAN","Tashkent"},
    // Europe
    {"ALBANIA","Tirana"},
    {"ANDORRA","Andorra la Vella"},
    {"AUSTRIA","Vienna"},
    {"BELGIUM","Brussels"},
    {"BOSNIA-HERZEGOVINA","Sarajevo"},
    {"CROATIA","Zagreb"},
    {"CZECH REPUBLIC","Prague"},
    {"DENMARK","Copenhagen"},
    {"ESTONIA","Tallinn"},
    {"FINLAND","Helsinki"},
    {"FRANCE","Paris"},
    {"GERMANY","Berlin"},
    {"GREECE","Athens"},
    {"HUNGARY","Budapest"},
    {"ICELAND","Reykjavik"},
    {"IRELAND","Dublin"},
    {"ITALY","Rome"},
    {"LATVIA","Riga"},
    {"LIECHTENSTEIN","Vaduz"},
    {"LITHUANIA","Vilnius"},
    {"LUXEMBOURG","Luxembourg"},
    {"MACEDONIA","Skopje"},
    {"MALTA","Valletta"},
    {"MONACO","Monaco"},
    {"MONTENEGRO","Podgorica"},
    {"THE NETHERLANDS","Amsterdam"},
    {"NORWAY","Oslo"},
    {"POLAND","Warsaw"},
    {"PORTUGAL","Lisbon"},
    {"ROMANIA","Bucharest"},
    {"SAN MARINO","San Marino"},
    {"SERBIA","Belgrade"},
    {"SLOVAKIA","Bratislava"},
    {"SLOVENIA","Ljuijana"},
    {"SPAIN","Madrid"},
```

```
    {"SWEDEN","Stockholm"},
    {"SWITZERLAND","Berne"},
    {"UNITED KINGDOM","London"},
    {"VATICAN CITY","Vatican City"},
    // North and Central America
    {"ANTIGUA AND BARBUDA","Saint John's"},
    {"BAHAMAS","Nassau"},
    {"BARBADOS","Bridgetown"},
    {"BELIZE","Belmopan"},
    {"CANADA","Ottawa"},
    {"COSTA RICA","San Jose"},
    {"CUBA","Havana"},
    {"DOMINICA","Roseau"},
    {"DOMINICAN REPUBLIC","Santo Domingo"},
    {"EL SALVADOR","San Salvador"},
    {"GRENADA","Saint George's"},
    {"GUATEMALA","Guatemala City"},
    {"HAITI","Port-au-Prince"},
    {"HONDURAS","Tegucigalpa"},
    {"JAMAICA","Kingston"},
    {"MEXICO","Mexico City"},
    {"NICARAGUA","Managua"},
    {"PANAMA","Panama City"},
    {"ST. KITTS AND NEVIS","Basseterre"},
    {"ST. LUCIA","Castries"},
    {"ST. VINCENT AND THE GRENADINES","Kingstown"},
    {"UNITED STATES OF AMERICA","Washington, D.C."},
    // South America
    {"ARGENTINA","Buenos Aires"},
    {"BOLIVIA","Sucre (legal)/La Paz(administrative)"},
    {"BRAZIL","Brasilia"},
    {"CHILE","Santiago"},
    {"COLOMBIA","Bogota"},
    {"ECUADOR","Quito"},
    {"GUYANA","Georgetown"},
    {"PARAGUAY","Asuncion"},
    {"PERU","Lima"},
    {"SURINAME","Paramaribo"},
    {"TRINIDAD AND TOBAGO","Port of Spain"},
    {"URUGUAY","Montevideo"},
    {"VENEZUELA","Caracas"},
  };
```

```
// Use AbstractMap by implementing entrySet()
private static class FlyweightMap
extends AbstractMap<String,String> {
  private static class Entry
  implements Map.Entry<String,String> {
    int index;
    Entry(int index) { this.index = index; }
    @Override public boolean equals(Object o) {
      return o instanceof FlyweightMap &&
        Objects.equals(DATA[index][0], o);
    }
    @Override public int hashCode() {
      return Objects.hashCode(DATA[index][0]);
    }
    @Override
    public String getKey() { return DATA[index][0]; }
    @Override public String getValue() {
      return DATA[index][1];
    }
    @Override public String setValue(String value) {
      throw new UnsupportedOperationException();
    }
  }
  // Implement size() & iterator() for AbstractSet:
  static class EntrySet
  extends AbstractSet<Map.Entry<String,String>> {
    private int size;
    EntrySet(int size) {
      if(size < 0)
        this.size = 0;
      // Can't be any bigger than the array:
      else if(size > DATA.length)
        this.size = DATA.length;
      else
        this.size = size;
    }
    @Override public int size() { return size; }
    private class Iter
    implements Iterator<Map.Entry<String,String>> {
      // Only one Entry object per Iterator:
      private Entry entry = new Entry(-1);
      @Override public boolean hasNext() {
```

```
        return entry.index < size - 1;
      }
      @Override
      public Map.Entry<String,String> next() {
        entry.index++;
        return entry;
      }
      @Override public void remove() {
        throw new UnsupportedOperationException();
      }
    }
    @Override public
    Iterator<Map.Entry<String,String>> iterator() {
      return new Iter();
    }
  }
  private static
  Set<Map.Entry<String,String>> entries =
    new EntrySet(DATA.length);
  @Override
  public Set<Map.Entry<String,String>> entrySet() {
    return entries;
  }
}
// Create a partial map of 'size' countries:
static Map<String,String> select(final int size) {
  return new FlyweightMap() {
    @Override
    public Set<Map.Entry<String,String>> entrySet() {
      return new EntrySet(size);
    }
  };
}
static Map<String,String> map = new FlyweightMap();
public static Map<String,String> capitals() {
  return map; // The entire map
}
public static Map<String,String> capitals(int size) {
  return select(size); // A partial map
}
static List<String> names =
  new ArrayList<>(map.keySet());
```

```
  // All the names:
  public static List<String> names() { return names; }
  // A partial list:
  public static List<String> names(int size) {
    return new ArrayList<>(select(size).keySet());
  }
  public static void main(String[] args) {
    System.out.println(capitals(10));
    System.out.println(names(10));
    System.out.println(new HashMap<>(capitals(3)));
    System.out.println(
      new LinkedHashMap<>(capitals(3)));
    System.out.println(new TreeMap<>(capitals(3)));
    System.out.println(new Hashtable<>(capitals(3)));
    System.out.println(new HashSet<>(names(6)));
    System.out.println(new LinkedHashSet<>(names(6)));
    System.out.println(new TreeSet<>(names(6)));
    System.out.println(new ArrayList<>(names(6)));
    System.out.println(new LinkedList<>(names(6)));
    System.out.println(capitals().get("BRAZIL"));
  }
}
/* Output:
{ALGERIA=Algiers, ANGOLA=Luanda, BENIN=Porto-Novo,
BOTSWANA=Gaberone, BURKINA FASO=Ouagadougou,
BURUNDI=Bujumbura, CAMEROON=Yaounde, CAPE VERDE=Praia,
CENTRAL AFRICAN REPUBLIC=Bangui, CHAD=N'djamena}
[ALGERIA, ANGOLA, BENIN, BOTSWANA, BURKINA FASO,
BURUNDI, CAMEROON, CAPE VERDE, CENTRAL AFRICAN
REPUBLIC, CHAD]
{BENIN=Porto-Novo, ANGOLA=Luanda, ALGERIA=Algiers}
{ALGERIA=Algiers, ANGOLA=Luanda, BENIN=Porto-Novo}
{ALGERIA=Algiers, ANGOLA=Luanda, BENIN=Porto-Novo}
{ALGERIA=Algiers, ANGOLA=Luanda, BENIN=Porto-Novo}
[BENIN, BOTSWANA, ANGOLA, BURKINA FASO, ALGERIA,
BURUNDI]
[ALGERIA, ANGOLA, BENIN, BOTSWANA, BURKINA FASO,
BURUNDI]
[ALGERIA, ANGOLA, BENIN, BOTSWANA, BURKINA FASO,
BURUNDI]
[ALGERIA, ANGOLA, BENIN, BOTSWANA, BURKINA FASO,
BURUNDI]
```

```
[ALGERIA, ANGOLA, BENIN, BOTSWANA, BURKINA FASO,
BURUNDI]
Brasilia
*/
```

The `String DATA` two-dimensional array is `public` so it can be used elsewhere. `FlyweightMap` must implement the `entrySet()` method, which requires both a custom `Set` implementation and a custom `Map.Entry` class. Here's another way to implement a flyweight: each `Map.Entry` object stores its index, rather than the actual key and value. When you call `getKey()` or `getValue()`, it uses the index to return the appropriate `DATA` element. The `EntrySet` ensures its `size` is no bigger than `DATA`.

The other part of the flyweight is implemented in `EntrySet.Iterator`. Instead of creating a `Map.Entry` object for each data pair in `DATA`, there's only one `Map.Entry` object *per iterator*. The `Entry` object is used as a window into the data; it only contains an `index` into the static array of `Strings`. Every time you call `next()` for the iterator, the `index` in the `Entry` is incremented so it points to the next element pair, then that `Iterator`'s single `Entry` object is returned from `next()`.[198]

The `select()` method produces a `FlyweightMap` containing an `EntrySet` of the desired size, and this is used in the overloaded `capitals()` and `names()` methods you see demonstrated in `main()`.

`Collection` Functionality

The following table shows everything you can do with a `Collection` (not including the methods that automatically come through with `Object`), and thus, everything you can do with a `List`, `Set`, `Queue` or `Deque` (These interfaces may also provide also has additional functionality). `Maps` are not inherited from `Collection` and are treated separately.

[198] The `Maps` in `java.util` perform bulk copies using `getKey()` and `getValue()` for `Maps`, so this works. If a custom `Map` were to simply copy the entire `Map.Entry` then this approach would cause a problem.

Operation	Effect
`boolean add(T)`	Ensures that the collection holds the argument of generic type `T`. Returns `false` if it doesn't add the argument. (This is an "optional" method, described in the next section.)
`boolean addAll(Collection<? extends T>)`	Adds all the elements in the argument. Returns `true` if any elements were added. ("Optional.")
`void clear()`	Removes all the elements in the collection. ("Optional.")
`boolean contains(T)`	`true` if the collection holds the argument of generic type `T`.
`Boolean containsAll(Collection<?>)`	`true` if the collection holds all the elements in the argument.
`boolean isEmpty()`	`true` if the collection has no elements.
`Iterator<T> iterator()` `Spliterator<T> spliterator()`	Returns an iterator to move through the elements in the collection. `Spliterators` are much more complex, and used for concurrency.
`Boolean remove(Object)`	If the argument is in the collection, one instance of that element is removed. Returns `true` if a removal occurred. ("Optional.")
`boolean removeAll(Collection<?>)`	Removes all the elements contained in the argument. Returns `true` if any removals occurred. ("Optional.")
`Boolean retainAll(Collection<?>)`	Retains only elements contained in the argument (an "intersection," from set theory). Returns `true` if any changes occurred. ("Optional.")

Operation	Effect
`boolean removeIf(Predicate<? super E>)`	Removes every element in this collection that satisfies the given predicate.
`Stream<E> stream()` `Stream<E> parallelStream()`	Returns a `Stream` of the elements in this `Collection`
`int size()`	Returns the number of elements in the collection.
`Object[] toArray()`	Returns an array containing all the elements in the collection.
`<T> T[] toArray(T[] a)`	Returns an array containing all the elements in the collection. The runtime type of the result is that of the argument array a rather than plain `Object`.

There's no `get()` method for random-access element selection because `Collection` also includes `Set`, which maintains its own internal ordering (and thus makes random-access lookup meaningless). Thus, to examine the elements of a `Collection`, you must use an iterator.

This demonstrates all `Collection` methods. `ArrayList` is used as a "least-common denominator" `Collection`:

```
// collectiontopics/CollectionMethods.java
// Things you can do with all Collections
import java.util.*;
import static onjava.HTMLColors.*;

public class CollectionMethods {
  public static void main(String[] args) {
    Collection<String> c =
        new ArrayList<>(LIST.subList(0, 4));
    c.add("ten");
    c.add("eleven");
    show(c);
    border();
    // Make an array from the List:
    Object[] array = c.toArray();
```

```
    // Make a String array from the List:
    String[] str = c.toArray(new String[0]);
    // Find max and min elements; this means
    // different things depending on the way
    // the Comparable interface is implemented:
    System.out.println(
      "Collections.max(c) = " + Collections.max(c));
    System.out.println(
      "Collections.min(c) = " + Collections.min(c));
    border();
    // Add a Collection to another Collection
    Collection<String> c2 =
        new ArrayList<>(LIST.subList(10, 14));
    c.addAll(c2);
    show(c);
    border();
    c.remove(LIST.get(0));
    show(c);
    border();
    // Remove all components that are
    // in the argument collection:
    c.removeAll(c2);
    show(c);
    border();
    c.addAll(c2);
    show(c);
    border();
    // Is an element in this Collection?
    String val = LIST.get(3);
    System.out.println(
      "c.contains(" + val  + ") = " + c.contains(val));
    // Is a Collection in this Collection?
    System.out.println(
      "c.containsAll(c2) = " + c.containsAll(c2));
    Collection<String> c3 =
      ((List<String>)c).subList(3, 5);
    // Keep all the elements that are in both
    // c2 and c3 (an intersection of sets):
    c2.retainAll(c3);
    show(c2);
    // Discard all c2 elements that also appear in c3:
    c2.removeAll(c3);
```

```
    System.out.println(
      "c2.isEmpty() = " +  c2.isEmpty());
    border();
    // Functional operation:
    c = new ArrayList<>(LIST);
    c.removeIf(s -> !s.startsWith("P"));
    c.removeIf(s -> s.startsWith("Pale"));
    // Stream operation:
    c.stream().forEach(System.out::println);
    c.clear(); // Remove all elements
    System.out.println("after c.clear():" + c);
  }
}
/* Output:
AliceBlue
AntiqueWhite
Aquamarine
Azure
ten
eleven
*****************************
Collections.max(c) = ten
Collections.min(c) = AliceBlue
*****************************
AliceBlue
AntiqueWhite
Aquamarine
Azure
ten
eleven
Brown
BurlyWood
CadetBlue
Chartreuse
*****************************
AntiqueWhite
Aquamarine
Azure
ten
eleven
Brown
BurlyWood
```

```
CadetBlue
Chartreuse
*****************************
AntiqueWhite
Aquamarine
Azure
ten
eleven
*****************************
AntiqueWhite
Aquamarine
Azure
ten
eleven
Brown
BurlyWood
CadetBlue
Chartreuse
*****************************
c.contains(Azure) = true
c.containsAll(c2) = true
c2.isEmpty() = true
*****************************
PapayaWhip
PeachPuff
Peru
Pink
Plum
PowderBlue
Purple
after c.clear():[]
*/
```

To demonstrate that nothing other than the `Collection` interface is used, `ArrayLists` are created containing different sets of data and upcast to `Collection` objects.

Optional Operations

The methods in the `Collection` interface that perform various kinds of addition and removal are *optional operations*. This means the implementing class is not required to provide functioning definitions for these methods.

This is a very unusual way to define an interface. As you've seen, an interface is a contract. It says, "No matter how you choose to implement this interface, I guarantee you can send these messages to this object" (I use the term "interface" here to describe both the formal `interface` keyword and the more general meaning of "the methods supported by any class or subclass"). But an "optional" operation violates this very fundamental principle; it says that calling some methods will *not* perform meaningful behavior. Instead, they will throw exceptions! It appears that compile-time type safety is discarded.

It's not that bad. If an operation is optional, the compiler still restricts you to calling only the methods in that interface. It's not like a dynamic language, where you can call any method for any object, and find out at run time whether a particular call will work.[199] In addition, most methods that take a `Collection` as an argument only *read* from that `Collection`, and all the "read" methods of `Collection` are *not* optional.

Why would you define methods as "optional?" Doing so prevents an explosion of interfaces in the design. Other designs for collection libraries tend to produce a confusing plethora of interfaces to describe each of the variations on the main theme. It's not even possible to capture all special cases in interfaces, because someone can always invent a new interface. The "unsupported operation" approach achieves an important goal of the Java collections library: The collections are simple to learn and use. Unsupported operations are a special case that can be delayed until necessary. For this approach to work, however:

1. The `UnsupportedOperationException` must be a rare event. That is, for most classes, all operations should work, and only in special cases should an operation be unsupported. This is true in the Java collections library because the classes you'll use 99 percent of the time—`ArrayList`, `LinkedList`, `HashSet`, and `HashMap`, as well as the other concrete implementations—support all

[199] Although this sounds odd and possibly useless when I describe it this way, you've seen, especially in the Reflection chapter, that this kind of dynamic behavior can be very powerful.

operations. The design does provide a "back door" to create a new `Collection` without providing meaningful definitions for all the methods in the `Collection` interface, that still fits into the existing library.

2. When an operation *is* unsupported, it should be reasonably likely that an `UnsupportedOperationException` will appear at implementation time, rather than after you've shipped the product to the customer. After all, it indicates a programming error: You've used an implementation incorrectly.

It's worth noting that unsupported operations are only detectable at run time, and therefore represent dynamic type checking. If you're coming from a statically typed language like C++, Java might appear to be just another statically typed language. Java certainly *has* static type checking, but it also has a significant amount of dynamic typing, so it's hard to say it's exactly one type of language or another. Once you begin to notice this, you'll start to see other examples of dynamic type checking in Java.

Unsupported Operations

A common source of unsupported operations are collections backed by fixed-sized data structures. You get such a collection when you turn an array into a `List` with the `Arrays.asList()` method. You can also *choose* to make any collection (including a `Map`) throw `UnsupportedOperationExceptions` by using the "unmodifiable" methods in the `Collections` class. This example shows both cases:

```
// collectiontopics/Unsupported.java
// Unsupported operations in Java collections
import java.util.*;

public class Unsupported {
  static void
  check(String description, Runnable tst) {
    try {
      tst.run();
    } catch(Exception e) {
      System.out.println(description + "(): " + e);
    }
  }
  static void test(String msg, List<String> list) {
```

```
    System.out.println("--- " + msg + " ---");
    Collection<String> c = list;
    Collection<String> subList = list.subList(1,8);
    // Copy of the sublist:
    Collection<String> c2 = new ArrayList<>(subList);
    check("retainAll", () -> c.retainAll(c2));
    check("removeAll", () -> c.removeAll(c2));
    check("clear", () -> c.clear());
    check("add", () -> c.add("X"));
    check("addAll", () -> c.addAll(c2));
    check("remove", () -> c.remove("C"));
    // The List.set() method modifies the value but
    // doesn't change the size of the data structure:
    check("List.set", () -> list.set(0, "X"));
  }
  public static void main(String[] args) {
    List<String> list = Arrays.asList(
      "A B C D E F G H I J K L".split(" "));
    test("Modifiable Copy", new ArrayList<>(list));
    test("Arrays.asList()", list);
    test("unmodifiableList()",
      Collections.unmodifiableList(
        new ArrayList<>(list)));
  }
}
/* Output:
--- Modifiable Copy ---
--- Arrays.asList() ---
retainAll(): java.lang.UnsupportedOperationException
removeAll(): java.lang.UnsupportedOperationException
clear(): java.lang.UnsupportedOperationException
add(): java.lang.UnsupportedOperationException
addAll(): java.lang.UnsupportedOperationException
remove(): java.lang.UnsupportedOperationException
--- unmodifiableList() ---
retainAll(): java.lang.UnsupportedOperationException
removeAll(): java.lang.UnsupportedOperationException
clear(): java.lang.UnsupportedOperationException
add(): java.lang.UnsupportedOperationException
addAll(): java.lang.UnsupportedOperationException
remove(): java.lang.UnsupportedOperationException
List.set(): java.lang.UnsupportedOperationException
```

```
*/
```

Because `Arrays.asList()` produces a `List` that is backed by a fixed-size array, it makes sense that the only supported operations are the ones that don't change the *size* of the array. Any method that would cause a change to the size of the underlying data structure produces an `UnsupportedOperationException`, to indicate a call to an unsupported method (a programming error).

Note you can always pass the result of `Arrays.asList()` as a constructor argument to any `Collection` (or use the `addAll()` method, or the `static Collections.addAll()` method) to create a regular collection that allows all the methods—this is shown in the first call to `test()` in `main()`. Such a call produces a new resizable underlying data structure.

The "unmodifiable" methods in the `Collections` class wrap the collection in a proxy that produces an `UnsupportedOperationException` if you perform any operation that modifies the collection in any way. The goal of using these methods is to produce a "constant" collection object. The full list of "unmodifiable" `Collections` methods is described later.

The last `check()` in `test()` examines the `set()` method that's part of `List`. Here, the granularity of the "unsupported operation" technique comes in handy—the resulting "interface" can vary by one method between the object returned by `Arrays.asList()` and that returned by `Collections.unmodifiableList()`. `Arrays.asList()` returns a fixed-sized `List`, but `Collections.unmodifiableList()` produces a list that cannot be changed. As seen in the output, it's OK to *modify* the elements in the `List` returned by `Arrays.asList()`, because this would not violate the "fixed-sized" nature of that `List`. But clearly, the result of `unmodifiableList()` should not be modifiable in any way. If interfaces were used, this would require two additional interfaces, one with a working `set()` method and one without. Additional interfaces would be required for various unmodifiable subtypes of `Collection`.

The documentation for a method that takes a collection as an argument should specify which of the optional methods must be implemented.

Sets and Storage Order

The Set examples in the Collections chapter provide a good introduction to the operations on basic Sets. However, those examples conveniently use predefined Java types such as Integer and String, which were designed to be usable inside collections. When creating your own types, be aware that a Set (and also a Map, which we'll look at shortly) needs a way to maintain storage order, which varies from one implementation of Set to another. Thus, different Set implementations not only have different behaviors, they have different requirements for the type of object you can put into a particular Set:

Collection	Behavior
Set (interface)	Each element you add to the Set must be unique; otherwise, the Set doesn't add the duplicate element. Elements added to a Set must at least define equals() to establish object uniqueness. Set has exactly the same interface as Collection. The Set interface does not guarantee it will maintain its elements in any particular order.
HashSet*	For Sets where fast lookup time is important. Elements must define hashCode() and equals().
TreeSet	An ordered Set backed by a tree. This way, you can extract an ordered sequence from a Set. Elements must also implement the Comparable interface.
LinkedHashSet	Has the lookup speed of a HashSet, but internally maintains the order you add the elements (the insertion order) using a linked list. Thus, when you iterate through the Set, the results appear in insertion order. Elements must define hashCode() and equals().

The asterisk on HashSet indicates that, in the absence of other constraints, this should be your default choice because it is optimized for speed.

Defining hashCode() is described in the Appendix: Understanding equals() and hashCode(). You must create an equals() for both hashed and tree storage, but the hashCode() is necessary only if the class is placed in a HashSet (this is likely because that should generally be your first choice as a Set implementation) or LinkedHashSet. However, for good programming style, always override hashCode() when you

override `equals()`.

This example demonstrates the methods required to successfully use a type with a particular `Set` implementation:

```
// collectiontopics/TypesForSets.java
// Methods necessary to put your own type in a Set
import java.util.*;
import java.util.function.*;
import java.util.Objects;

class SetType {
  protected int i;
  SetType(int n) { i = n; }
  @Override public boolean equals(Object o) {
    return o instanceof SetType &&
      Objects.equals(i, ((SetType)o).i);
  }
  @Override public String toString() {
    return Integer.toString(i);
  }
}

class HashType extends SetType {
  HashType(int n) { super(n); }
  @Override public int hashCode() {
    return Objects.hashCode(i);
  }
}

class TreeType extends SetType
implements Comparable<TreeType> {
  TreeType(int n) { super(n); }
  @Override public int compareTo(TreeType arg) {
    return Integer.compare(arg.i, i);
    // Equivalent to:
    // return arg.i < i ? -1 : (arg.i == i ? 0 : 1);
  }
}

public class TypesForSets {
  static <T> void
```

```
  fill(Set<T> set, Function<Integer, T> type) {
    for(int i = 10; i >= 5; i--) // Descending
      set.add(type.apply(i));
    for(int i = 0; i < 5; i++) // Ascending
      set.add(type.apply(i));
  }
  static <T> void
  test(Set<T> set, Function<Integer, T> type) {
    fill(set, type);
    fill(set, type); // Try to add duplicates
    fill(set, type);
    System.out.println(set);
  }
  public static void main(String[] args) {
    test(new HashSet<>(), HashType::new);
    test(new LinkedHashSet<>(), HashType::new);
    test(new TreeSet<>(), TreeType::new);
    // Things that don't work:
    test(new HashSet<>(), SetType::new);
    test(new HashSet<>(), TreeType::new);
    test(new LinkedHashSet<>(), SetType::new);
    test(new LinkedHashSet<>(), TreeType::new);
    try {
      test(new TreeSet<>(), SetType::new);
    } catch(Exception e) {
      System.out.println(e.getMessage());
    }
    try {
      test(new TreeSet<>(), HashType::new);
    } catch(Exception e) {
      System.out.println(e.getMessage());
    }
  }
}
/* Output:
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
[10, 9, 8, 7, 6, 5, 0, 1, 2, 3, 4]
[10, 9, 8, 7, 6, 5, 4, 3, 2, 1, 0]
[4, 1, 5, 1, 0, 5, 6, 8, 7, 0, 2, 8, 4, 9, 6, 10, 6, 7,
2, 9, 10, 3, 8, 4, 10, 3, 9, 5, 3, 7, 1, 2, 0]
[9, 8, 4, 8, 5, 0, 1, 6, 2, 9, 3, 7, 2, 2, 7, 0, 9, 5,
5, 6, 4, 7, 10, 1, 6, 4, 1, 10, 3, 3, 0, 10, 8]
```

```
[10, 9, 8, 7, 6, 5, 0, 1, 2, 3, 4, 10, 9, 8, 7, 6, 5,
0, 1, 2, 3, 4, 10, 9, 8, 7, 6, 5, 0, 1, 2, 3, 4]
[10, 9, 8, 7, 6, 5, 0, 1, 2, 3, 4, 10, 9, 8, 7, 6, 5,
0, 1, 2, 3, 4, 10, 9, 8, 7, 6, 5, 0, 1, 2, 3, 4]
SetType cannot be cast to java.lang.Comparable
HashType cannot be cast to java.lang.Comparable
*/
```

To prove which methods are necessary for a particular `Set` and at the same time to avoid code duplication, three classes are created. The base class, `SetType`, stores an `int`, and produces it via `toString()`. All classes stored in `Sets` must have an `equals()` so that method is also placed in the base class. Equality is based on the value of the `int i`.

`HashType` inherits from `SetType` and adds the `hashCode()` method necessary for an object to be placed in a hashed implementation of a `Set`.

The `Comparable` interface, implemented by `TreeType`, is necessary to use an object in any kind of sorted collection, such as a `SortedSet` (of which `TreeSet` is the only implementation). In `compareTo()`, note I did *not* use the "simple and obvious" form `return i-i2`. Although this is a common programming error, it would only work properly if `i` and `i2` were "unsigned" `ints` (if Java *had* an "unsigned" keyword, which it does not). It breaks for Java's signed `int`, which is not big enough to represent the difference of two signed `ints`. If `i` is a large positive integer and `j` is a large negative integer, `i-j` will overflow and return a negative value, which will not work.

You'll usually want the `compareTo()` method to produce a natural ordering consistent with the `equals()` method. If `equals()` produces `true` for a particular comparison, `compareTo()` should produce a zero result for that comparison, and if `equals()` produces `false` for a comparison then `compareTo()` should produce a nonzero result for that comparison.

In `TypesForSets`, both `fill()` and `test()` are defined using generics, to prevent code duplication. To verify the behavior of a `Set`, `test()` calls `fill()` on the test `set` three times, attempting to introduce duplicate objects. The `fill()` method takes a `Set` of any type, and a `Function` object that produces that type. Because all the objects used in this example have a constructor that takes a single `int` argument, you can pass the constructor as this `Function` and it will supply the objects to fill the `Set`.

Note that the `fill()` method adds its first elements in descending order, and the last in ascending order, to point out the resulting storage order. The output shows that the `HashSet` keeps the elements in ascending order—however, in the Appendix: Understanding `equals()` and `hashCode()` you'll see that this is incidental, because hashing creates its own storage order. Only because our value is a simple `int` is it ascending in this case. The `LinkedHashSet` keeps the elements in the order they were inserted, and the `TreeSet` maintains the elements in sorted order (descending order in this example, because of the way `compareTo()` is implemented).

If we try to use types that don't properly support the necessary operations with `Sets` that require those operations, things go very wrong. Placing a `SetType` or `TreeType` object, which doesn't include a redefined `hashCode()` method, into any hashed implementations results in duplicate values, so the primary contract of the `Set` is violated. This is rather disturbing because there's not even a runtime error. However, the default `hashCode()` is legitimate and so this is legal behavior, even if it's incorrect. The only reliable way to ensure the correctness of such a program is to incorporate unit tests into your build system.

If you try to use a type that doesn't implement `Comparable` in a `TreeSet`, you get a more definitive result: An exception is thrown when the `TreeSet` attempts to use the object as a `Comparable`.

SortedSet

The elements in a `SortedSet` are guaranteed to be in sorted order, yielding additional functionality from the following methods in the `SortedSet` interface:

- `Comparator comparator()`: Produces the `Comparator` used for this `Set`, or `null` for natural ordering.
- `Object first()`: Produces the lowest element.
- `Object last()`: Produces the highest element.
- `SortedSet subSet(fromElement, toElement)`: Produces a view of this `Set` with elements from `fromElement`, inclusive, to `toElement`, exclusive.
- `SortedSet headSet(toElement)`: Produces a view of this `Set` with elements less than `toElement`.
- `SortedSet tailSet(fromElement)`: Produces a view of this `Set` with elements greater than or equal to `fromElement`.

Here's a simple demonstration:

```
// collectiontopics/SortedSetDemo.java
import java.util.*;
import static java.util.stream.Collectors.*;

public class SortedSetDemo {
  public static void main(String[] args) {
    SortedSet<String> sortedSet =
      Arrays.stream(
        "one two three four five six seven eight"
        .split(" "))
        .collect(toCollection(TreeSet::new));
    System.out.println(sortedSet);
    String low = sortedSet.first();
    String high = sortedSet.last();
    System.out.println(low);
    System.out.println(high);
    Iterator<String> it = sortedSet.iterator();
    for(int i = 0; i <= 6; i++) {
      if(i == 3) low = it.next();
      if(i == 6) high = it.next();
      else it.next();
    }
    System.out.println(low);
    System.out.println(high);
    System.out.println(sortedSet.subSet(low, high));
    System.out.println(sortedSet.headSet(high));
    System.out.println(sortedSet.tailSet(low));
  }
}
/* Output:
[eight, five, four, one, seven, six, three, two]
eight
two
one
two
[one, seven, six, three]
[eight, five, four, one, seven, six, three]
[one, seven, six, three, two]
*/
```

Note that `SortedSet` means "sorted according to the comparison function of the object," not "insertion order." Insertion order can be preserved using a `LinkedHashSet`.

Queues

There are numerous Queue implementations, most of which are designed for concurrency applications. Some are differentiated by ordering behavior rather than performance. Here's a basic example that involves most of the Queue implementations, including the concurrency-based Queues. You place elements in one end and extract them from the other:

```
// collectiontopics/QueueBehavior.java
// Compares basic behavior
import java.util.*;
import java.util.stream.*;
import java.util.concurrent.*;

public class QueueBehavior {
  static Stream<String> strings() {
    return Arrays.stream(
      ("one two three four five six seven " +
       "eight nine ten").split(" "));
  }
  static void test(int id, Queue<String> queue) {
    System.out.print(id + ": ");
    strings().map(queue::offer).count();
    while(queue.peek() != null)
      System.out.print(queue.remove() + " ");
    System.out.println();
  }
  public static void main(String[] args) {
    int count = 10;
    test(1, new LinkedList<>());
    test(2, new PriorityQueue<>());
    test(3, new ArrayBlockingQueue<>(count));
    test(4, new ConcurrentLinkedQueue<>());
    test(5, new LinkedBlockingQueue<>());
    test(6, new PriorityBlockingQueue<>());
    test(7, new ArrayDeque<>());
    test(8, new ConcurrentLinkedDeque<>());
    test(9, new LinkedBlockingDeque<>());
    test(10, new LinkedTransferQueue<>());
    test(11, new SynchronousQueue<>());
```

```
  }
}
/* Output:
1: one two three four five six seven eight nine ten
2: eight five four nine one seven six ten three two
3: one two three four five six seven eight nine ten
4: one two three four five six seven eight nine ten
5: one two three four five six seven eight nine ten
6: eight five four nine one seven six ten three two
7: one two three four five six seven eight nine ten
8: one two three four five six seven eight nine ten
9: one two three four five six seven eight nine ten
10: one two three four five six seven eight nine ten
11:
*/
```

The `Deque` interface also inherits from `Queue`. With the exception of the priority queues, a `Queue` produces elements in the same order as they are placed in the `Queue`. In this example, `SynchronousQueue` doesn't produce any results because it is a blocking queue where each insert operation must wait for a corresponding remove operation by another thread, and vice versa.

Priority Queues

Consider a to-do list, where each object contains a `String` and a primary and secondary priority value. The ordering of this list is controlled by implementing `Comparable`:

```
// collectiontopics/ToDoList.java
// A more complex use of PriorityQueue
import java.util.*;

class ToDoItem implements Comparable<ToDoItem> {
  private char primary;
  private int secondary;
  private String item;
  ToDoItem(String td, char pri, int sec) {
    primary = pri;
    secondary = sec;
    item = td;
```

```
  }
  @Override public int compareTo(ToDoItem arg) {
    if(primary > arg.primary)
      return +1;
    if(primary == arg.primary)
      if(secondary > arg.secondary)
        return +1;
      else if(secondary == arg.secondary)
        return 0;
    return -1;
  }
  @Override public String toString() {
    return Character.toString(primary) +
      secondary + ": " + item;
  }
}

class ToDoList {
  public static void main(String[] args) {
    PriorityQueue<ToDoItem> toDo =
      new PriorityQueue<>();
    toDo.add(new ToDoItem("Empty trash", 'C', 4));
    toDo.add(new ToDoItem("Feed dog", 'A', 2));
    toDo.add(new ToDoItem("Feed bird", 'B', 7));
    toDo.add(new ToDoItem("Mow lawn", 'C', 3));
    toDo.add(new ToDoItem("Water lawn", 'A', 1));
    toDo.add(new ToDoItem("Feed cat", 'B', 1));
    while(!toDo.isEmpty())
      System.out.println(toDo.remove());
  }
}
/* Output:
A1: Water lawn
A2: Feed dog
B1: Feed cat
B7: Feed bird
C3: Mow lawn
C4: Empty trash
*/
```

This shows the automatic ordering of the items via the priority queue.

Deque

A Deque (double-ended queue) is like a queue, but you can add and remove elements from either end. Java 6 added an explicit interface for Deque. Here is a test of the most fundamental Deque methods for the classes that implement Deque:

```
// collectiontopics/SimpleDeques.java
// Very basic test of Deques
import java.util.*;
import java.util.concurrent.*;
import java.util.function.*;

class CountString implements Supplier<String> {
  private int n = 0;
  CountString() {}
  CountString(int start) { n = start; }
  @Override public String get() {
    return Integer.toString(n++);
  }
}

public class SimpleDeques {
  static void test(Deque<String> deque) {
    CountString s1 = new CountString(),
                s2 = new CountString(20);
    for(int n = 0; n < 8; n++) {
        deque.offerFirst(s1.get());
        deque.offerLast(s2.get()); // Same as offer()
    }
    System.out.println(deque);
    String result = "";
    while(deque.size() > 0) {
      System.out.print(deque.peekFirst() + " ");
      result += deque.pollFirst() + " ";
      System.out.print(deque.peekLast() + " ");
      result += deque.pollLast() + " ";
    }
    System.out.println("\n" + result);
  }
  public static void main(String[] args) {
    int count = 10;
```

```
    System.out.println("LinkedList");
    test(new LinkedList<>());
    System.out.println("ArrayDeque");
    test(new ArrayDeque<>());
    System.out.println("LinkedBlockingDeque");
    test(new LinkedBlockingDeque<>(count));
    System.out.println("ConcurrentLinkedDeque");
    test(new ConcurrentLinkedDeque<>());
  }
}
/* Output:
LinkedList
[7, 6, 5, 4, 3, 2, 1, 0, 20, 21, 22, 23, 24, 25, 26,
27]
7 27 6 26 5 25 4 24 3 23 2 22 1 21 0 20
7 27 6 26 5 25 4 24 3 23 2 22 1 21 0 20
ArrayDeque
[7, 6, 5, 4, 3, 2, 1, 0, 20, 21, 22, 23, 24, 25, 26,
27]
7 27 6 26 5 25 4 24 3 23 2 22 1 21 0 20
7 27 6 26 5 25 4 24 3 23 2 22 1 21 0 20
LinkedBlockingDeque
[4, 3, 2, 1, 0, 20, 21, 22, 23, 24]
4 24 3 23 2 22 1 21 0 20
4 24 3 23 2 22 1 21 0 20
ConcurrentLinkedDeque
[7, 6, 5, 4, 3, 2, 1, 0, 20, 21, 22, 23, 24, 25, 26,
27]
7 27 6 26 5 25 4 24 3 23 2 22 1 21 0 20
7 27 6 26 5 25 4 24 3 23 2 22 1 21 0 20
*/
```

I've only used the "offer" and "poll" versions of the `Deque` methods because they don't throw exceptions when the `LinkedBlockingDeque` has a limited size. Notice that `LinkedBlockingDeque` only fills to its limit, then ignores further offers.

Understanding `Map`s

As you learned in the Collections chapter, a `Map` (also called an *associative array*) maintains key-value associations (pairs) so you can look up a value using a key.

The standard Java library contains different basic implementations of `Maps`, such as `HashMap`, `TreeMap`, `LinkedHashMap`, `WeakHashMap`, `ConcurrentHashMap`, and `IdentityHashMap`. They all have the same basic `Map` interface, but they differ in behaviors including efficiency, the order the pairs are held and presented, how long the objects are held by the map, how the map works in multithreaded programs, and how key equality is determined. The number of implementations of the `Map` interface should tell you something about the importance of this tool.

To gain a deeper understanding of `Maps`, it is helpful to learn how to construct an associative array. Here is an extremely simple implementation:

```
// collectiontopics/AssociativeArray.java
// Associates keys with values

public class AssociativeArray<K, V> {
  private Object[][] pairs;
  private int index;
  public AssociativeArray(int length) {
    pairs = new Object[length][2];
  }
  public void put(K key, V value) {
    if(index >= pairs.length)
      throw new ArrayIndexOutOfBoundsException();
    pairs[index++] = new Object[]{ key, value };
  }
  @SuppressWarnings("unchecked")
  public V get(K key) {
    for(int i = 0; i < index; i++)
      if(key.equals(pairs[i][0]))
        return (V)pairs[i][1];
    return null; // Did not find key
  }
  @Override public String toString() {
    StringBuilder result = new StringBuilder();
    for(int i = 0; i < index; i++) {
      result.append(pairs[i][0].toString());
      result.append(" : ");
      result.append(pairs[i][1].toString());
      if(i < index - 1)
        result.append("\n");
    }
```

```
    return result.toString();
  }
  public static void main(String[] args) {
    AssociativeArray<String,String> map =
      new AssociativeArray<>(6);
    map.put("sky", "blue");
    map.put("grass", "green");
    map.put("ocean", "dancing");
    map.put("tree", "tall");
    map.put("earth", "brown");
    map.put("sun", "warm");
    try {
      map.put("extra", "object"); // Past the end
    } catch(ArrayIndexOutOfBoundsException e) {
      System.out.println("Too many objects!");
    }
    System.out.println(map);
    System.out.println(map.get("ocean"));
  }
}
/* Output:
Too many objects!
sky : blue
grass : green
ocean : dancing
tree : tall
earth : brown
sun : warm
dancing
*/
```

The essential methods in an associative array are `put()` and `get()`, but for easy display, `toString()` is overridden to print the key-value pairs. To show it works, `main()` loads an `AssociativeArray` with pairs of `Strings` and prints the resulting map, followed by a `get()` of one of the values.

To use the `get()` method, you pass in the `key` you want it to look up, and it produces the associated value as the result or returns `null` if it can't be found. The `get()` method is using what is possibly the least efficient approach imaginable to locate the value: starting at the top of the array and using `equals()` to compare keys. But the point here is simplicity, not efficiency.

This version is instructive, but it isn't very efficient and it has a fixed size, which is inflexible. Fortunately, the `Maps` in `java.util` do not have these problems.

Performance

Performance is a fundamental issue for maps, and it's very slow to use a linear search in `get()` when hunting for a key. This is where `HashMap` speeds things up. Instead of a slow search for the key, it uses a special value called a *hash code*. The hash code is a way to take some information from the object in question and turn it into a "relatively unique" `int` for that object. `hashCode()` is a method in the root class `Object`, so all Java objects can produce a hash code. A `HashMap` takes the `hashCode()` of the object and uses it to quickly hunt for the key. This results in dramatic performance improvements.[200]

Here are the basic `Map` implementations. The asterisk on `HashMap` indicates that, in the absence of other constraints, this should be your default choice because it is optimized for speed. The other implementations emphasize other characteristics, and are thus not as fast as `HashMap`.

Collection	Behavior
`HashMap`*	Implementation based on a hash table. (Use this class instead of `Hashtable`.) Provides constant-time performance for inserting and locating pairs. Performance can be adjusted via constructors that allow you to set the *capacity* and *load factor* of the hash table.
`LinkedHashMap`	Like a `HashMap`, but when you iterate through, you get the pairs in insertion order, or in least-recently-used (LRU) order. Only slightly slower than `HashMap`, except when iterating, where it is faster due to the linked list used to maintain the internal ordering.

[200] If these speedups still don't meet your performance needs, you can further accelerate table lookup by writing your own `Map` and customizing it to your particular types to avoid delays due to casting to and from `Objects`. To reach even higher levels of performance, speed enthusiasts can use Donald Knuth's *The Art of Computer Programming, Volume 3: Sorting and Searching, Second Edition*, to replace overflow bucket lists with arrays that have two additional benefits: they can be optimized for disk storage characteristics and they can save most of the time of creating and garbage collecting individual records.

Collection	Behavior
`TreeMap`	Implementation based on a red-black tree. When you view the keys or the pairs, they are in sorted order (determined by `Comparable` or `Comparator`). The point of a `TreeMap` is that you get the results in sorted order. `TreeMap` is the only `Map` with the `subMap()` method, which returns a portion of the tree.
`WeakHashMap`	A `Map` of *weak keys* that allow objects referred to by the map to be released; designed to solve certain types of problems. If no references to a particular key are held outside the map, that key can be garbage collected.
`ConcurrentHashMap`	A thread-safe `Map` that does not use synchronization locking. This is discussed in the Concurrent Programming chapter.
`IdentityHashMap`	A hash map that uses == instead of `equals()` to compare keys. Only for solving special types of problems; not for general use.

Hashing is the most commonly used way to store elements in a map.

The requirements for the keys used in a `Map` are the same as for the elements in a `Set`. You saw these demonstrated in `TypesForSets.java`. Any key must have an `equals()` method. If the key is used in a hashed `Map`, it must also have a proper `hashCode()`. If the key is used in a `TreeMap`, it must implement `Comparable`.

The following example shows the operations available through the `Map` interface, using the previously defined `CountMap` test data set:

```
// collectiontopics/MapOps.java
// Things you can do with Maps
import java.util.concurrent.*;
import java.util.*;
import onjava.*;

public class MapOps {
  public static
  void printKeys(Map<Integer,String> map) {
    System.out.print("Size = " + map.size() + ", ");
    System.out.print("Keys: ");
    // Produce a Set of the keys:
    System.out.println(map.keySet());
```

```
  }
  public static
  void test(Map<Integer,String> map) {
    System.out.println(
      map.getClass().getSimpleName());
    map.putAll(new CountMap(25));
    // Map has 'Set' behavior for keys:
    map.putAll(new CountMap(25));
    printKeys(map);
    // Producing a Collection of the values:
    System.out.print("Values: ");
    System.out.println(map.values());
    System.out.println(map);
    System.out.println("map.containsKey(11): " +
      map.containsKey(11));
    System.out.println(
      "map.get(11): " + map.get(11));
    System.out.println("map.containsValue(\"F0\"): "
      + map.containsValue("F0"));
    Integer key = map.keySet().iterator().next();
    System.out.println("First key in map: " + key);
    map.remove(key);
    printKeys(map);
    map.clear();
    System.out.println(
      "map.isEmpty(): " + map.isEmpty());
    map.putAll(new CountMap(25));
    // Operations on the Set change the Map:
    map.keySet().removeAll(map.keySet());
    System.out.println(
      "map.isEmpty(): " + map.isEmpty());
  }
  public static void main(String[] args) {
    test(new HashMap<>());
    test(new TreeMap<>());
    test(new LinkedHashMap<>());
    test(new IdentityHashMap<>());
    test(new ConcurrentHashMap<>());
    test(new WeakHashMap<>());
  }
}
/* Output: (First 11 Lines)
```

```
HashMap
Size = 25, Keys: [0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11,
12, 13, 14, 15, 16, 17, 18, 19, 20, 21, 22, 23, 24]
Values: [A0, B0, C0, D0, E0, F0, G0, H0, I0, J0, K0,
L0, M0, N0, O0, P0, Q0, R0, S0, T0, U0, V0, W0, X0, Y0]
{0=A0, 1=B0, 2=C0, 3=D0, 4=E0, 5=F0, 6=G0, 7=H0, 8=I0,
9=J0, 10=K0, 11=L0, 12=M0, 13=N0, 14=O0, 15=P0, 16=Q0,
17=R0, 18=S0, 19=T0, 20=U0, 21=V0, 22=W0, 23=X0, 24=Y0}
map.containsKey(11): true
map.get(11): L0
map.containsValue("F0"): true
First key in map: 0
Size = 24, Keys: [1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11,
12, 13, 14, 15, 16, 17, 18, 19, 20, 21, 22, 23, 24]
map.isEmpty(): true
map.isEmpty(): true
                  ...
*/
```

The `printKeys()` method demonstrates how to produce a `Collection` view of a `Map`. The `keySet()` method produces a `Set` backed by the keys in the `Map`. Printing the result of the `values()` method produces a `Collection` containing all the values in the `Map`. (Note that keys must be unique, but values can contain duplicates.) These `Collections` are backed by the `Map` so any changes in a `Collection` are reflected in the associated `Map`.

The rest of the program provides simple examples of each `Map` operation and tests each basic type of `Map`.

SortedMap

With a `SortedMap` (implemented by `TreeMap` or `ConcurrentSkipListMap`), the keys are guaranteed to be in sorted order, which allows additional functionality to be provided with these methods in the `SortedMap` interface:

- `Comparator comparator()`: Produces the comparator used for this Map, or `null` for natural ordering.
- `T firstKey()`: Produces the lowest key.
- `T lastKey()`: Produces the highest key.

- `SortedMap subMap(fromKey, toKey)`: Produces a view of this Map with keys from `fromKey`, inclusive, to `toKey`, exclusive.
- `SortedMap headMap(toKey)`: Produces a view of this Map with keys less than `toKey`.
- `SortedMap tailMap(fromKey)`: Produces a view of this `Map` with keys greater than or equal to `fromKey`.

Here's an example that's similar to `SortedSetDemo.java` and shows this additional behavior of `TreeMaps`:

```
// collectiontopics/SortedMapDemo.java
// What you can do with a TreeMap
import java.util.*;
import onjava.*;

public class SortedMapDemo {
  public static void main(String[] args) {
    TreeMap<Integer,String> sortedMap =
      new TreeMap<>(new CountMap(10));
    System.out.println(sortedMap);
    Integer low = sortedMap.firstKey();
    Integer high = sortedMap.lastKey();
    System.out.println(low);
    System.out.println(high);
    Iterator<Integer> it =
      sortedMap.keySet().iterator();
    for(int i = 0; i <= 6; i++) {
      if(i == 3) low = it.next();
      if(i == 6) high = it.next();
      else it.next();
    }
    System.out.println(low);
    System.out.println(high);
    System.out.println(sortedMap.subMap(low, high));
    System.out.println(sortedMap.headMap(high));
    System.out.println(sortedMap.tailMap(low));
  }
}
/* Output:
{0=A0, 1=B0, 2=C0, 3=D0, 4=E0, 5=F0, 6=G0, 7=H0, 8=I0,
9=J0}
```

```
0
9
3
7
{3=D0, 4=E0, 5=F0, 6=G0}
{0=A0, 1=B0, 2=C0, 3=D0, 4=E0, 5=F0, 6=G0}
{3=D0, 4=E0, 5=F0, 6=G0, 7=H0, 8=I0, 9=J0}
*/
```

Here, the pairs are stored by key-sorted order. Because there is a sense of order in the `TreeMap`, the concept of "location" makes sense, so you can have first and last elements and submaps.

`LinkedHashMap`

The `LinkedHashMap` hashes everything for speed, but also produces the pairs in insertion order during a traversal (`System.out.println()` iterates through the map, so you see the results of traversal). In addition, a `LinkedHashMap` can be configured in the constructor to use a *least-recently-used* (LRU) algorithm based on accesses, so elements that haven't been accessed (and thus are candidates for removal) appear at the front of the list. This allows easy creation of programs that do periodic cleanup to save space. Here's a simple example showing both features:

```
// collectiontopics/LinkedHashMapDemo.java
// What you can do with a LinkedHashMap
import java.util.*;
import onjava.*;

public class LinkedHashMapDemo {
  public static void main(String[] args) {
    LinkedHashMap<Integer,String> linkedMap =
      new LinkedHashMap<>(new CountMap(9));
    System.out.println(linkedMap);
    // Least-recently-used order:
    linkedMap =
      new LinkedHashMap<>(16, 0.75f, true);
    linkedMap.putAll(new CountMap(9));
    System.out.println(linkedMap);
    for(int i = 0; i < 6; i++)
      linkedMap.get(i);
```

```
    System.out.println(linkedMap);
    linkedMap.get(0);
    System.out.println(linkedMap);
  }
}
/* Output:
{0=A0, 1=B0, 2=C0, 3=D0, 4=E0, 5=F0, 6=G0, 7=H0, 8=I0}
{0=A0, 1=B0, 2=C0, 3=D0, 4=E0, 5=F0, 6=G0, 7=H0, 8=I0}
{6=G0, 7=H0, 8=I0, 0=A0, 1=B0, 2=C0, 3=D0, 4=E0, 5=F0}
{6=G0, 7=H0, 8=I0, 1=B0, 2=C0, 3=D0, 4=E0, 5=F0, 0=A0}
*/
```

The pairs are indeed traversed in insertion order, even for the LRU version. However, after the first six items (only) are accessed in the LRU version, the last three items move to the front of the list. Then, when "`0`" is accessed again, it moves to the back of the list.

Utilities

There are standalone utilities for collections, expressed as `static` methods in `java.util.Collections`. You've already seen some of these, such as `addAll()`, `reverseOrder()` and `binarySearch()`. Here are the others (the `synchronized` and `unmodifiable` utilities are covered in sections that follow). In this table, generics are used when they are relevant:

Utility Function	Effect
`checkedCollection(Collection<T>, Class<T> type)` `checkedList(List<T>, Class<T> type)` `checkedMap(Map<K, V>, Class<K> keyType, Class<V> valueType)` `checkedSet(Set<T>, Class<T> type)` `checkedSortedMap(SortedMap<K, V>, Class<K> keyType, Class<V> valueType)` `checkedSortedSet(SortedSet<T>, Class<T> type)`	Produces a *dynamically* type-safe view of a `Collection`, or a specific subtype of `Collection`. Use this when it's not possible to use the statically checked version. These were shown in the Generics chapter under the heading "Dynamic type safety."

Utility Function	Effect
`max(Collection)` `min(Collection)`	Produces the maximum or minimum element in the argument using the natural comparison method of the objects in the `Collection`.
`max(Collection, Comparator)` `min(Collection, Comparator)`	Produces the maximum or minimum element in the `Collection` using the `Comparator`.
`indexOfSubList(List source, List target)`	Produces starting index of the *first* place where `target` appears inside `source`, or `-1` if none occurs.
`lastIndexOfSubList(List source, List target)`	Produces starting index of the *last* place where `target` appears inside `source`, or `-1` if none occurs.
`replaceAll(List<T>, T oldVal, T newVal)`	Replaces all `oldVal` with `newVal`.
`reverse(List)`	Reverses all the elements in place.
`reverseOrder()` `reverseOrder(Comparator<T>)`	Returns a `Comparator` that reverses the natural ordering of a collection of objects that implement `Comparable<T>`. The second version reverses the order of the supplied `Comparator`.
`rotate(List, int distance)`	Moves all elements forward by `distance`, taking the ones off the end and placing them at the beginning.
`shuffle(List)` `shuffle(List, Random)`	Randomly permutes the specified list. The first form provides its own randomization source, or you can provide your own with the second form.
`sort(List<T>)` `sort(List<T>, Comparator<? super T> c)`	Sorts the `List<T>` using its natural ordering. The second form takes a `Comparator` for sorting.

Utility Function	Effect
`copy(List<? super T> dest, List<? extends T> src)`	Copies elements from `src` to `dest`.
`swap(List, int i, int j)`	Swaps elements at locations `i` and `j` in the `List`. Probably faster than what you'd write by hand.
`fill(List<? super T>, T x)`	Replaces all the elements of the `List` with `x`.
`nCopies(int n, T x)`	Returns an immutable `List<T>` of size `n` whose references all point to `x`.
`disjoint(Collection, Collection)`	Returns `true` if the two `Collections` have no elements in common.
`frequency(Collection, Object x)`	Returns the number of elements in the `Collection` equal to `x`.
`emptyList()` `emptyMap()` `emptySet()`	Returns an immutable empty `List`, `Map`, or `Set`. These are generic, so the resulting `Collection` is parameterized to the desired type.
`singleton(T x)` `singletonList(T x)` `singletonMap(K key, V value)`	Produces an immutable `Set<T>`, `List<T>`, or `Map<K, V>` containing a single entry based on the given argument(s).
`list(Enumeration<T> e)`	Produces an `ArrayList<T>` containing the elements in the order in which they are returned by the (old-style) `Enumeration` (predecessor to the `Iterator`). For converting from legacy code.
`enumeration(Collection<T>)`	Produces an old-style `Enumeration<T>`.

Note that `min()` and `max()` work with `Collection` objects, not with `Lists`, so you don't worry if the `Collection` should be sorted or not. (As mentioned earlier, you *do* `sort()` a `List` or an array before performing a `binarySearch()`.)

Here's an example showing the basic use of most of the utilities in the above table:

```
// collectiontopics/Utilities.java
// Simple demonstrations of the Collections utilities
import java.util.*;

public class Utilities {
  static List<String> list = Arrays.asList(
    "one Two three Four five six one".split(" "));
  public static void main(String[] args) {
    System.out.println(list);
    System.out.println("'list' disjoint (Four)?: " +
      Collections.disjoint(list,
        Collections.singletonList("Four")));
    System.out.println(
      "max: " + Collections.max(list));
    System.out.println(
      "min: " + Collections.min(list));
    System.out.println(
      "max w/ comparator: " + Collections.max(list,
      String.CASE_INSENSITIVE_ORDER));
    System.out.println(
      "min w/ comparator: " + Collections.min(list,
      String.CASE_INSENSITIVE_ORDER));
    List<String> sublist =
      Arrays.asList("Four five six".split(" "));
    System.out.println("indexOfSubList: " +
      Collections.indexOfSubList(list, sublist));
    System.out.println("lastIndexOfSubList: " +
      Collections.lastIndexOfSubList(list, sublist));
    Collections.replaceAll(list, "one", "Yo");
    System.out.println("replaceAll: " + list);
    Collections.reverse(list);
    System.out.println("reverse: " + list);
    Collections.rotate(list, 3);
    System.out.println("rotate: " + list);
    List<String> source =
      Arrays.asList("in the matrix".split(" "));
    Collections.copy(list, source);
    System.out.println("copy: " + list);
    Collections.swap(list, 0, list.size() - 1);
    System.out.println("swap: " + list);
```

```
    Collections.shuffle(list, new Random(47));
    System.out.println("shuffled: " + list);
    Collections.fill(list, "pop");
    System.out.println("fill: " + list);
    System.out.println("frequency of 'pop': " +
      Collections.frequency(list, "pop"));
    List<String> dups =
      Collections.nCopies(3, "snap");
    System.out.println("dups: " + dups);
    System.out.println("'list' disjoint 'dups'?: " +
      Collections.disjoint(list, dups));
    // Getting an old-style Enumeration:
    Enumeration<String> e =
      Collections.enumeration(dups);
    Vector<String> v = new Vector<>();
    while(e.hasMoreElements())
      v.addElement(e.nextElement());
    // Converting an old-style Vector
    // to a List via an Enumeration:
    ArrayList<String> arrayList =
      Collections.list(v.elements());
    System.out.println("arrayList: " + arrayList);
  }
}
/* Output:
[one, Two, three, Four, five, six, one]
'list' disjoint (Four)?: false
max: three
min: Four
max w/ comparator: Two
min w/ comparator: five
indexOfSubList: 3
lastIndexOfSubList: 3
replaceAll: [Yo, Two, three, Four, five, six, Yo]
reverse: [Yo, six, five, Four, three, Two, Yo]
rotate: [three, Two, Yo, Yo, six, five, Four]
copy: [in, the, matrix, Yo, six, five, Four]
swap: [Four, the, matrix, Yo, six, five, in]
shuffled: [six, matrix, the, Four, Yo, five, in]
fill: [pop, pop, pop, pop, pop, pop, pop]
frequency of 'pop': 7
dups: [snap, snap, snap]
```

```
'list' disjoint 'dups'?: true
arrayList: [snap, snap, snap]
*/
```

The output explains the behavior of each utility method. Note the difference in `min()` and `max()` with the `String.CASE_INSENSITIVE_ORDER Comparator` because of capitalization.

Sorting and Searching `Lists`

Utilities to perform sorting and searching for `Lists` have the same names and signatures as those for sorting arrays of objects, but are `static` methods of `Collections` instead of `Arrays`. Here's an example that uses the `list` data from `Utilities.java`:

```
// collectiontopics/ListSortSearch.java
// Sorting/searching Lists with Collections utilities
import java.util.*;

public class ListSortSearch {
  public static void main(String[] args) {
    List<String> list =
      new ArrayList<>(Utilities.list);
    list.addAll(Utilities.list);
    System.out.println(list);
    Collections.shuffle(list, new Random(47));
    System.out.println("Shuffled: " + list);
    // Use ListIterator to trim off last elements:
    ListIterator<String> it = list.listIterator(10);
    while(it.hasNext()) {
      it.next();
      it.remove();
    }
    System.out.println("Trimmed: " + list);
    Collections.sort(list);
    System.out.println("Sorted: " + list);
    String key = list.get(7);
    int index = Collections.binarySearch(list, key);
    System.out.println(
      "Location of " + key + " is " + index +
      ", list.get(" + index + ") = " +
```

```
      list.get(index));
    Collections.sort(list,
      String.CASE_INSENSITIVE_ORDER);
    System.out.println(
      "Case-insensitive sorted: " + list);
    key = list.get(7);
    index = Collections.binarySearch(list, key,
      String.CASE_INSENSITIVE_ORDER);
    System.out.println(
      "Location of " + key + " is " + index +
      ", list.get(" + index + ") = " +
      list.get(index));
  }
}
/* Output:
[one, Two, three, Four, five, six, one, one, Two,
three, Four, five, six, one]
Shuffled: [Four, five, one, one, Two, six, six, three,
three, five, Four, Two, one, one]
Trimmed: [Four, five, one, one, Two, six, six, three,
three, five]
Sorted: [Four, Two, five, five, one, one, six, six,
three, three]
Location of six is 7, list.get(7) = six
Case-insensitive sorted: [five, five, Four, one, one,
six, six, three, three, Two]
Location of three is 7, list.get(7) = three
*/
```

Just as when searching and sorting with arrays, if you sort using a `Comparator`, you must `binarySearch()` using the same `Comparator`.

This program also demonstrates the `shuffle()` method in `Collections`, which randomizes the order of a `List`. A `ListIterator` is created at a particular location in the shuffled list, and used to remove the elements from that location until the end of the list.

Making a `Collection` or `Map` Unmodifiable

Often it is convenient to create a read-only version of a `Collection` or `Map`. The `Collections` class does this by accepting the original collection into a method

that hands back a read-only version. There are variations on this method: for `Collections` (if you can't treat a `Collection` as a more specific type), `Lists`, `Sets`, and `Maps`. This example shows the proper way to build read-only versions of each:

```
// collectiontopics/ReadOnly.java
// Using the Collections.unmodifiable methods
import java.util.*;
import onjava.*;

public class ReadOnly {
  static Collection<String> data =
    new ArrayList<>(Countries.names(6));
  public static void main(String[] args) {
    Collection<String> c =
      Collections.unmodifiableCollection(
        new ArrayList<>(data));
    System.out.println(c); // Reading is OK
    //- c.add("one"); // Can't change it

    List<String> a = Collections.unmodifiableList(
        new ArrayList<>(data));
    ListIterator<String> lit = a.listIterator();
    System.out.println(lit.next()); // Reading is OK
    //- lit.add("one"); // Can't change it

    Set<String> s = Collections.unmodifiableSet(
      new HashSet<>(data));
    System.out.println(s); // Reading is OK
    //- s.add("one"); // Can't change it

    // For a SortedSet:
    Set<String> ss =
      Collections.unmodifiableSortedSet(
        new TreeSet<>(data));

    Map<String,String> m =
      Collections.unmodifiableMap(
        new HashMap<>(Countries.capitals(6)));
    System.out.println(m); // Reading is OK
    //- m.put("Ralph", "Howdy!");

    // For a SortedMap:
```

```
    Map<String,String> sm =
      Collections.unmodifiableSortedMap(
        new TreeMap<>(Countries.capitals(6)));
  }
}
/* Output:
[ALGERIA, ANGOLA, BENIN, BOTSWANA, BURKINA FASO,
BURUNDI]
ALGERIA
[BENIN, BOTSWANA, ANGOLA, BURKINA FASO, ALGERIA,
BURUNDI]
{BENIN=Porto-Novo, BOTSWANA=Gaberone, ANGOLA=Luanda,
BURKINA FASO=Ouagadougou, ALGERIA=Algiers,
BURUNDI=Bujumbura}
*/
```

Calling the "unmodifiable" method for a particular type does not cause compile-time checking, but once the transformation has occurred, any calls to methods that modify the contents of a particular collection will produce an `UnsupportedOperationException`.

In each case, you must fill the collection with meaningful data *before* you make it read-only. Once it is loaded, the best approach is to replace the existing reference with the reference that is produced by the "unmodifiable" call. That way, you don't run the risk of accidentally trying to change the contents once you've made it unmodifiable. On the other hand, this tool also lets you keep a modifiable collection as `private` within a class and to return a read-only reference to that collection from a method call. So, you can change it from within the class, but everyone else can only read it.

Synchronizing a `Collection` or `Map`

The `synchronized` keyword is an important part of the subject of *multithreading*, a more complicated topic introduced in the Concurrent Programming chapter. Here, I shall note only that the `Collections` class contains a way to automatically synchronize an entire collection. The syntax is similar to the "unmodifiable" methods:

```
// collectiontopics/Synchronization.java
// Using the Collections.synchronized methods
import java.util.*;

public class Synchronization {
  public static void main(String[] args) {
    Collection<String> c =
      Collections.synchronizedCollection(
        new ArrayList<>());
    List<String> list = Collections
      .synchronizedList(new ArrayList<>());
    Set<String> s = Collections
      .synchronizedSet(new HashSet<>());
    Set<String> ss = Collections
      .synchronizedSortedSet(new TreeSet<>());
    Map<String,String> m = Collections
      .synchronizedMap(new HashMap<>());
    Map<String,String> sm = Collections
      .synchronizedSortedMap(new TreeMap<>());
  }
}
```

It is best to immediately pass the new collection through the appropriate "synchronized" method, as shown above. That way, there's no chance of accidentally exposing the unsynchronized version.

Fail Fast

The Java collections also have a mechanism to prevent more than one process from modifying the contents of a collection. The problem occurs if you're in the middle of iterating through a collection, then some other process steps in and inserts, removes, or changes an object in that collection. Maybe you've already passed that element in the collection, maybe it's ahead of you, maybe the size of the collection shrinks after you call `size()`—there are many scenarios for disaster. The Java collections library uses a *fail-fast* mechanism that looks for any changes to the collection apart from those caused by your process. If it detects that someone else is modifying the collection, it immediately produces a `ConcurrentModificationException`. This is the "fail-fast" aspect—it doesn't try to detect a problem later on using a more complex algorithm.

It's easy to see the fail-fast mechanism in operation by creating an iterator and adding an element to the collection where the iterator points, like this:

```
// collectiontopics/FailFast.java
// Demonstrates the "fail-fast" behavior
import java.util.*;

public class FailFast {
  public static void main(String[] args) {
    Collection<String> c = new ArrayList<>();
    Iterator<String> it = c.iterator();
    c.add("An object");
    try {
      String s = it.next();
    } catch(ConcurrentModificationException e) {
      System.out.println(e);
    }
  }
}
/* Output:
java.util.ConcurrentModificationException
*/
```

The exception comes from attempting to place an element in the collection *after* the iterator is acquired from the collection. The possibility that two parts of the program might modify the same collection produces an uncertain state, so the exception notifies you to change your code—in this case, acquire the iterator *after* you have added all the elements to the collection.

The `ConcurrentHashMap`, `CopyOnWriteArrayList`, and `CopyOnWriteArraySet` use techniques that avoid `ConcurrentModificationExceptions`.

Holding References

The `java.lang.ref` library contains a set of classes that allow greater flexibility in garbage collection. These classes are especially useful when you have large objects that might cause memory exhaustion. There are three classes inherited from the abstract class `Reference`: `SoftReference`, `WeakReference`, and `PhantomReference`.

Each of these provides a different level of indirection for the garbage collector if the object in question is only reachable through one of these `Reference` objects.

If an object is *reachable*, it means somewhere in your program the object can be found. This could mean you have an ordinary reference on the stack that goes right to the object, but you might also have a reference to an object that has a reference to the object in question; there can be many intermediate links. If an object is reachable, the garbage collector cannot release it because it's still in use by your program. If an object isn't reachable, there's no way for your program to use it, so it's safe to garbage collect that object.

You use `Reference` objects to continue holding a reference to that object—to reach that object—but also to allow the garbage collector to release that object. Thus, you have a way to use the object, but if memory exhaustion is imminent, you allow that object to be released.

You accomplish this by using a `Reference` object as an intermediary (a *proxy*) between you and the ordinary reference. In addition, there must be no ordinary references to the object (ones not wrapped inside `Reference` objects). If the garbage collector discovers that an object is reachable through an ordinary reference, it will not release that object.

In the order of `SoftReference`, `WeakReference`, and `PhantomReference`, each one is "weaker" than the last and corresponds to a different level of reachability. Soft references are for implementing memory-sensitive caches. Weak references are for implementing "canonicalizing mappings"—where instances of objects can be simultaneously used in multiple places in a program, to save storage—that do not prevent their keys (or values) from being reclaimed. Phantom references are for scheduling pre-mortem cleanup actions in a more flexible way than is possible with the Java finalization mechanism.

With `SoftReferences` and `WeakReferences`, you have a choice about whether to place them on a `ReferenceQueue` (the device used for pre-mortem cleanup actions), but a `PhantomReference` can only be built on a `ReferenceQueue`. Here's a simple demonstration:

```
// collectiontopics/References.java
// Demonstrates Reference objects
import java.lang.ref.*;
import java.util.*;

class VeryBig {
  private static final int SIZE = 10000;
  private long[] la = new long[SIZE];
  private String ident;
  VeryBig(String id) { ident = id; }
  @Override public String toString() { return ident; }
  @SuppressWarnings("deprecation")
  @Override protected void finalize() {
    System.out.println("Finalizing " + ident);
  }
}

public class References {
  private static ReferenceQueue<VeryBig> rq =
    new ReferenceQueue<>();
  public static void checkQueue() {
    Reference<? extends VeryBig> inq = rq.poll();
    if(inq != null)
      System.out.println("In queue: " + inq.get());
  }
  public static void main(String[] args) {
    int size = 10;
    // Or, choose size via the command line:
    if(args.length > 0)
      size = Integer.valueOf(args[0]);
    LinkedList<SoftReference<VeryBig>> sa =
      new LinkedList<>();
    for(int i = 0; i < size; i++) {
      sa.add(new SoftReference<>(
        new VeryBig("Soft " + i), rq));
      System.out.println(
        "Just created: " + sa.getLast());
      checkQueue();
    }
    LinkedList<WeakReference<VeryBig>> wa =
      new LinkedList<>();
    for(int i = 0; i < size; i++) {
```

```
      wa.add(new WeakReference<>(
        new VeryBig("Weak " + i), rq));
      System.out.println(
        "Just created: " + wa.getLast());
      checkQueue();
    }
    SoftReference<VeryBig> s =
      new SoftReference<>(new VeryBig("Soft"));
    WeakReference<VeryBig> w =
      new WeakReference<>(new VeryBig("Weak"));
    System.gc();
    LinkedList<PhantomReference<VeryBig>> pa =
      new LinkedList<>();
    for(int i = 0; i < size; i++) {
      pa.add(new PhantomReference<>(
        new VeryBig("Phantom " + i), rq));
      System.out.println(
        "Just created: " + pa.getLast());
      checkQueue();
    }
  }
}
/* Output: (First and Last 10 Lines)
Just created: java.lang.ref.SoftReference@19e0bfd
Just created: java.lang.ref.SoftReference@139a55
Just created: java.lang.ref.SoftReference@1db9742
Just created: java.lang.ref.SoftReference@106d69c
Just created: java.lang.ref.SoftReference@52e922
Just created: java.lang.ref.SoftReference@25154f
Just created: java.lang.ref.SoftReference@10dea4e
Just created: java.lang.ref.SoftReference@647e05
Just created: java.lang.ref.SoftReference@1909752
Just created: java.lang.ref.SoftReference@1f96302
...________...________...________...________...
Just created: java.lang.ref.PhantomReference@16f6e28
In queue: null
Just created: java.lang.ref.PhantomReference@15fbaa4
In queue: null
Just created: java.lang.ref.PhantomReference@1ee12a7
In queue: null
Just created: java.lang.ref.PhantomReference@10bedb4
In queue: null
```

```
Just created: java.lang.ref.PhantomReference@103dbd3
In queue: null
*/
```

When you run this program (redirect the output into a text file to view the output in pages), you'll see that the objects are garbage collected, even though you can still access them through the `Reference` object (to get the actual object reference, use `get()`). You'll also see that the `ReferenceQueue` always produces a `Reference` containing a `null` object. To use this, inherit from a particular `Reference` class and add more useful methods to the new class.

The `WeakHashMap`

The collections library has a special `Map` to hold weak references: the `WeakHashMap`. This class makes it easier to create canonicalized mappings. In such a mapping, you save storage by creating only one instance of a particular value. When the program needs that value, it looks up the existing object in the mapping and uses that (rather than creating one from scratch). The mapping can make the values as part of its initialization, but it's more likely that the values are made on demand.

This is a storage-saving technique so it's very convenient that the `WeakHashMap` allows the garbage collector to automatically clean up the keys and values. You don't do anything special to the keys and values you place in the `WeakHashMap`; these are automatically wrapped in `WeakReferences` by the map. The trigger to allow cleanup is that the key is no longer in use, as demonstrated here:

```
// collectiontopics/CanonicalMapping.java
// Demonstrates WeakHashMap
import java.util.*;

class Element {
  private String ident;
  Element(String id) { ident = id; }
  @Override public String toString() { return ident; }
  @Override public int hashCode() {
    return Objects.hashCode(ident);
  }
  @Override public boolean equals(Object r) {
    return r instanceof Element &&
```

```
      Objects.equals(ident, ((Element)r).ident);
  }
  @SuppressWarnings("deprecation")
  @Override protected void finalize() {
    System.out.println("Finalizing " +
      getClass().getSimpleName() + " " + ident);
  }
}

class Key extends Element {
  Key(String id) { super(id); }
}

class Value extends Element {
  Value(String id) { super(id); }
}

public class CanonicalMapping {
  public static void main(String[] args) {
    int size = 1000;
    // Or, choose size via the command line:
    if(args.length > 0)
      size = Integer.valueOf(args[0]);
    Key[] keys = new Key[size];
    WeakHashMap<Key,Value> map =
      new WeakHashMap<>();
    for(int i = 0; i < size; i++) {
      Key k = new Key(Integer.toString(i));
      Value v = new Value(Integer.toString(i));
      if(i % 3 == 0)
        keys[i] = k; // Save as "real" references
      map.put(k, v);
    }
    System.gc();
  }
}
```

The `Key` class must have a `hashCode()` and an `equals()` because it is used as a key in a hashed data structure. The subject of `hashCode()` was described in the Appendix: Understanding `equals()` and `hashCode()`.

Run the program and you'll see the garbage collector skip every third key. An

ordinary reference to that key has also been placed in the `keys` array, and thus those objects cannot be garbage collected.

Java 1.0/1.1 Collections

Unfortunately, much code was written using the Java 1.0/1.1 collections, and even new code is sometimes written using these classes. Never use the old collections when writing new code. The old collections were limited, so there's not that much to say about them. They are anachronistic so I try to refrain from overemphasizing some of their hideous design decisions.

`Vector` & `Enumeration`

The only self-expanding sequence in Java 1.0/1.1 was the `Vector`, so it saw a lot of use. Its flaws are too numerous to describe here (see the 1st edition of *Thinking in Java*). Basically, you can think of it as an `ArrayList` with long, awkward method names. In the revised Java collection library, `Vector` was adapted so it could work as a `Collection` and a `List`. This turns out to be a bit perverse, as it can confuse some people into thinking that `Vector` has gotten better, when it is actually included only to support older Java code.

The Java 1.0/1.1 version of the iterator chose to invent a new name, "enumeration," instead of using a term that everyone was already familiar with ("iterator"). The `Enumeration` interface is smaller than `Iterator`, with only two methods, and it uses longer method names: `boolean hasMoreElements()` produces `true` if this enumeration contains more elements, and `Object nextElement()` returns the next element of this enumeration if there are any more (otherwise it throws an exception).

`Enumeration` is only an interface, not an implementation, and even new libraries sometimes still use the old `Enumeration`, which is unfortunate but generally harmless. Always use `Iterator` when you can in your own code, but be prepared for libraries that hand you an `Enumeration`.

In addition, you can produce an `Enumeration` for any `Collection` by using the `Collections.enumeration()` method, as seen in this example:

```
// collectiontopics/Enumerations.java
// Java 1.0/1.1 Vector and Enumeration
import java.util.*;
import onjava.*;

public class Enumerations {
  public static void main(String[] args) {
    Vector<String> v =
      new Vector<>(Countries.names(10));
    Enumeration<String> e = v.elements();
    while(e.hasMoreElements())
      System.out.print(e.nextElement() + ", ");
    // Produce an Enumeration from a Collection:
    e = Collections.enumeration(new ArrayList<>());
  }
}
/* Output:
ALGERIA, ANGOLA, BENIN, BOTSWANA, BURKINA FASO,
BURUNDI, CAMEROON, CAPE VERDE, CENTRAL AFRICAN
REPUBLIC, CHAD,
*/
```

To produce an `Enumeration`, you call `elements()`, then you can use it to perform a forward iteration.

The last line creates an `ArrayList` and uses `enumeration()` to adapt an `Enumeration` from the `ArrayList Iterator`. Thus, if you have old code that wants an `Enumeration`, you can still use the new collections.

`Hashtable`

As you've seen in the performance comparison in this appendix, the basic `Hashtable` is very similar to the `HashMap`, even down to the method names. There's no reason to use `Hashtable` instead of `HashMap` in new code.

`Stack`

The concept of the stack was introduced earlier, with the `LinkedList`. What's rather odd about the Java 1.0/1.1 `Stack` is that instead of using a `Vector` with composition,

`Stack` is *inherited* from `Vector`. So it has all characteristics and behaviors of a `Vector` plus some extra `Stack` behaviors. It's difficult to know whether the designers consciously thought this was a helpful way of doing things, or whether it was just naïve; in any event it was clearly not reviewed before it was rushed into distribution, so this bad design is *still* hanging around (but don't use it).

Here's a simple demonstration of `Stack` that pushes each `String` representation of an `enum`. It also shows how you can just as easily use a `LinkedList` as a stack, or the `Stack` class created in the Collections chapter:

```
// collectiontopics/Stacks.java
// Demonstration of Stack Class
import java.util.*;

enum Month { JANUARY, FEBRUARY, MARCH, APRIL,
  MAY, JUNE, JULY, AUGUST, SEPTEMBER,
  OCTOBER, NOVEMBER }

public class Stacks {
  public static void main(String[] args) {
    Stack<String> stack = new Stack<>();
    for(Month m : Month.values())
      stack.push(m.toString());
    System.out.println("stack = " + stack);
    // Treating a stack as a Vector:
    stack.addElement("The last line");
    System.out.println(
      "element 5 = " + stack.elementAt(5));
    System.out.println("popping elements:");
    while(!stack.empty())
      System.out.print(stack.pop() + " ");

    // Using a LinkedList as a Stack:
    LinkedList<String> lstack = new LinkedList<>();
    for(Month m : Month.values())
      lstack.addFirst(m.toString());
    System.out.println("lstack = " + lstack);
    while(!lstack.isEmpty())
      System.out.print(lstack.removeFirst() + " ");

    // Using the Stack class from
    // the Collections Chapter:
```

```
    onjava.Stack<String> stack2 =
      new onjava.Stack<>();
    for(Month m : Month.values())
      stack2.push(m.toString());
    System.out.println("stack2 = " + stack2);
    while(!stack2.isEmpty())
      System.out.print(stack2.pop() + " ");

  }
}
/* Output:
stack = [JANUARY, FEBRUARY, MARCH, APRIL, MAY, JUNE,
JULY, AUGUST, SEPTEMBER, OCTOBER, NOVEMBER]
element 5 = JUNE
popping elements:
The last line NOVEMBER OCTOBER SEPTEMBER AUGUST JULY
JUNE MAY APRIL MARCH FEBRUARY JANUARY lstack =
[NOVEMBER, OCTOBER, SEPTEMBER, AUGUST, JULY, JUNE, MAY,
APRIL, MARCH, FEBRUARY, JANUARY]
NOVEMBER OCTOBER SEPTEMBER AUGUST JULY JUNE MAY APRIL
MARCH FEBRUARY JANUARY stack2 = [NOVEMBER, OCTOBER,
SEPTEMBER, AUGUST, JULY, JUNE, MAY, APRIL, MARCH,
FEBRUARY, JANUARY]
NOVEMBER OCTOBER SEPTEMBER AUGUST JULY JUNE MAY APRIL
MARCH FEBRUARY JANUARY
*/
```

A `String` representation is generated from the `Month` constants, inserted into the `Stack` with `push()`, and later fetched from the top of the stack with a `pop()`. To make a point, `Vector` operations are also performed on the `Stack` object. This is possible because, by virtue of inheritance, a `Stack` *is* a `Vector`. Thus, all operations that can be performed on a `Vector` can also be performed on a `Stack`, such as `elementAt()`.

As mentioned earlier, use a `LinkedList` when you want stack behavior, or the `onjava.Stack` class created from the `LinkedList` class.

BitSet

A `BitSet` efficiently stores a set of on-off data. It's efficient only from the standpoint of size; if you're looking for efficient access, it is slightly slower than using a native array.

In addition, the minimum size of the `BitSet` is that of a `long`: 64 bits. This implies that if you're storing anything smaller, like 8 bits, a `BitSet` is wasteful; you're better off creating your own class, or just an array, to hold your flags if size is an issue. (This will only be the case if you're creating many objects containing lists of on-off information, and should only be decided based on profiling and other metrics. If you make this decision because you just think something is too big, you end up creating needless complexity and wasting a lot of time.)

A normal collection expands as you add more elements, and the `BitSet` does this as well. The following example shows how the `BitSet` works:

```
// collectiontopics/Bits.java
// Demonstration of BitSet
import java.util.*;

public class Bits {
  public static void printBitSet(BitSet b) {
    System.out.println("bits: " + b);
    StringBuilder bbits = new StringBuilder();
    for(int j = 0; j < b.size() ; j++)
      bbits.append(b.get(j) ? "1" : "0");
    System.out.println("bit pattern: " + bbits);
  }
  public static void main(String[] args) {
    Random rand = new Random(47);
    // Take the LSB of nextInt():
    byte bt = (byte)rand.nextInt();
    BitSet bb = new BitSet();
    for(int i = 7; i >= 0; i--)
      if(((1 << i) &  bt) != 0)
        bb.set(i);
      else
        bb.clear(i);
    System.out.println("byte value: " + bt);
    printBitSet(bb);

    short st = (short)rand.nextInt();
    BitSet bs = new BitSet();
    for(int i = 15; i >= 0; i--)
      if(((1 << i) &  st) != 0)
        bs.set(i);
```

```
      else
        bs.clear(i);
    System.out.println("short value: " + st);
    printBitSet(bs);

    int it = rand.nextInt();
    BitSet bi = new BitSet();
    for(int i = 31; i >= 0; i--)
      if(((1 << i) &  it) != 0)
        bi.set(i);
      else
        bi.clear(i);
    System.out.println("int value: " + it);
    printBitSet(bi);

    // Test bitsets >= 64 bits:
    BitSet b127 = new BitSet();
    b127.set(127);
    System.out.println("set bit 127: " + b127);
    BitSet b255 = new BitSet(65);
    b255.set(255);
    System.out.println("set bit 255: " + b255);
    BitSet b1023 = new BitSet(512);
    b1023.set(1023);
    b1023.set(1024);
    System.out.println("set bit 1023: " + b1023);
  }
}
/* Output:
byte value: -107
bits: {0, 2, 4, 7}
bit pattern: 10101001000000000000000000000000000000000000
0000000000000000000000
short value: 1302
bits: {1, 2, 4, 8, 10}
bit pattern: 01101000101000000000000000000000000000000000
0000000000000000000000
int value: -2014573909
bits: {0, 1, 3, 5, 7, 9, 11, 18, 19, 21, 22, 23, 24,
25, 26, 31}
bit pattern: 11010101010100000011011111100001000000000000
0000000000000000000000
```

```
set bit 127: {127}
set bit 255: {255}
set bit 1023: {1023, 1024}
*/
```

The random number generator creates a random `byte`, `short`, and `int`. Each is transformed into a corresponding bit pattern in a `BitSet`. This works fine because a `BitSet` is 64 bits, so none of these cause it to increase in size. Then larger `BitSets` are created. Notice that the `BitSet` is expanded as necessary.

An `EnumSet` (see the Enumerations chapter) is usually a better choice than a `BitSet` for a fixed set of flags that you can name, because the `EnumSet` lets you manipulate the names rather than numerical bit locations, and thus reduces errors. `EnumSet` also prevents you from accidentally adding new flag locations, which could cause some serious, difficult-to-find bugs. The only reasons to use `BitSet` instead of `EnumSet` is if you don't know how many flags you need until run time, or if it is unreasonable to assign names to the flags, or you need one of the special operations in `BitSet` (see the JDK documentation for `BitSet` and `EnumSet`).

Summary

Collections are arguably the most often-used tools in a programming language. Some languages (Python, for example) even include the fundamental collection components (lists, maps and sets) as built-ins.

As you saw in the Collections chapter, it's possible to do some very useful things with collections, without much effort. However, at some point you're forced to know more about collections to use them properly—in particular, you must know enough about hashing operations to write your own `hashCode()` method (and you must know when it is necessary), and you must know enough about various collection implementations to choose the appropriate one for your needs. This appendix covered these concepts and discussed additional useful details about the collection library. You're now reasonably well prepared to use the Java collections in your everyday programming tasks.

The design of a collections library is difficult (this is true of most library design problems). In C++, the collection classes covered the bases with many different

classes. This was better than what was available prior to the C++ collection classes (nothing), but it didn't translate well into Java. At the other extreme, I've seen a collections library that consists of a single class, "collection," which acts like both a linear sequence and an associative array at the same time. The Java collection library attempts to strike a balance between power and complexity. The result can seem a bit odd in places. Unlike some of the decisions made in the early Java libraries, these oddities were not accidents, but carefully considered decisions based on trade-offs in complexity.

Appendix: Low-Level Concurrency

> Although it is never advisable to write low-level Java concurrency code yourself, it is often helpful to understand something of how it works.

The Concurrent Programming chapter introduced the concepts of concurrency at a high level, including more recent and safer constructs (parallel `Streams` and `CompletableFutures`) for concurrent Java programming. This appendix introduces low-level concurrency concepts in Java, so you have a grasp of such code when you read it. You will also gain further insight into the general issues of concurrency.

Low-level concurrency concepts were a big part of concurrent programming in early versions of Java. We'll look at the complexity around those techniques and why you should avoid them. The Concurrent Programming chapter demonstrates the improved techniques afforded by more recent versions of Java (especially Java 8) that make concurrency, if not easy, then much easier.

What is a Thread?

Concurrency partitions a program into separate, independently running tasks. Each task is driven by a *thread of execution*, which we typically shorten to just *thread*. A *thread* is a single sequential flow of control within an operating-system process. A process can thus have multiple concurrently-executing tasks, but you program as if each task has a processor to itself. The threading model is a programming convenience to simplify juggling multiple tasks within a single program. The operating system allocates time from the processor(s) among all of your threads.

The core mechanism of Java concurrency is the `Thread` class. In initial versions of the language, `Threads` were intended to be created and managed directly by the programmer. As the language evolved and people discovered better approaches,

intermediate mechanisms—in particular the `Executor`—were added to remove the mental overhead (and mistakes) of managing threads yourself. Eventually, even better mechanisms than `Executor` were developed, as shown in the Concurrent Programming chapter.

A `Thread` is a software construct that connects a task to a processor. Although creating and using `Threads` seems similar to any other class, under the covers they are very different. When you create a `Thread`, the JVM allocates a significant chunk of memory in a special area reserved just for `Threads`, to provide everything necessary to run a task:

- A program counter indicating the next JVM bytecode instruction to execute.
- A stack to support the execution of Java code, containing information about the methods this thread has called to reach the current point of execution. It also contains all local variables for each method being executed (including primitives and references to heap objects). This stack is typically between 64K and 1M *per thread.*[201]
- A second stack for native code.
- Storage for *thread-local variables.*
- Housekeeping state variables to control the thread.

All code including `main()` runs inside some thread. Whenever a method is called, the current program counter is pushed onto that thread's stack, then the stack pointer is moved down enough to create a *stack frame* with all the storage for that method's local variables, arguments, and the return value. All local primitives go directly on the stack. While any *references* to objects created in the method (or used by the method) live in the stack frame, the objects themselves go on the heap. There's only one heap, shared between all threads in the program.

On top of all this, the `Thread` must be registered with the operating system (OS), so it can actually be connected to a processor at some point. This is managed for you as part of the `Thread` construction process. Java uses the mechanisms in the underlying OS to manage the execution of threads.

[201] On some platforms, notably Windows, the default value can be remarkably hard to discover. You can adjust the stack size with the `-Xss` flag.

The Optimal Number of Threads

If you look at the examples in Concurrent Programming that use *CachedThreadPool*, you'll see that the `ExecutorService` allocates a thread for each task we submit. However, the parallel `Stream` in `CountingStream.java` only allocated 8 threads (*worker*s 1-7 and a thread for `main()`, which it cleverly used for an additional parallel stream). If you try increasing the upper bound of the `range()`, you'll see no additional threads are created. Why is this?

We can discover the number of processors on the current machine:

```
// lowlevel/NumberOfProcessors.java

public class NumberOfProcessors {
  public static void main(String[] args) {
    System.out.println(
      Runtime.getRuntime().availableProcessors());
  }
}
/* Output:
8
*/
```

On *my* machine (using an Intel Core i7), I have four cores, each presenting two *hyperthreads* (a hardware trick to produce very rapid context switching on a single processor, which can in some situations make it look like two hardware threads). Although this is a common configuration on "recent" machines (at the time of this writing), you might see a different result—along with an equivalent number of default threads in `CountingStream.java`.

Your operating system might have a way to discover more information about your processor. For example, on Windows 10, press the "Start" button, type "Task Manager" and the "Enter" key. Click "More Details." Choose the "Performance" tab, and you'll see all kinds of information about your hardware, including "Cores" and "Logical Processors."

It turns out that the "generic" optimal number of threads is the number of processors available (this might not be true for specific problems). This comes from the cost of *context switching* between Java threads: storing the current state of the thread being suspended and retrieving the other thread's current state to continue execution from

where it entered suspension. For eight processors and eight (compute-intensive) Java threads, the JVM never has to switch contexts when running those eight tasks. For fewer tasks than the number of processors, it doesn't help to allocate more threads.

Intel hyperthreading, which defines the number of "logical processors," does not increase computational capacity—the feature maintains extra thread *contexts* at a hardware level, which speeds context switches, and this helps in the responsiveness of user interfaces, for example. For compute-intensive tasks, consider matching the number of threads to the number of physical cores (not hyperthreads). Although Java considers each hyperthread to be a processor, this appears to be a mistake influenced by Intel's over-marketing of hyperthreads. Despite this, for programming simplicity I'll just allow the JVM to decide the default number of threads. You will want to experiment with your production applications. This doesn't mean that matching the number of threads to the number of processors is appropriate for all problems; on the contrary it's primarily used only for compute-intensive solutions.

How Many Threads Can I Create?

The largest part of a `Thread` object is the Java stack for executing methods. Discovering the size of a `Thread` object varies between operating systems. This program tests it by creating `Thread` objects until the JVM runs out of memory:

```
// lowlevel/ThreadSize.java
// {ExcludeFromGradle} Takes a long time or hangs
import java.util.concurrent.*;
import onjava.Nap;

public class ThreadSize {
  static class Dummy extends Thread {
    @Override public void run() { new Nap(1); }
  }
  public static void main(String[] args) {
    ExecutorService exec =
      Executors.newCachedThreadPool();
    int count = 0;
    try {
      while(true) {
        exec.execute(new Dummy());
        count++;
```

```
      }
    } catch(Error e) {
      System.out.println(
        e.getClass().getSimpleName() + ": " + count);
      System.exit(0);
    } finally {
      exec.shutdown();
    }
  }
}
```

A *CachedThreadPool* continues creating `Threads` as long as you keep passing it tasks. Passing a `Dummy` object to `execute()` starts the task, allocating a new `Thread` if one isn't available. The `pause()` size must be large enough that tasks don't start finishing (thus freeing up existing `Threads` for new tasks). As long as tasks keep coming in and not finishing, a *CachedThreadPool* will eventually run out of memory.

I wasn't always able to cause out-of-memory errors on every machine I tried. On one machine, I see this result:

```
>java ThreadSize
OutOfMemoryError: 2816
```

We can reduce each `Thread`'s stack size using the `-Xss` flag. The minimum stack size allowed is 64k:

```
>java -Xss64K ThreadSize
OutOfMemoryError: 4952
```

If we increase the stack size to 2 Megabytes, we can allocate far fewer threads:

```
>java -Xss2M ThreadSize
OutOfMemoryError: 722
```

The default Windows stack size is 320K, which we can verify by noting that it gives us about the same number as when we don't set the stack size at all:

```
>java -Xss320K ThreadSize
OutOfMemoryError: 2816
```

You can also increase the JVM's maximum memory allocation using the `-Xmx` flag:

```
>java -Xss64K -Xmx5M ThreadSize
OutOfMemoryError: 5703
```

Notice that the operating system may also apply limits to the number of threads allowed.

So the answer to the question "how many threads can I have" is "a few thousand." If you find yourself allocating thousands of threads, however, you might want to rethink your approach; the appropriate question is "how many threads do I *need*?"

The `WorkStealingPool`

This is an `ExecutorService` that automatically creates a thread pool using all available processors (as reported by the JVM).

```
// lowlevel/WorkStealingPool.java
import java.util.stream.*;
import java.util.concurrent.*;

class ShowThread implements Runnable {
  @Override public void run() {
    System.out.println(
      Thread.currentThread().getName());
  }
}

public class WorkStealingPool {
  public static void main(String[] args)
    throws InterruptedException {
    System.out.println(
      Runtime.getRuntime().availableProcessors());
    ExecutorService exec =
      Executors.newWorkStealingPool();
    IntStream.range(0, 10)
      .mapToObj(n -> new ShowThread())
      .forEach(exec::execute);
    exec.awaitTermination(1, TimeUnit.SECONDS);
  }
}
/* Output:
8
```

```
ForkJoinPool-1-worker-1
ForkJoinPool-1-worker-2
ForkJoinPool-1-worker-1
ForkJoinPool-1-worker-3
ForkJoinPool-1-worker-4
ForkJoinPool-1-worker-4
ForkJoinPool-1-worker-2
ForkJoinPool-1-worker-3
ForkJoinPool-1-worker-5
ForkJoinPool-1-worker-1
*/
```

The Work Stealing[202] algorithm allows threads that have run out of work items in their input queue to "steal" work items from other queues. The goal is to distribute work items among processors, thus making maximal use of all available processors for compute-intensive tasks. It is also used in Java's fork/join framework.

Catching Exceptions

This might surprise you:

```
// lowlevel/SwallowedException.java
import java.util.concurrent.*;

public class SwallowedException {
  public static void main(String[] args)
    throws InterruptedException {
    ExecutorService exec =
      Executors.newSingleThreadExecutor();
    exec.submit(() -> {
      throw new RuntimeException();
    });
    exec.shutdown();
  }
}
```

This program outputs ... nothing (if, however, you replace `submit()` with `execute()`, you *will* see the exception). This points out that exceptions thrown inside threads are tricky, and require special attention.

[202] https://en.wikipedia.org/wiki/Work_stealing

You can't catch an exception that has escaped from a thread. Once an exception gets outside of a task's `run()` method, it will propagate out to the console unless you take special steps to capture such errant exceptions.

Here's a task that throws an exception which propagates outside of its `run()` method, and a `main()` that shows what happens when you run it:

```
// lowlevel/ExceptionThread.java
// {ThrowsException}
import java.util.concurrent.*;

public class ExceptionThread implements Runnable {
  @Override public void run() {
    throw new RuntimeException();
  }
  public static void main(String[] args) {
    ExecutorService es =
      Executors.newCachedThreadPool();
    es.execute(new ExceptionThread());
    es.shutdown();
  }
}
/* Output:
___[ Error Output ]___
Exception in thread "pool-1-thread-1"
java.lang.RuntimeException
        at ExceptionThread.run(ExceptionThread.java:7)
        at java.util.concurrent.ThreadPoolExecutor.runW
orker(ThreadPoolExecutor.java:1142)
        at java.util.concurrent.ThreadPoolExecutor$Work
er.run(ThreadPoolExecutor.java:617)
        at java.lang.Thread.run(Thread.java:745)
*/
```

The output is (after trimming some qualifiers to fit):

```
Exception in thread "pool-1-thread-1" RuntimeException
  at ExceptionThread.run(ExceptionThread.java:9)
  at ThreadPoolExecutor.runWorker(...)
  at ThreadPoolExecutor$Worker.run(...)
  at java.lang.Thread.run(Thread.java:745)
```

Encompassing the body of main within a `try-catch` block is unsuccessful:

```
// lowlevel/NaiveExceptionHandling.java
// {ThrowsException}
import java.util.concurrent.*;

public class NaiveExceptionHandling {
  public static void main(String[] args) {
    ExecutorService es =
      Executors.newCachedThreadPool();
    try {
      es.execute(new ExceptionThread());
    } catch(RuntimeException ue) {
      // This statement will NOT execute!
      System.out.println("Exception was handled!");
    } finally {
      es.shutdown();
    }
  }
}
/* Output:
___[ Error Output ]___
Exception in thread "pool-1-thread-1"
java.lang.RuntimeException
        at ExceptionThread.run(ExceptionThread.java:7)
        at java.util.concurrent.ThreadPoolExecutor.runW
orker(ThreadPoolExecutor.java:1142)
        at java.util.concurrent.ThreadPoolExecutor$Work
er.run(ThreadPoolExecutor.java:617)
        at java.lang.Thread.run(Thread.java:745)
*/
```

This produces the same result as the previous example: an uncaught exception. To solve the problem, we must change the way the `Executor` produces threads.

`Thread.UncaughtExceptionHandler` is an interface for attaching an exception handler to each `Thread` object. `Thread.UncaughtExceptionHandler.` uncaugh-

`tException()` is automatically called when that thread is about to die from an uncaught exception. To use it, we create a new type of `ThreadFactory` which attaches a new `Thread.UncaughtExceptionHandler` to each new `Thread` object it creates. We pass that factory to the `Executors` method that creates a new `ExecutorService`:

```
// lowlevel/CaptureUncaughtException.java
import java.util.concurrent.*;

class ExceptionThread2 implements Runnable {
  @Override public void run() {
    Thread t = Thread.currentThread();
    System.out.println("run() by " + t.getName());
    System.out.println(
      "eh = " + t.getUncaughtExceptionHandler());
    throw new RuntimeException();
  }
}

class MyUncaughtExceptionHandler implements
Thread.UncaughtExceptionHandler {
  @Override
  public void uncaughtException(Thread t, Throwable e) {
    System.out.println("caught " + e);
  }
}

class HandlerThreadFactory implements ThreadFactory {
  @Override public Thread newThread(Runnable r) {
    System.out.println(this + " creating new Thread");
    Thread t = new Thread(r);
    System.out.println("created " + t);
    t.setUncaughtExceptionHandler(
      new MyUncaughtExceptionHandler());
    System.out.println(
      "eh = " + t.getUncaughtExceptionHandler());
    return t;
  }
}

public class CaptureUncaughtException {
  public static void main(String[] args) {
```

```
    ExecutorService exec =
      Executors.newCachedThreadPool(
        new HandlerThreadFactory());
    exec.execute(new ExceptionThread2());
    exec.shutdown();
  }
}
/* Output:
HandlerThreadFactory@106d69c creating new Thread
created Thread[Thread-0,5,main]
eh = MyUncaughtExceptionHandler@52e922
run() by Thread-0
eh = MyUncaughtExceptionHandler@52e922
caught java.lang.RuntimeException
*/
```

Additional tracing verifies that the threads created by the factory get the new `UncaughtExceptionHandler`. The uncaught exceptions are now captured by `uncaughtException`.

The above example sets the handler on a case-by-case basis. If you know that you're going to use the same exception handler everywhere, an even simpler approach is to set the *default* uncaught exception handler, which sets a `static` field inside the `Thread` class:

```
// lowlevel/SettingDefaultHandler.java
import java.util.concurrent.*;

public class SettingDefaultHandler {
  public static void main(String[] args) {
    Thread.setDefaultUncaughtExceptionHandler(
      new MyUncaughtExceptionHandler());
    ExecutorService es =
      Executors.newCachedThreadPool();
    es.execute(new ExceptionThread());
    es.shutdown();
  }
}
/* Output:
caught java.lang.RuntimeException
*/
```

This handler is only called if there is no per-thread uncaught exception handler. The system checks for a per-thread version, and if it doesn't find one it checks to see if the thread group specializes its `uncaughtException()` method; if not, it calls the `defaultUncaughtExceptionHandler`.

Compare this with the improved approach seen with `CompletableFutures`.

Sharing Resources

You can think of a single-threaded program as one lonely entity moving around through your problem space and doing one thing at a time. Because there's only one entity, you never think about the problem of two entities trying to use the same resource at the same time: problems such as two people trying to park in the same space, walk through a door at the same time, or even talk at the same time.

With concurrency, things aren't lonely anymore, but now two or more tasks might interfere with each other. If you don't prevent such a collision, you'll have two tasks trying to access the same bank account at the same time, print to the same printer, adjust the same valve, and so on.

Resource Contention

When you start a task to perform some work, the results of that work can be captured in two different ways: through *side effects* or via a return value.

Programmatically, side effects can seem easier: you just use your results to manipulate something in the environment. For example, your task might perform some calculation, then directly write its results into a collection.

The problem with this approach is that the collection is typically a *shared resource*. When more than one task is running, any task might read or write a shared resource at the same time. This reveals the problem of *resource contention*, one of the main pitfalls when working with tasks.

You don't think about resource contention in a single-threaded system because you're never doing more than one thing at a time. When you have multiple tasks, you must always guard against resource contention.

One approach to the problem is to use a collection that copes with resource contention. If more than one task tries to write to such a collection at the same time, such collections compensate for the problem. You'll find a number of classes in the Java concurrency libraries that attempt to solve resource contention problems; in this appendix you'll see a handful of these but the coverage is not comprehensive.

Consider the following example, where one task generates even numbers and other tasks consume those numbers. Here, the only job of the consumer tasks is to check the validity of the even numbers.

We shall define `EvenChecker`, the consumer task, to make it reusable in the subsequent examples. To decouple `EvenChecker` from our various experimental generators, we'll first create an abstract class called `IntGenerator`, which contains the minimum necessary methods that `EvenChecker` must know about: it has a `next()` method and it can be canceled.

```
// lowlevel/IntGenerator.java
import java.util.concurrent.atomic.AtomicBoolean;

public abstract class IntGenerator {
  private AtomicBoolean canceled =
    new AtomicBoolean();
  public abstract int next();
  public void cancel() { canceled.set(true); }
  public boolean isCanceled() {
    return canceled.get();
  }
}
```

`cancel()` changes the state of the `canceled` flag and `isCanceled()` tells whether the flag is set. Because the `canceled` flag is `AtomicBoolean`, it is *atomic*, which means simple operations like assignment and value return happen without the possibility of interruption, so you can't see the field in an intermediate state in the midst of those simple operations. You'll learn more about atomicity and the `Atomic` classes later in this appendix.

Any `IntGenerator` can be tested with the following `EvenChecker` class:

```
// lowlevel/EvenChecker.java
import java.util.*;
import java.util.stream.*;
import java.util.concurrent.*;
import onjava.TimedAbort;

public class EvenChecker implements Runnable {
  private IntGenerator generator;
  private final int id;
  public EvenChecker(IntGenerator generator, int id) {
    this.generator = generator;
    this.id = id;
  }
  @Override public void run() {
    while(!generator.isCanceled()) {
      int val = generator.next();
      if(val % 2 != 0) {
        System.out.println(val + " not even!");
        generator.cancel(); // Cancels all EvenCheckers
      }
    }
  }
  // Test any IntGenerator:
  public static void test(IntGenerator gp, int count) {
    List<CompletableFuture<Void>> checkers =
      IntStream.range(0, count)
        .mapToObj(i -> new EvenChecker(gp, i))
        .map(CompletableFuture::runAsync)
        .collect(Collectors.toList());
    checkers.forEach(CompletableFuture::join);
  }
  // Default value for count:
  public static void test(IntGenerator gp) {
    new TimedAbort(4, "No odd numbers discovered");
    test(gp, 10);
  }
}
```

The `test()` method starts multiple `EvenCheckers` that access the same `IntGenerator`. `EvenChecker` tasks constantly read and test the values from their associated `IntGenerator`. If the `IntGenerator` causes a failure, `test()` reports it and returns.

All `EvenChecker` tasks that depend on the `IntGenerator` object check it to see whether it's been canceled. If `generator.isCanceled()` is `true`, `run()` returns. Any `EvenChecker` task can call `cancel()` on the `IntGenerator`, which causes all other `EvenCheckers` using that `IntGenerator` to gracefully shut down.

In this design, tasks sharing a common resource (the `IntGenerator`) watch that resource for the signal to terminate. This eliminates the so-called race condition, where two or more tasks race to respond to a condition and thus collide or otherwise produce inconsistent results.

You must carefully think about and protect against all possible ways a concurrent system can fail. For example, a task cannot depend on another task, because task shutdown order is not guaranteed. Here, by making tasks depend on a non-task object, we eliminate the potential race condition.

Normally, we presume that `test()` eventually fails because the `EvenChecker` tasks are able to access the information in `IntGenerator` while it's in an "incorrect" state. However, it might not detect the problem until the `IntGenerator` has completed many cycles, depending on the particulars of your operating system and other implementation details. To ensure that this book's automated build doesn't get stuck, we use `TimedAbort`, defined here:

```
// onjava/TimedAbort.java
// Terminate a program after t seconds
package onjava;
import java.util.concurrent.*;

public class TimedAbort {
  private volatile boolean restart = true;
  public TimedAbort(double t, String msg) {
    CompletableFuture.runAsync(() -> {
      try {
        while(restart) {
          restart = false;
          TimeUnit.MILLISECONDS
            .sleep((int)(1000 * t));
        }
      } catch(InterruptedException e) {
        throw new RuntimeException(e);
      }
      System.out.println(msg);
```

```
      System.exit(0);
    });
  }
  public TimedAbort(double t) {
    this(t, "TimedAbort " + t);
  }
  public void restart() { restart = true; }
}
```

We create a `Runnable` using a lambda expression, which is executed using the `static runAsync()` method of `CompletableFuture`. The value of `runAsync()` is that it returns immediately. As a result, `TimedAbort` doesn't hold any task open that would otherwise complete, but if it takes too long it will still terminate that task (`TimedAbort` is sometimes referred to as a *daemon*).

`TimedAbort` also allows you to `restart()` it, in order to keep a program open if there is some kind of useful activity going on.

We can see `TimedAbort` in action:

```
// lowlevel/TestAbort.java
import onjava.*;

public class TestAbort {
  public static void main(String[] args) {
    new TimedAbort(1);
    System.out.println("Napping for 4");
    new Nap(4);
  }
}
/* Output:
Napping for 4
TimedAbort 1.0
*/
```

If you comment out the `Nap` line, the program exits immediately, showing that `TimedAbort` is not holding the program open.

The first `IntGenerator` we'll look at has a `next()` that produces a series of even values:

```
// lowlevel/EvenProducer.java
// When threads collide
// {VisuallyInspectOutput}

public class EvenProducer extends IntGenerator {
  private int currentEvenValue = 0;
  @Override public int next() {
    ++currentEvenValue;                    // [1]
    ++currentEvenValue;
    return currentEvenValue;
  }
  public static void main(String[] args) {
    EvenChecker.test(new EvenProducer());
  }
}
/* Output:
419 not even!
425 not even!
423 not even!
421 not even!
417 not even!
*/
```

- **[1]** It's possible for one task to call `next()` after another task has performed the first increment of `currentEvenValue` but not the second. This puts the value into an "incorrect" state.

To prove this can happen, `EvenChecker.test()` creates a group of `EvenChecker` objects to continually read the output of an `EvenProducer` and test to see if each one is even. If not, the error is reported and the program is shut down.

Part of the problem with multithreaded programs is they can appear correct even when there's a bug, if the probability for failure is very low.

It's important to note that the increment operation itself requires multiple steps, and the task can be suspended by the threading mechanism in the midst of an increment—that is, increment is not an atomic operation in Java. So even a single increment isn't safe without protecting the task.

The program doesn't always terminate the first time a non-even number is produced. All the tasks don't instantly shut down, which is typical of a concurrent program.

Resolving Resource Contention

The previous example shows a fundamental problem when you use threads: You never know when a thread might run. Imagine sitting at a table with a fork, about to spear the last piece of food on a platter, and as your fork reaches for it, the food suddenly vanishes … because your thread was suspended and another diner came in and ate the food. That's the problem you're dealing with when writing concurrent programs. For concurrency to work, you need some way to prevent two tasks from accessing the same resource, at least during critical periods.

Preventing this kind of collision is a matter of putting a lock on a resource when one task is using it. The first task that accesses a resource must lock it, then the other tasks cannot access that resource until it is unlocked, at which time another task locks and uses it, and so on. If the front seat of the car is the limited resource, the child who shouts "shotgun!" acquires the lock (for the duration of that trip).

To solve the problem of thread collision, basic concurrency schemes *serialize access to shared resources*. This means only one task at a time is allowed to access the shared resource. This is ordinarily accomplished by putting a clause around a piece of code that only allows one task at a time to pass through that piece of code. Because this clause produces *mutual exclusion*, a common name for such a mechanism is *mutex*.

Consider the bathroom in your house; multiple people (tasks driven by threads) want exclusive use of the bathroom (the shared resource). To access the bathroom, a person knocks on the door to see if it's available. If so, they enter and lock the door. Any other task that wants to use the bathroom is "blocked" from using it, so those tasks wait at the door until the bathroom is available.

The analogy breaks down a bit when the bathroom is released and it comes time to give access to another task. There isn't actually a line of people, and we don't know for sure who gets the bathroom next, because the thread scheduler isn't deterministic. Instead, it's as if there is a group of blocked tasks milling about in front of the bathroom, and when the task that has locked the bathroom unlocks it and emerges, the thread scheduler decides which task will be the next to go in.

To prevent collisions over resources, Java has built-in support in the form of the `synchronized` keyword. When a task wishes to execute a piece of code guarded by the `synchronized` keyword, the Java compiler generates the code to see if the lock is available. If it is, the task acquires it, executes the code, and releases it.

The shared resource is typically just a piece of memory in the form of an object, but it can also be a file, an I/O port, or something like a printer. To control access to a shared resource, you first put it inside an object. Then any method that uses the resource can be made `synchronized`. If a task is inside a call to one of the `synchronized` methods, all other tasks are blocked from entering *any* of the `synchronized` methods of that object until the first task returns from its call.

You typically make fields `private` and access those fields only through methods. You prevent collisions by declaring methods using the `synchronized` keyword, like this:

```
synchronized void f() { /* ... */ }
synchronized void g() { /* ... */ }
```

All objects automatically contain a single lock (also called a *monitor*). When you call any `synchronized` method, that object is locked and no other `synchronized` method of that object can be called until the first one finishes and releases the lock. If `f()` is called for an object by one task, a different task cannot call `f()` *or* `g()` for the same object until `f()` is completed and releases the lock. Thus, there is a single lock shared by all `synchronized` methods of a particular object, and this lock can prevent object memory from being written by more than one task at a time.

It's especially important to make fields `private` when working with concurrency; otherwise the `synchronized` keyword cannot prevent another task from accessing a field directly, and thus producing collisions.

One thread can acquire an object's lock multiple times. This happens if one method calls a second method on the same object, which in turn calls another method on the same object, etc. The JVM keeps track of the number of times the object is locked. If the object is unlocked, it has a count of zero. As a thread acquires the lock for the first time, the count goes to one. Each time the same thread acquires another lock on the same object, the count is incremented. Naturally, multiple lock acquisition is only allowed for the thread that acquired the lock in the first place. Each time the thread leaves a `synchronized` method, the count is decremented, until the count goes to zero, releasing the lock entirely for use by other threads.

There's also a single lock *per class* (as part of the `Class` object for that class), so `synchronized static` methods can lock each other out from simultaneous access of `static` data on a class-wide basis.

When should you synchronize? Apply *Brian's Rule of Synchronization*:[203]

> If you are writing a variable that might next be read by another thread, or reading a variable that might have last been written by another thread, you must use synchronization, and further, both the reader and the writer must synchronize using the same monitor lock.

With more than one method in your class dealing with critical data, you must synchronize all relevant methods. If you synchronize only one of the methods, then the others are free to ignore the object lock and can be called with impunity. This is an important point: Every operation that accesses a critical shared resource must be `synchronized` or it won't work right.

Synchronizing the `EvenProducer`

By adding `synchronized` to `EvenProducer.java`, we can prevent the undesirable thread access:

```
// lowlevel/SynchronizedEvenProducer.java
// Simplifying mutexes with the synchronized keyword
import onjava.Nap;

public class
SynchronizedEvenProducer extends IntGenerator {
  private int currentEvenValue = 0;
  @Override public synchronized int next() {
    ++currentEvenValue;
    new Nap(0.01); // Cause failure faster
    ++currentEvenValue;
    return currentEvenValue;
  }
  public static void main(String[] args) {
    EvenChecker.test(new SynchronizedEvenProducer());
  }
}
/* Output:
No odd numbers discovered
*/
```

[203] From Brian Goetz, author of *Java Concurrency in Practice*, by Brian Goetz, Tim Peierls, Joshua Bloch, Joseph Bowbeer, David Holmes, and Doug Lea (Addison-Wesley, 2006).

A `Nap()` is inserted between the two increments, to raise the likelihood of a context switch while `currentEvenValue` is in an odd state. Because the mutex prevents more than one task at a time in the critical section, this will not produce a failure. The first task that enters `next()` acquires the lock, and any further tasks that try to acquire the lock are blocked from doing so until the first task releases the lock. At that point, the scheduling mechanism selects another task that is waiting on the lock. This way, only one task at a time can pass through the code that is guarded by the mutex.

The `volatile` Keyword

`volatile` is perhaps the most subtle and difficult keyword in Java. Fortunately, in modern Java you can virtually always avoid using it, and if you *do* see it used in code you should be skeptical and suspicious—there's a good chance that either the code is old, or whoever wrote that code doesn't understand the ramifications of `volatile` or concurrency in general (or both).

`volatile` is used for three reasons.

Word Tearing

Word tearing occurs when your data type is large enough (`long` and `double` in Java, which are both 64 bits) that the process of writing to a variable happens in two steps. The JVM is allowed to perform reads and writes of 64-bit quantities as two separate 32-bit operations,[204] raising the possibility that a context switch happens in the middle of a read or write, so other tasks see incorrect results. This is called *word tearing* because you might see the value after only part of it changes. Basically, a task can sometimes read the variable after the first step but before the second, resulting in a garbage value (This is not a problem with small variables such as `boolean` or `int`; anything except `long` or `double`).

Defining a `long` or `double` as `volatile`—in the absence of any other protection—prevents word tearing. However, `volatile` is superseded if those variables are guarded using `synchronized` or one of the `java.util.concurrent.atomic` classes. Also, `volatile` doesn't affect the fact that an increment isn't an atomic operation.

[204] Note that on a 64 bit processor this might not happen, eliminating the issue.

Visibility

The second problem falls under the "everything matters" part of Maxim 2 of The Four Maxims of Java Concurrency. You must assume that every task has its own processor, and each processor has its own local memory cache. This cache allows the processor to run faster because it won't always need to fetch data from main memory, which takes significantly longer than using cached values.

The problem arises because Java tries to be as efficient as possible. The whole point of the cache is to avoid reading from main memory. But with concurrency, it sometimes becomes unclear when Java should refresh values from main memory into the local cache—this issue is called *cache coherence*.

Each thread can store local copies of variables in the processor cache. Defining a field as `volatile` prevents these compiler optimizations so that reads and writes go directly to memory, and are not cached. As soon as a write occurs for that field, all reads across all tasks will see the change. If a `volatile` field happens to be held in a local cache, it is immediately written through to main memory, and any reads of that field will always occur from main memory.

`volatile` should be applied to a variable when:

1. That variable is simultaneously accessed by multiple tasks.
2. At least one of those accesses is a write.
3. You are trying to avoid synchronization (in modern Java, you can avoid synchronization using higher-level tools).

For example, if you use a variable as a flag to stop a task, that variable must at least be declared `volatile` (although that doesn't necessarily guarantee thread-safety for such a flag). Otherwise, when one task makes changes to the flag, those changes can be stored in the local processor cache and not flushed to main memory. When another task looks at the flag it doesn't see the changes. (I prefer the `AtomicBoolean` approach for flags shown in Concurrent Programming in the section Terminating Long-Running Tasks).

Any writes that a task makes to its own variables are always visible to that task, so you don't need to make a variable `volatile` if it is only used within a task.

If a single thread writes to a variable and other threads only read it, you can get away with making that variable `volatile`. In general, if you have multiple threads

writing to a variable, `volatile` won't solve your problems, and you must use `synchronized` instead to prevent race conditions. There is a special exception to this: It's possible to have multiple threads writing to that variable, *as long as they don't need to read it first and use the value to create the new value to write into the variable.* If those multiple threads use the old value in the result, you have a race condition because one of the other threads could modify the variable while your thread is doing its calculation. Even if you start out doing it right, imagine how easy it is to forget and introduce a breaking change during code modifications or maintenance, or for a different programmer who doesn't understand the issue (This is especially problematic in Java because programmers tend to rely heavily on compile-time checking to tell them if their code is correct).

It's important to understand that atomicity and volatility are distinct concepts. An atomic operation on a non-`volatile` variable has no guarantees about whether it is flushed to main memory.

Synchronization causes flushing to main memory, so if a variable is guarded by `synchronized` methods or blocks (or is a `java.util.concurrent.atomic` type), it is not necessary to make it `volatile`.

Reordering and *Happens Before*

Java may optimize performance by reordering instructions, as long as the result produces no changes in program behavior. This reordering, however, can affect the way the local processor cache interacts with main memory, producing subtle program bugs. It wasn't until Java 5 that this issue was understood and fixed. Now the `volatile` keyword prevents problematic reordering of reading and writing instructions around `volatile` variables. This reordering rule is called the *happens before* guarantee.

Instructions that appear before the read or write of a `volatile` variable are guaranteed to happen before that read or write. Similarly, any instructions that follow a read or write of a `volatile` variable are guaranteed to happen after the read or write. For example:

```
// lowlevel/ReOrdering.java

public class ReOrdering implements Runnable {
  int one, two, three, four, five, six;
  volatile int volaTile;
  @Override public void run() {
    one = 1;
    two = 2;
    three = 3;
    volaTile = 92;
    int x = four;
    int y = five;
    int z = six;
  }
}
```

The assignments of `one`, `two` and `three` may be reordered, as long as they all happen before the `volatile` write. Similarly, the `x`, `y` and `z` statements may be reordered as long as the `volatile` write happens before all of them. The `volatile` operation is often called a *memory barrier*. The *happens before* guarantee ensures that read and write instructions of `volatile` variables cannot be reordered across a memory barrier.

The *happens before* guarantee has another effect: When a thread writes to a `volatile` variable, then all other variables—including non-`volatiles`—changed by the thread before writing to the `volatile` variable are also flushed to main memory. When a thread reads a `volatile` variable it also reads all other variables—including non-`volatiles`—that were flushed to main memory together with the `volatile` variable. Although this is an important feature that solves some very devious bugs that appeared in Java before version 5, you shouldn't rely on it to "automatically" make surrounding variables `volatile`. If you intend for a variable to be `volatile`, that should be clear to anyone maintaining your code.

When to use `volatile`

With earlier versions of Java it wasn't too hard to write an example that demonstrated the need for `volatile`. If you search you can find such examples, but if you try them with Java 8 they don't work (none that I've found). I have struggled to write such an example but to no avail. This might result from improvements in the JVM or the

hardware or both. The effect could be beneficial for existing programs with storage that ought to be `volatile` but isn't; for such programs, failures will occur far less often—and the problem will be that much harder to track down.

If you are attempting to use `volatile`, you're probably trying to make a variable thread-safe without incurring the cost of synchronization. Because `volatile` is so subtle and tricky to use, I suggest not using it at all; instead, use one of the `java.util.concurrent.atomic` classes described later in this appendix. Those give complete thread-safety at a much lower cost than synchronization.

If you are trying to debug someone else's concurrency code, first look for uses of `volatile` and replace those with `Atomic` variables. Unless you know for sure that the programmer has a high level of understanding of concurrency, it's likely they are misusing `volatile`.

Atomicity

An oft-repeated but incorrect piece of lore in Java threading discussions is that "Atomic operations do not need synchronization." An *atomic operation* is one that cannot be interrupted by the thread scheduler; if the operation begins, it runs to completion before the possibility of a context switch. Relying on *innate* atomicity (atomicity that is part of the nature of a particular data type) is tricky and dangerous—you should only try to use innate atomicity instead of synchronization or a thread-safe data structure if you are a concurrency expert, or you have help from such an expert. If you think you're smart enough to play with this kind of fire, take this test:

> The Goetz Test: If you can write a high-performance JVM for a modern microprocessor, then you are qualified to think about whether you can avoid synchronizing.[205]

It's useful to *know* about innate atomicity, and to know that it helps implement some of the more clever `java.util.concurrent` library components. But strongly resist the urge to rely on it yourself.

[205] A corollary to this test is, "If someone implies that threading is straightforward, make sure that person is not making important decisions about your project. If that person already is, then you've got trouble."

Atomicity applies to "simple operations" on primitive types except for `longs` and `doubles`. Reading and writing primitive variables other than `long` and `double` is guaranteed to happen as indivisible (atomic) operations.

Atomic operations are thus not interruptible by the threading mechanism. Expert programmers can take advantage of this to write *lock-free code*, which does not need synchronization. But even this is an oversimplification. Sometimes, even when it seems like an atomic operation should be safe, it might not be. Readers of this book will not typically pass the aforementioned Goetz Test, and will thus not be qualified to try to replace synchronization with atomic operations. Trying to remove synchronization is usually a sign of premature optimization, and will cause you much trouble, probably without gaining much, or anything.

On multicore systems, *visibility* rather than atomicity is much more of an issue than on single-processor systems. Changes made by one task, even if they're atomic in the sense of not being interruptible, might not be visible to other tasks (the changes might be temporarily stored in a local processor cache, for example), so different tasks have a different view of the application's state. The synchronization mechanism, on the other hand, forces changes by one task on a multiprocessor system to be visible across the application. Without synchronization, it is indeterminate when changes become visible.

What qualifies as an atomic operation? Assignment and returning the value in a field might be. In C++ even the following *might* be atomic:

```
i++;    // Might be atomic in C++
i += 2; // Might be atomic in C++
```

But in C++, this depends on the compiler and processor. You're unable to write cross-platform code in C++ that relies on atomicity, because C++[206] didn't have a consistent *memory model*, as Java does.

In Java, the above operations are definitely *not* atomic, as shown by the JVM instructions produced from the following methods:

[206]The version I worked on; this might have been fixed in later standards.

```
// lowlevel/NotAtomic.java
// javap -c NotAtomic
// {ExcludeFromGradle}
// {VisuallyInspectOutput}

public class NotAtomic {
  int i;
  void f1() { i++; }
  void f2() { i += 3; }
}
/* Output:
Compiled from "NotAtomic.java"
public class NotAtomic {
  int i;

  public NotAtomic();
    Code:
       0: aload_0
       1: invokespecial #1 // Method java/lang/Object."<init>":()V
       4: return

  void f1();
    Code:
       0: aload_0
       1: dup
       2: getfield      #2 // Field
i:I
       5: iconst_1
       6: iadd
       7: putfield      #2 // Field
i:I
      10: return

  void f2();
    Code:
       0: aload_0
       1: dup
       2: getfield      #2 // Field
i:I
       5: iconst_3
       6: iadd
       7: putfield      #2 // Field
```

```
i:I
      10: return
}
*/
```

Each instruction produces a "get" and a "put," with instructions in between. So in between getting and putting, another task could modify the field, and thus the operations are not atomic.

Let's test this idea of atomicity by defining an abstract class with a method that increments a integer value by even amounts, and a `run()` that constantly calls that method:

```
// lowlevel/IntTestable.java
import java.util.function.*;

public abstract class
IntTestable implements Runnable, IntSupplier {
  abstract void evenIncrement();
  @Override public void run() {
    while(true)
      evenIncrement();
  }
}
```

`IntSupplier` is a functional interface with a `getAsInt()` method.

Now we can create a test that starts `run()` as a separate task and then fetches values to check whether they are even:

```
// lowlevel/Atomicity.java
import java.util.concurrent.*;
import onjava.TimedAbort;

public class Atomicity {
  public static void test(IntTestable it) {
    new TimedAbort(4, "No failures found");
    CompletableFuture.runAsync(it);
    while(true) {
      int val = it.getAsInt();
      if(val % 2 != 0) {
        System.out.println("failed with: " + val);
```

```
        System.exit(0);
      }
    }
  }
}
```

It's easy to blindly apply the idea of atomicity. Here, `getAsInt()` appears to be safely atomic:

```
// lowlevel/UnsafeReturn.java
import java.util.function.*;
import java.util.concurrent.*;

public class UnsafeReturn extends IntTestable {
  private int i = 0;
  public int getAsInt() { return i; }
  @Override
  public synchronized void evenIncrement() {
    i++; i++;
  }
  public static void main(String[] args) {
    Atomicity.test(new UnsafeReturn());
  }
}
/* Output:
failed with: 39
*/
```

However, `Atomicity.test()` fails with non-even values. Although `return i` is indeed an atomic operation, the lack of synchronization allows the value to be read while the object is in an unstable intermediate state. On top of this, `i` isn't `volatile` either, so there are visibility problems. Both `getValue()` *and* `evenIncrement()` must be `synchronized` (which also takes care of `i` without making it `volatile`):

```
// lowlevel/SafeReturn.java
import java.util.function.*;
import java.util.concurrent.*;

public class SafeReturn extends IntTestable {
  private int i = 0;
  public synchronized int getAsInt() { return i; }
  @Override
  public synchronized void evenIncrement() {
    i++; i++;
  }
  public static void main(String[] args) {
    Atomicity.test(new SafeReturn());
  }
}
/* Output:
No failures found
*/
```

Only concurrency experts are qualified to attempt optimizations in situations like this; again, apply Brian's Rule of Synchronization.

Josh's Serial Numbers

As a second example, consider something even simpler: a class producing serial numbers, inspired by Joshua Bloch's *Effective Java Programming Language Guide* (Addison-Wesley, 2001), p. 190. Each call to `nextSerialNumber()` must return a unique value:

```
// lowlevel/SerialNumbers.java

public class SerialNumbers {
  private volatile int serialNumber = 0;
  public int nextSerialNumber() {
    return serialNumber++; // Not thread-safe
  }
}
```

If you're coming from C++ or some other low-level background, you might expect the increment to be an atomic operation, because a C++ increment can often be

implemented as a single microprocessor instruction (although not in any consistent, reliable, cross-platform fashion). As noted before, however, a Java increment is *not* atomic and involves both a read and a write, so there's room for threading problems even in such a simple operation.

We throw in `volatile` here just to see if it might help. However, the real problem is that `nextSerialNumber()` accesses a shared mutable value without synchronizing.

To test `SerialNumbers`, we'll create a set that doesn't run out of memory, in case it takes a long time to detect a problem. The `CircularSet` shown here reuses the memory that stores `int`s, eventually overwriting old values (duplicates often happen quickly enough that you could probably use a `java.util.Set` instead):

```
// lowlevel/CircularSet.java
// Reuses storage so we don't run out of memory
import java.util.*;

public class CircularSet {
  private int[] array;
  private int size;
  private int index = 0;
  public CircularSet(int size) {
    this.size = size;
    array = new int[size];
    // Initialize to a value not produced
    // by SerialNumbers:
    Arrays.fill(array, -1);
  }
  public synchronized void add(int i) {
    array[index] = i;
    // Wrap index and write over old elements:
    index = ++index % size;
  }
  public synchronized boolean contains(int val) {
    for(int i = 0; i < size; i++)
      if(array[i] == val) return true;
    return false;
  }
}
```

The `add()` and `contains()` methods are `synchronized` to prevent thread collisions.

`SerialNumberChecker` contains a `CircularSet` holding the most recent serial numbers, and a `run()` that fills the `CircularSet` and ensures those serial numbers are unique:

```
// lowlevel/SerialNumberChecker.java
// Test SerialNumbers implementations for thread-safety
import java.util.concurrent.*;
import onjava.Nap;

public class SerialNumberChecker implements Runnable {
  private CircularSet serials = new CircularSet(1000);
  private SerialNumbers producer;
  public SerialNumberChecker(SerialNumbers producer) {
    this.producer = producer;
  }
  @Override public void run() {
    while(true) {
      int serial = producer.nextSerialNumber();
      if(serials.contains(serial)) {
        System.out.println("Duplicate: " + serial);
        System.exit(0);
      }
      serials.add(serial);
    }
  }
  static void test(SerialNumbers producer) {
    for(int i = 0; i < 10; i++)
      CompletableFuture.runAsync(
        new SerialNumberChecker(producer));
    new Nap(4, "No duplicates detected");
  }
}
```

`test()` creates multiple tasks to contend over a single `SerialNumbers` object. The competing `SerialNumberChecker` tasks try to produce a duplicate serial number (this happens more quickly on machines with more cores).

When we test the basic `SerialNumbers` class, it fails:

```
// lowlevel/SerialNumberTest.java

public class SerialNumberTest {
  public static void main(String[] args) {
    SerialNumberChecker.test(new SerialNumbers());
  }
}
/* Output:
Duplicate: 33280
Duplicate: 33278
Duplicate: 33290
Duplicate: 33277
*/
```

`volatile` is no help here. To solve the problem, add the `synchronized` keyword to `nextSerialNumber()`:

```
// lowlevel/SynchronizedSerialNumbers.java

public class
SynchronizedSerialNumbers extends SerialNumbers {
  private int serialNumber = 0;
  @Override
  public synchronized int nextSerialNumber() {
    return serialNumber++;
  }
  public static void main(String[] args) {
    SerialNumberChecker.test(
      new SynchronizedSerialNumbers());
  }
}
/* Output:
No duplicates detected
*/
```

`volatile` is no longer necessary, because the `synchronized` keyword ensures the behavior of `volatile`.

Reading and assigning primitives are supposed to be safe atomic operations. However, as seen in `UnsafeReturn.java`, it's still easy to use an atomic operation that accesses your object while it's in an unstable intermediate state. Making assumptions about this issue is tricky and dangerous. The most sensible thing to do is just to follow

Brian's Rule of Synchronization (and if you can, don't share variables in the first place).

Atomic Classes

Java 5 introduced special atomic variable classes such as `AtomicInteger`, `AtomicLong`, `AtomicReference`, etc. that provide atomic updates. These are fast, lock-free operations that take advantage of machine-level atomicity available on modern processors.

We can rewrite `UnsafeReturn.java` using `AtomicInteger`:

```
// lowlevel/AtomicIntegerTest.java
import java.util.concurrent.*;
import java.util.concurrent.atomic.*;
import java.util.*;
import onjava.*;

public class AtomicIntegerTest extends IntTestable {
  private AtomicInteger i = new AtomicInteger(0);
  public int getAsInt() { return i.get(); }
  @Override
  public void evenIncrement() { i.addAndGet(2); }
  public static void main(String[] args) {
    Atomicity.test(new AtomicIntegerTest());
  }
}
/* Output:
No failures found
*/
```

We've eliminated the `synchronized` keyword by using `AtomicInteger`.

Here is `SynchronizedEvenProducer.java` rewritten to use `AtomicInteger`:

```
// lowlevel/AtomicEvenProducer.java
// Atomic classes: occasionally useful in regular code
import java.util.concurrent.atomic.*;

public class AtomicEvenProducer extends IntGenerator {
  private AtomicInteger currentEvenValue =
    new AtomicInteger(0);
  @Override public int next() {
    return currentEvenValue.addAndGet(2);
  }
  public static void main(String[] args) {
    EvenChecker.test(new AtomicEvenProducer());
  }
}
/* Output:
No odd numbers discovered
*/
```

Again, the need for all other forms of synchronization is eliminated using `AtomicInteger`.

Here is an implementation of `SerialNumbers` using `AtomicInteger`:

```
// lowlevel/AtomicSerialNumbers.java
import java.util.concurrent.atomic.*;

public class
AtomicSerialNumbers extends SerialNumbers {
  private AtomicInteger serialNumber =
    new AtomicInteger();
  @Override public int nextSerialNumber() {
    return serialNumber.getAndIncrement();
  }
  public static void main(String[] args) {
    SerialNumberChecker.test(
      new AtomicSerialNumbers());
  }
}
/* Output:
No duplicates detected
*/
```

These are simple examples with a single field; when you create more complex classes you must determine which fields need protection, and in some cases you might still end up using the `synchronized` keyword on methods.

Critical Sections

Sometimes, you only want to prevent multiple thread access to part of the code inside a method instead of the entire method. The section of code to isolate is called a *critical section* and is created with the same `synchronized` keyword we use to protect an entire method, but using a different syntax. Here, `synchronized` specifies the object whose lock is used to synchronize the enclosed code:

```
synchronized(syncObject) {
  // This code can be accessed
  // by only one task at a time
}
```

This is also called a *synchronized block*. Before it can be entered, the `syncObject` lock must be acquired. If some other task already has this lock, then the critical section cannot be entered until the lock is released. When this happens, the task trying to acquire that lock is suspended. The scheduler periodically comes back and checks to see if the lock has been released; if so it wakes up the task.

The primary motivation for using a `synchronized` block instead of synchronizing the whole method is performance (and sometimes, clever algorithms—but be especially wary of cleverness where concurrency is concerned). The following example demonstrates that synchronizing a block instead of the whole method can make a method much more accessible to other tasks. The example counts the number of successful accesses to `method()` and launches tasks that compete to try to call `method()`:

```
// lowlevel/SynchronizedComparison.java
// Synchronizing blocks instead of entire methods
// speeds up access.
import java.util.*;
import java.util.stream.*;
import java.util.concurrent.*;
import java.util.concurrent.atomic.*;
import onjava.Nap;

abstract class Guarded {
  AtomicLong callCount = new AtomicLong();
  public abstract void method();
  @Override public String toString() {
    return getClass().getSimpleName() +
      ": " + callCount.get();
  }
}

class SynchronizedMethod extends Guarded {
  @Override public synchronized void method() {
    new Nap(0.01);
    callCount.incrementAndGet();
  }
}

class CriticalSection extends Guarded {
  @Override public void method() {
    new Nap(0.01);
    synchronized(this) {
      callCount.incrementAndGet();
    }
  }
}

class Caller implements Runnable {
  private Guarded g;
  Caller(Guarded g) { this.g = g; }
  private AtomicLong successfulCalls =
    new AtomicLong();
  private AtomicBoolean stop =
    new AtomicBoolean(false);
  @Override public void run() {
```

```
    new Timer().schedule(new TimerTask() {
      public void run() { stop.set(true); }
    }, 2500);
    while(!stop.get()) {
      g.method();
      successfulCalls.getAndIncrement();
    }
    System.out.println(
      "-> " + successfulCalls.get());
  }
}

public class SynchronizedComparison {
  static void test(Guarded g) {
    List<CompletableFuture<Void>> callers =
      Stream.of(
        new Caller(g),
        new Caller(g),
        new Caller(g),
        new Caller(g))
        .map(CompletableFuture::runAsync)
        .collect(Collectors.toList());
    callers.forEach(CompletableFuture::join);
    System.out.println(g);
  }
  public static void main(String[] args) {
    test(new CriticalSection());
    test(new SynchronizedMethod());
  }
}
/* Output:
-> 159
-> 159
-> 159
-> 159
CriticalSection: 636
-> 65
-> 21
-> 11
-> 68
SynchronizedMethod: 165
*/
```

The `Guarded` class keeps track of the number of successful calls to `method()` in `callCount`. `SynchronizedMethod` synchronizes the entire `method()`, while `CriticalSection` uses a `synchronized` block to only synchronize part of the method. This way, the time-consuming `Nap` can be excluded from the `synchronized` block. The output shows how much more available `method()` is for `CriticalSection`.

Keep in mind that using a `synchronized` block has risks: it requires that you know for certain that the un-`synchronized` code outside the block is actually safe.

`Caller` is a task that tries to call `method()` as many times as possible in a given period (and reports this number). To establish that period, we use `java.util.Timer` which is a little dated but still works well. It takes a `TimerTask`, which is not a functional interface so we can't use a lambda and must explicitly create the class (in this case, using an anonymous inner class). When it times out, it sets the `AtomicBoolean stop` flag to `true` so the loop will quit.

The `test()` method accepts a `Guarded` and creates four `Caller` tasks, all attached to the same `Guarded` object, so they compete to acquire the lock used by `method()`.

You will typically see variation in output from one run to the next. The results show that that `CriticalSection` allows much more access to its `method()` than `SynchronizedMethod`. This is typically the reason to use a `synchronized` block instead of synchronizing the whole method: To allow other tasks more access (as long as it is safe to do so).

Synchronizing on Other Objects

A `synchronized` block must synchronize upon an object. The most sensible object to use is usually just the current object via `synchronized(this)`, which is the approach taken in `CriticalSection` in the previous example. That way, when the lock is acquired for the `synchronized` block, other `synchronized` methods and critical sections in the same object cannot be called. Thus, the effect of the critical section when synchronizing on `this` is to reduce the scope of synchronization.

Sometimes you must synchronize on another object, but if you do this you must ensure that all relevant tasks are synchronizing on the same object. The following example demonstrates that two tasks can enter an object when the methods in that object synchronize on different locks:

```
// lowlevel/SyncOnObject.java
// Synchronizing on another object
import java.util.*;
import java.util.stream.*;
import java.util.concurrent.*;
import onjava.Nap;

class DualSynch {
  ConcurrentLinkedQueue<String> trace =
    new ConcurrentLinkedQueue<>();
  public synchronized void f(boolean nap) {
    for(int i = 0; i < 5; i++) {
      trace.add(String.format("f() " + i));
      if(nap) new Nap(0.01);
    }
  }
  private Object syncObject = new Object();
  public void g(boolean nap) {
    synchronized(syncObject) {
      for(int i = 0; i < 5; i++) {
        trace.add(String.format("g() " + i));
        if(nap) new Nap(0.01);
      }
    }
  }
}

public class SyncOnObject {
  static void test(boolean fNap, boolean gNap) {
    DualSynch ds = new DualSynch();
    List<CompletableFuture<Void>> cfs =
      Arrays.stream(new Runnable[] {
        () -> ds.f(fNap), () -> ds.g(gNap) })
        .map(CompletableFuture::runAsync)
        .collect(Collectors.toList());
    cfs.forEach(CompletableFuture::join);
    ds.trace.forEach(System.out::println);
  }
  public static void main(String[] args) {
    test(true, false);
    System.out.println("****");
    test(false, true);
```

```
  }
}
/* Output:
f() 0
g() 0
g() 1
g() 2
g() 3
g() 4
f() 1
f() 2
f() 3
f() 4
****
f() 0
g() 0
f() 1
f() 2
f() 3
f() 4
g() 1
g() 2
g() 3
g() 4
*/
```

`DualSync.f()` synchronizes on `this` (by synchronizing the entire method), and `g()` has a `synchronized` block that synchronizes on `syncObject`. Thus, the two synchronizations are independent. This is demonstrated in `test()` by running two independent task that call `f()` and `g`. The `fNap` and `gNap` flags indicate whether `f()` or `g()`, respectively, should call `Nap()` within its `for` loop. When `f()` naps, for example, it continues to hold its lock, but you can see that this doesn't prevent `g()` from being called, and vice-versa.

Using Explicit `Lock` Objects

The `java.util.concurrent` library contains an explicit mutex mechanism defined in `java.util.concurrent.locks`. The `Lock` object must be explicitly created, locked and unlocked, so it produces less elegant code than the built-in `synchronized`

keyword. However, it is more flexible for solving certain types of problems. Here is `SynchronizedEvenProducer.java` rewritten to use explicit `Lock`s:

```
// lowlevel/MutexEvenProducer.java
// Preventing thread collisions with mutexes
import java.util.concurrent.locks.*;
import onjava.Nap;

public class MutexEvenProducer extends IntGenerator {
  private int currentEvenValue = 0;
  private Lock lock = new ReentrantLock();
  @Override public int next() {
    lock.lock();
    try {
      ++currentEvenValue;
      new Nap(0.01); // Cause failure faster
      ++currentEvenValue;
      return currentEvenValue;
    } finally {
      lock.unlock();
    }
  }
  public static void main(String[] args) {
    EvenChecker.test(new MutexEvenProducer());
  }
}
/*
No odd numbers discovered
*/
```

`MutexEvenProducer` adds a mutex called `lock` and uses the `lock()` and `unlock()` methods to create a critical section within `next()`. When you use `Lock` objects, it is important to use the idiom shown here: Right after the call to `lock()`, you must place a `try-finally` statement with `unlock()` in the `finally` clause—this is the only way to guarantee that the lock is always released. Note that the `return` statement must occur inside the `try` clause to ensure that the `unlock()` doesn't happen too early and expose the data to a second task.

Although the `try-finally` requires more code than using the `synchronized` keyword, it also represents one of the advantages of explicit `Lock` objects. If something fails using the `synchronized` keyword, an exception is thrown, but you don't get the

chance to do any cleanup to maintain your system in a good state. With explicit `Lock` objects, you can maintain proper state in your system using the `finally` clause.

In general, when you use `synchronized`, there is less code to write, and the opportunity for user error is greatly reduced, so you'll usually only use the explicit `Lock` objects when you're solving special problems. For example, with the `synchronized` keyword, you can't try and fail to acquire a lock, or try to acquire a lock for a certain amount of time, then give up—to do this, you must use the `concurrent` library:

```
// lowlevel/AttemptLocking.java
// Locks in the concurrent library allow you
// to give up on trying to acquire a lock
import java.util.concurrent.*;
import java.util.concurrent.locks.*;
import onjava.Nap;

public class AttemptLocking {
  private ReentrantLock lock = new ReentrantLock();
  public void untimed() {
    boolean captured = lock.tryLock();
    try {
      System.out.println("tryLock(): " + captured);
    } finally {
      if(captured)
        lock.unlock();
    }
  }
  public void timed() {
    boolean captured = false;
    try {
      captured = lock.tryLock(2, TimeUnit.SECONDS);
    } catch(InterruptedException e) {
      throw new RuntimeException(e);
    }
    try {
      System.out.println(
        "tryLock(2, TimeUnit.SECONDS): " + captured);
    } finally {
      if(captured)
        lock.unlock();
    }
  }
```

```
  public static void main(String[] args) {
    final AttemptLocking al = new AttemptLocking();
    al.untimed(); // True -- lock is available
    al.timed();   // True -- lock is available
    // Now create a second task to grab the lock:
    CompletableFuture.runAsync( () -> {
        al.lock.lock();
        System.out.println("acquired");
    });
    new Nap(0.1);  // Give the second task a chance
    al.untimed(); // False -- lock grabbed by task
    al.timed();   // False -- lock grabbed by task
  }
}
/* Output:
tryLock(): true
tryLock(2, TimeUnit.SECONDS): true
acquired
tryLock(): false
tryLock(2, TimeUnit.SECONDS): false
*/
```

A `ReentrantLock` can try and fail to acquire the lock, so if someone else already has the lock, you can decide to go off and do something else rather than waiting until it is free, as in the `untimed()` method. In `timed()`, an attempt is made to acquire the lock which can fail after 2 seconds (note the `TimeUnit` class to specify units). In `main()`, a separate `Thread` is created as an anonymous class, and it acquires the lock so the `untimed()` and `timed()` methods have something to contend with.

The explicit `Lock` object provides finer-grained control over locking and unlocking than does the built-in `synchronized` lock. This is useful for implementing specialized synchronization structures, such as *hand-over-hand locking* (also called *lock coupling*), used for traversing the nodes of a linked list—the traversal code must capture the lock of the next node before it releases the current node's lock.

Library Components

The `java.util.concurrent` library provides a significant number of classes designed to solve concurrency problems, and can help you produce simpler and

more robust concurrent programs. Note, however, that these tools are lower-level mechanisms than parallel streams or `CompletableFutures`.

In this section we'll look at a couple of examples using different components, then talk a little about how *lock-free* library components work.

`DelayQueue`

This is an unbounded `BlockingQueue` of objects that implement the `Delayed` interface. An object can only be taken from the queue when its delay has expired. The queue is sorted so the object at the head has a delay that has expired for the longest time. If no delay has expired, then there is no head element and `poll()` will return `null` (because of this, you cannot place `null` elements in the queue).

Here's an example where the `Delayed` objects are themselves tasks, and the `DelayedTaskConsumer` takes the most "urgent" task (the one that expired for the longest time) off the queue and runs it. Note that `DelayQueue` is thus a variation of a priority queue.

```
// lowlevel/DelayQueueDemo.java
import java.util.*;
import java.util.stream.*;
import java.util.concurrent.*;
import static java.util.concurrent.TimeUnit.*;

class DelayedTask implements Runnable, Delayed {
  private static int counter = 0;
  private final int id = counter++;
  private final int delta;
  private final long trigger;
  protected static List<DelayedTask> sequence =
    new ArrayList<>();
  DelayedTask(int delayInMilliseconds) {
    delta = delayInMilliseconds;
    trigger = System.nanoTime() +
      NANOSECONDS.convert(delta, MILLISECONDS);
    sequence.add(this);
  }
  @Override public long getDelay(TimeUnit unit) {
    return unit.convert(
```

```
      trigger - System.nanoTime(), NANOSECONDS);
  }
  @Override public int compareTo(Delayed arg) {
    DelayedTask that = (DelayedTask)arg;
    if(trigger < that.trigger) return -1;
    if(trigger > that.trigger) return 1;
    return 0;
  }
  @Override public void run() {
    System.out.print(this + " ");
  }
  @Override public String toString() {
    return
      String.format("[%d] Task %d", delta, id);
  }
  public String summary() {
    return String.format("(%d:%d)", id, delta);
  }
  public static class EndTask extends DelayedTask {
    EndTask(int delay) { super(delay); }
    @Override public void run() {
      sequence.forEach(dt ->
        System.out.println(dt.summary()));
    }
  }
}

public class DelayQueueDemo {
  public static void
  main(String[] args) throws Exception {
    DelayQueue<DelayedTask> tasks =
      Stream.concat( // Random delays:
        new Random(47).ints(20, 0, 4000)
          .mapToObj(DelayedTask::new),
        // Add the summarizing task:
        Stream.of(new DelayedTask.EndTask(4000)))
      .collect(Collectors
        .toCollection(DelayQueue::new));
    while(tasks.size() > 0)
      tasks.take().run();
  }
}
```

```
/* Output:
[128] Task 12 [429] Task 6 [555] Task 2 [551] Task 13
[693] Task 3 [809] Task 15 [961] Task 5 [1258] Task 1
[1258] Task 20 [1520] Task 19 [1861] Task 4 [1998] Task
17 [2200] Task 8 [2207] Task 10 [2288] Task 11 [2522]
Task 9 [2589] Task 14 [2861] Task 18 [2868] Task 7
[3278] Task 16 (0:4000)
(1:1258)
(2:555)
(3:693)
(4:1861)
(5:961)
(6:429)
(7:2868)
(8:2200)
(9:2522)
(10:2207)
(11:2288)
(12:128)
(13:551)
(14:2589)
(15:809)
(16:3278)
(17:1998)
(18:2861)
(19:1520)
(20:1258)
*/
```

`DelayedTask` contains a `List<DelayedTask>` called `sequence` that preserves the order of task creation so we can see that sorting does in fact take place.

The `Delayed` interface has one method, `getDelay()`, which says how long until the delay time expires or how long ago the delay time has expired. This method forces us to use the `TimeUnit` class because that's the argument type. This turns out to be a convenient class because you can convert units without doing any calculations. For example, the value of `delta` is stored in milliseconds, but `System.nanoTime()` produces time in nanoseconds. You can convert the value of `delta` by giving the units it is in and the units you want it to be in, like this:

```
NANOSECONDS.convert(delta, MILLISECONDS);
```

In `getDelay()`, the desired units are passed as the `unit` argument, and you use this to convert the time difference from the trigger time into the units requested by the caller, without even knowing what those units are (this is a simple example of the *Strategy* design pattern, where part of the algorithm is passed in as an argument).

For sorting, the `Delayed` interface also inherits the `Comparable` interface. `compareTo()` must be implemented so it produces a reasonable comparison.

The output shows that that task creation order has no effect on execution order—instead, the tasks are executed in delay order as expected.

`PriorityBlockingQueue`

This is basically a priority queue that has blocking retrieval operations. In the following example, A `Prioritized` is given a priority number. Several instances of the `Producer` task insert `Prioritized` objects into the `PriorityBlockingQueue`, but with random delays between insertions. The single `Consumer` task is then presented with multiple options when it does a `take()`, and the `PriorityBlockingQueue` hands it the `Prioritized` object with the highest priority *at that moment.*

The `static counter` in `Prioritized` is an `AtomicInteger`. This is necessary because there are multiple `Producers` running in parallel; without `AtomicInteger` you'll see duplicate `id` numbers. This issue was covered in Concurrent Programming in the section Constructors are not Thread-Safe.

```
// lowlevel/PriorityBlockingQueueDemo.java
import java.util.*;
import java.util.stream.*;
import java.util.concurrent.*;
import java.util.concurrent.atomic.*;
import onjava.Nap;

class Prioritized implements Comparable<Prioritized>  {
  private static AtomicInteger counter =
    new AtomicInteger();
  private final int id = counter.getAndIncrement();
  private final int priority;
  private static List<Prioritized> sequence =
```

```
    new CopyOnWriteArrayList<>();
  Prioritized(int priority) {
    this.priority = priority;
    sequence.add(this);
  }
  @Override public int compareTo(Prioritized arg) {
    return priority < arg.priority ? 1 :
      (priority > arg.priority ? -1 : 0);
  }
  @Override public String toString() {
    return String.format(
      "[%d] Prioritized %d", priority, id);
  }
  public void displaySequence() {
    int count = 0;
    for(Prioritized pt : sequence) {
      System.out.printf("(%d:%d)", pt.id, pt.priority);
      if(++count % 5 == 0)
        System.out.println();
    }
  }
  public static class EndSentinel extends Prioritized {
    EndSentinel() { super(-1); }
  }
}

class Producer implements Runnable {
  private static AtomicInteger seed =
    new AtomicInteger(47);
  private SplittableRandom rand =
    new SplittableRandom(seed.getAndAdd(10));
  private Queue<Prioritized> queue;
  Producer(Queue<Prioritized> q) {
    queue = q;
  }
  @Override public void run() {
    rand.ints(10, 0, 20)
      .mapToObj(Prioritized::new)
      .peek(p -> new Nap(rand.nextDouble() / 10))
      .forEach(p -> queue.add(p));
    queue.add(new Prioritized.EndSentinel());
  }
```

```
}

class Consumer implements Runnable {
  private PriorityBlockingQueue<Prioritized> q;
  private SplittableRandom rand =
    new SplittableRandom(47);
  Consumer(PriorityBlockingQueue<Prioritized> q) {
    this.q = q;
  }
  @Override public void run() {
    while(true) {
      try {
        Prioritized pt = q.take();
        System.out.println(pt);
        if(pt instanceof Prioritized.EndSentinel) {
          pt.displaySequence();
          break;
        }
        new Nap(rand.nextDouble() / 10);
      } catch(InterruptedException e) {
        throw new RuntimeException(e);
      }
    }
  }
}

public class PriorityBlockingQueueDemo {
  public static void main(String[] args) {
    PriorityBlockingQueue<Prioritized> queue =
      new PriorityBlockingQueue<>();
    CompletableFuture.runAsync(new Producer(queue));
    CompletableFuture.runAsync(new Producer(queue));
    CompletableFuture.runAsync(new Producer(queue));
    CompletableFuture.runAsync(new Consumer(queue))
      .join();
  }
}
/* Output:
[15] Prioritized 1
[17] Prioritized 0
[17] Prioritized 5
[16] Prioritized 6
```

```
[14] Prioritized 9
[12] Prioritized 2
[11] Prioritized 4
[11] Prioritized 12
[13] Prioritized 13
[12] Prioritized 16
[14] Prioritized 17
[15] Prioritized 23
[18] Prioritized 26
[16] Prioritized 29
[12] Prioritized 18
[11] Prioritized 30
[11] Prioritized 24
[10] Prioritized 15
[10] Prioritized 22
[8] Prioritized 25
[8] Prioritized 11
[8] Prioritized 10
[6] Prioritized 31
[3] Prioritized 7
[2] Prioritized 20
[1] Prioritized 3
[0] Prioritized 19
[0] Prioritized 8
[0] Prioritized 14
[0] Prioritized 21
[-1] Prioritized 28
(0:17)(1:15)(2:12)(3:1)(4:11)
(5:17)(6:16)(7:3)(8:0)(9:14)
(10:8)(11:8)(12:11)(13:13)(14:0)
(15:10)(16:12)(17:14)(18:12)(19:0)
(20:2)(21:0)(22:10)(23:15)(24:11)
(25:8)(26:18)(27:-1)(28:-1)(29:16)
(30:11)(31:6)(32:-1)
*/
```

As with the previous example, the creation order of the `Prioritized` objects is remembered in the `List sequence`, for comparison with the actual order of execution. The `EndSentinel` is a special type that tells the `Consumers` to shut down.

`Producer` uses an `AtomicInteger` to seed the `SplittableRandom` so that different `Producers` produce different sequences. This is required because multiple `Producers`

are created in parallel, and otherwise the construction process would not be thread-safe.

The `Producers` and `Consumer` connect to each other through a `PriorityBlockingQueue`. Because the blocking nature of the queue provides all necessary synchronization, notice that no explicit synchronization is necessary—you don't think about whether the queue has any elements in it when you're reading from it, because the queue will block the reader when it is out of elements.

Lock-Free Collections

The Collections chapter emphasized collections as a fundamental programming tool, and this includes concurrency. For this reason, early collections like `Vector` and `Hashtable` have many methods using the `synchronized` mechanism. This caused unacceptable overhead when those collections were not used in multithreaded applications. In Java 1.2, the new collections library was unsynchronized, and the `Collections` class was given various `static synchronized` decoration methods to synchronize the different types of collections. Although this was an improvement because it gave you a choice about whether you use synchronization with your collection, the overhead is still based on `synchronized` locking. Java 5 added new collections specifically to increase thread-safe performance, using clever techniques to eliminate locking.

Lock-free collections have an intriguing feature: Modifications to the collections can happen at the same time that reads are occurring, as long as the readers can only see the results of *completed* modifications. This is implemented using a number of strategies. To give you a flavor of how they work, we'll look at a couple of them.

The Copying Strategy

With the "copying" strategy, a modification is performed on a separate copy of a portion of the data structure (or sometimes a copy of the whole thing), and this copy is invisible during the modification process. Only when the modification is complete is the modified structure safely swapped with the "main" data structure, and after that readers will see the modification.

In `CopyOnWriteArrayList`, a write copies the entire underlying array. The original array is left in place so reads can safely occur while the copied array is modified.

When the modification is complete, an atomic operation swaps in the new array so new reads will see the new information. One of the benefits of `CopyOnWriteArrayList` is it does not throw `ConcurrentModificationException` when multiple iterators are traversing and modifying the list, so you don't write special code to protect against such exceptions, as you've had to do in the past.

`CopyOnWriteArraySet` uses `CopyOnWriteArrayList` to achieve its lock-free behavior.

`ConcurrentHashMap` and `ConcurrentLinkedQueue` use similar techniques to allow concurrent reads and writes, but only portions of the collection are copied and modified rather than the entire collection. However, readers will still not see any incomplete modifications. `ConcurrentHashMap` doesn't throw `ConcurrentModificationExceptions`.

Compare-And-Swap (CAS)

In *Compare-And-Swap* (CAS), you take a value from memory and hold on to the original value while calculating a new one. Then you use the CAS instruction which compares the original value with the one currently in memory, and *if the two values are equal* swaps in the result of your calculation for the old value, all in a single atomic operation. If the original-value comparison fails, the swap does not take place because it means another thread has modified the memory in the meantime. In that case, your code must try again, taking a new original value and repeating the operation.

If the memory is only lightly contended, the CAS operation almost always goes through with no repeated attempts, so it is very fast. In contrast, a `synchronized` operation requires the cost of acquiring and releasing the lock every single time, which is much more expensive with no additional benefit. As contention over the memory increases, the operation using CAS slows down because it must repeat itself more often, but this is a dynamic response to more contention. It really is an elegant approach.

The best part is that many modern processors have a CAS instruction in their assembly language, and this is used by the CAS operations in the JVM (such as those in the `Atomic` classes). The CAS instruction is atomic in the hardware and as fast as you could hope for such an operation.

Summary

This appendix was included primarily so you have *some* understanding of low-level concurrency code when you encounter it, although it is far from a comprehensive treatment of the subject. For that, you'll need to start with *Java Concurrency in Practice*, by Brian Goetz, Tim Peierls, Joshua Bloch, Joseph Bowbeer, David Holmes, and Doug Lea (Addison-Wesley, 2006). Ideally, that book will completely scare you away from attempting low-level concurrency in Java. If it doesn't, you're almost certainly suffering from the *Dunning-Kruger Effect*, a cognitive bias whereby "the less you know, the more confident you are in your abilities." Keep in mind that the current language designers are *still* cleaning up the messes made from the overconfidence of the early language designers (for example, look at how many of the methods in the `Thread` class are `deprecated`, and that `volatile` wasn't working correctly until Java 5).

Here are the steps to follow for concurrent programming:

1. Don't do it. Figure out some other way to make your program faster.
2. If you must do it, use the modern, high-level tools shown in Concurrent Programming—parallel `Streams` and `CompletableFutures`.
3. Don't share variables between tasks. Any information that must pass between tasks should use concurrent data structures from the `java.util.concurrent` library.
4. If you must share variables, either use a `java.util.concurrent.atomic` type or apply `synchronized` to any method that can either directly or indirectly access those variables. It's easy to get fooled into thinking you've got everything covered when you don't. Seriously, try to use step #3 instead.
5. If step #4 produces results that are somehow too slow, you can try to tune things using `volatile` or some other technique but if you are reading this book and think you are ready to try those approaches, you're out of your depth. Return to step #1.

It's usually possible to write concurrent programs using only `java.util.concurrent` library components, completely avoiding the challenges of applying `volatile` and `synchronized`. Note that I was able to do this with the examples in Concurrent Programming.

Appendix: Data Compression

The Java I/O library contains classes that read and write streams in a compressed format. You wrap these around other I/O classes to provide compression functionality.

These classes are not derived from the `Reader` and `Writer` classes, but instead are part of the `InputStream` and `OutputStream` hierarchies. This is because the compression library works with bytes, not characters. However, you might sometimes be forced to mix the two types of streams. (Remember you can use `InputStreamReader` and `OutputStreamWriter` to provide easy conversion between one type and another.)

Compression class	Function
`CheckedInputStream`	`getCheckSum()` produces checksum for any `InputStream` (not just decompression).
`CheckedOutputStream`	`getCheckSum()` produces checksum for any `OutputStream` (not just compression).
`DeflaterOutputStream`	Base class for compression classes.
`ZipOutputStream`	A `DeflaterOutputStream` that compresses data into the Zip file format.
`GZIPOutputStream`	A `DeflaterOutputStream` that compresses data into the GZIP file format.
`InflaterInputStream`	Base class for decompression classes.
`ZipInputStream`	An `InflaterInputStream` that decompresses data that was stored in the Zip file format.
`GZIPInputStream`	An `InflaterInputStream` that decompresses data that was stored in the GZIP file format.

Although there are many compression algorithms, Zip and GZIP are probably the

most common. You can easily manipulate your compressed data with the many tools available for reading and writing these formats.

Simple Compression with GZIP

The GZIP interface is simple and thus is probably more appropriate when you have a single stream of data to compress (rather than a container of dissimilar pieces of data). This example compresses a single file:

```
// compression/GZIPcompress.java
// {java GZIPcompress GZIPcompress.java}
// {VisuallyInspectOutput}
import java.util.zip.*;
import java.io.*;

public class GZIPcompress {
  public static void main(String[] args) {
    if(args.length == 0) {
      System.out.println(
        "Usage: \nGZIPcompress file\n" +
        "\tUses GZIP compression to compress " +
        "the file to test.gz");
      System.exit(1);
    }
    try(
      InputStream in = new BufferedInputStream(
        new FileInputStream(args[0]));
      BufferedOutputStream out =
        new BufferedOutputStream(
          new GZIPOutputStream(
            new FileOutputStream("test.gz")))
    ) {
      System.out.println("Writing file");
      int c;
      while((c = in.read()) != -1)
        out.write(c);
    } catch(IOException e) {
      throw new RuntimeException(e);
    }
    System.out.println("Reading file");
```

```
    try(
      BufferedReader in2 = new BufferedReader(
        new InputStreamReader(new GZIPInputStream(
          new FileInputStream("test.gz"))))
    ) {
      in2.lines().forEach(System.out::println);
    } catch(IOException e) {
      throw new RuntimeException(e);
    }
  }
}
```

Using the compression classes is straightforward; you wrap your output stream in a `GZIPOutputStream` or `ZipOutputStream`, and your input stream in a `GZIPInputStream` or `ZipInputStream`. All else is ordinary I/O reading and writing. This is an example of mixing the `char`-oriented streams with the byte-oriented streams; `in` uses the `Reader` classes, whereas `GZIPOutputStream`'s constructor can accept only an `OutputStream` object, not a `Writer` object. When the file is opened, the `GZIPInputStream` is converted to a `Reader`.

Multifile Storage with Zip

The library that supports the Zip format is more extensive than the GZIP library. With it you can easily store multiple files, and there's even a separate class to make the process of reading a Zip file easy. The library uses the standard Zip format so it works seamlessly with all the Zip tools currently downloadable on the Internet. The following example has the same form as the previous example, but it handles as many command-line arguments as you want. In addition, it shows the `Checksum` classes calculating and verifying the checksum for the file. There are two `Checksum` types: `Adler32` (which is faster) and `CRC32` (slower but slightly more accurate).

```
// compression/ZipCompress.java
// Uses Zip compression to compress any
// number of files given on the command line
// {java ZipCompress ZipCompress.java}
// {VisuallyInspectOutput}
import java.util.zip.*;
import java.io.*;
import java.util.*;

public class ZipCompress {
  public static void main(String[] args) {
    try(
      FileOutputStream f =
        new FileOutputStream("test.zip");
      CheckedOutputStream csum =
        new CheckedOutputStream(f, new Adler32());
      ZipOutputStream zos = new ZipOutputStream(csum);
      BufferedOutputStream out =
        new BufferedOutputStream(zos)
    ) {
      zos.setComment("A test of Java Zipping");
      // No corresponding getComment(), though.
      for(String arg : args) {
        System.out.println("Writing file " + arg);
        try(
          InputStream in = new BufferedInputStream(
            new FileInputStream(arg))
        ) {
          zos.putNextEntry(new ZipEntry(arg));
          int c;
          while((c = in.read()) != -1)
            out.write(c);
        }
        out.flush();
      }
      // Checksum valid only after the file is closed!
      System.out.println(
        "Checksum: " + csum.getChecksum().getValue());
    } catch(IOException e) {
      throw new RuntimeException(e);
    }
    // Now extract the files:
```

```
    System.out.println("Reading file");
    try(
      FileInputStream fi =
        new FileInputStream("test.zip");
      CheckedInputStream csumi =
        new CheckedInputStream(fi, new Adler32());
      ZipInputStream in2 = new ZipInputStream(csumi);
      BufferedInputStream bis =
        new BufferedInputStream(in2)
    ) {
      ZipEntry ze;
      while((ze = in2.getNextEntry()) != null) {
        System.out.println("Reading file " + ze);
        int x;
        while((x = bis.read()) != -1)
          System.out.write(x);
      }
      if(args.length == 1)
        System.out.println(
          "Checksum: "+csumi.getChecksum().getValue());
    } catch(IOException e) {
      throw new RuntimeException(e);
    }
    // Alternative way to open and read Zip files:
    try(
      ZipFile zf = new ZipFile("test.zip")
    ) {
      Enumeration e = zf.entries();
      while(e.hasMoreElements()) {
        ZipEntry ze2 = (ZipEntry)e.nextElement();
        System.out.println("File: " + ze2);
        // ... and extract the data as before
      }
    } catch(IOException e) {
      throw new RuntimeException(e);
    }
  }
}
```

For each file to add to the archive, you must call `putNextEntry()` and pass it a `ZipEntry` object. The `ZipEntry` object contains an extensive interface that gets and sets all the data available on that particular entry in your Zip file: name,

compressed and uncompressed sizes, date, CRC checksum, extra field data, comment, compression method, and whether it's a directory entry. However, even though the Zip format has a way to set a password, this is not supported in Java's Zip library. And although `CheckedInputStream` and `CheckedOutputStream` support both `Adler32` and `CRC32` checksums, the `ZipEntry` class supports only an interface for CRC. This is a restriction of the underlying Zip format, but it might limit you from using the faster `Adler32`.

To extract files, `ZipInputStream` has a `getNextEntry()` method that returns the next `ZipEntry` if there is one. As a more succinct alternative, you can read the file using a `ZipFile` object, which has a method `entries()` to return an `Enumeration` to the `ZipEntries`.

To read the checksum, you must somehow have access to the associated `Checksum` object. Here, a reference to the `CheckedOutputStream` and `CheckedInputStream` objects is retained, but you can also just hold on to a reference to the `Checksum` object.

A baffling method in Zip streams is `setComment()`. As shown in `ZipCompress.java`, you can set a comment when you're writing a file, but there's no way to recover the comment in the `ZipInputStream`. Comments appear to be supported fully on an entry-by-entry basis only via `ZipEntry`.

You are not limited to files when using the `GZIP` or `Zip` libraries—you can compress anything, including data sent through a network connection.

Java Archives (Jars)

The Zip format is also used in the JAR (Java ARchive) file format, a way to collect a group of files into a single compressed file, just like Zip. However, like everything else in Java, JAR files are cross-platform, so you don't worry about platform issues. You can also include audio and image files as well as class files.

A JAR file consists of a single file containing a collection of zipped files along with a "manifest" that describes them. (You can create your own manifest file; otherwise, the `jar` program will do it for you.) You can find out more about JAR manifests in the JDK documentation.

The `jar` utility that comes with the JDK automatically compresses the files of your choice. You invoke it on the command line:

```
jar [options] destination [manifest] inputfile(s)
```

The options are a collection of letters (no hyphen or any other indicator is necessary). Unix/Linux users will note the similarity to `tar` options. These are:

Letter	Effect
`c`	Creates a new or empty archive.
`t`	Lists the table of contents.
`x`	Extracts all files.
`x` file	Extracts the named file.
`f`	Says, "I'm going to give you the name of the file." If you don't use this, `jar` assumes its input will come from standard input, or, if it is creating a file, its output will go to standard output.
`m`	Says that the first argument is the name of the user-created manifest file.
`v`	Generates verbose output describing what `jar` is doing.
`0`	Only stores the files; doesn't compress the files (use to create a JAR file you put in your classpath).
`M`	Doesn't automatically create a manifest file.

If a subdirectory is included in the files put into the JAR file, that subdirectory is automatically added, including all of its subdirectories, etc. Path information is also preserved.

Here are some typical ways to invoke `jar`. The following command creates a JAR file called `myJarFile.jar` that contains all class files in the current directory, along with an automatically generated manifest file:

```
jar cf myJarFile.jar *.class
```

The next command is like the previous example, but it adds a user-created manifest

file called `myManifestFile.mf`:

```
jar cmf myJarFile.jar myManifestFile.mf *.class
```

This produces a table of contents of the files in `myJarFile.jar`:

```
jar tf myJarFile.jar
```

This adds the "verbose" flag to give more detailed information about files in `myJarFile.jar`:

```
jar tvf myJarFile.jar
```

Assuming `audio`, `classes`, and `image` are subdirectories, this combines all subdirectories into the file `myApp.jar`. The "verbose" flag is also included to give extra feedback while the `jar` program is working:

```
jar cvf myApp.jar audio classes image
```

If you create a JAR file using the `0` (zero) option, that file can be placed in your CLASSPATH:

```
CLASSPATH="lib1.jar;lib2.jar;"
```

Then Java can search `lib1.jar` and `lib2.jar` for class files.

The `jar` tool isn't as general-purpose as a `Zip` utility. For example, you can't add or update files to an existing JAR file; you can create JAR files only from scratch. Also, you can't move files into a JAR file, erasing them as they are moved. However, a JAR file created on one platform is transparently readable by the `jar` tool on any other platform (a problem that sometimes plagues `Zip` utilities).

Appendix: Object Serialization

When you create an object, it exists for as long as you need it, but under no circumstances does it exist when the program terminates. While this makes sense at first, there are situations where it would be quite useful if an object existed and held its information even while the program wasn't running. Then, the next time you started the program, the object would be there with the same information it had the previous time the program was running. You can achieve this effect by writing the information to a file or to a database, but in the spirit of making everything an object, it would be convenient to declare an object as "persistent," and have all the details taken care of for you.

Java's *object serialization* takes any object that implements the `Serializable` interface and turns it into a sequence of bytes that can later regenerate the original object. This is even true across a network, which means that the serialization mechanism automatically compensates for differences in operating systems. That is, you can create an object on a Windows machine, serialize it, and send it across the network to a Unix machine, where it is correctly reconstructed. You don't worry about data representations on the different machines, the byte ordering, or any other details.

Object serialization can implement *lightweight persistence*. Persistence means that an object's lifetime is not determined by whether a program is executing; the object lives *in between* invocations of the program. By taking a serializable object and writing it to disk, then restoring that object when the program is reinvoked, you're able to produce the effect of persistence. The reason it's called "lightweight" is that you can't define an object using some kind of "persistent" keyword and let the system take care of the details. Instead, you must explicitly serialize and deserialize the objects in your program. If you need a more serious persistence mechanism, consider a tool like Hibernate[207].

Object serialization was added to the language to support two major features. Java's *Remote Method Invocation* (RMI) allows objects that live on other machines to behave

[207] http://hibernate.org/

as if they live on your machine. When messages are sent to remote objects, object serialization is necessary to transport the arguments and return values.

Object serialization was also necessary for JavaBeans (considered a failed technology at this writing). When a Bean is used, its state information is generally configured at design time. This state information must be stored and later recovered when the program is started; object serialization performs this task.

Serializing an object is simple as long as the object implements the `Serializable` interface, which is a tagging interface and has no methods. When serialization was added to the language, many standard library classes were changed to make them serializable, including all wrappers for the primitive types, all container classes, and many others. Even `Class` objects can be serialized.

To serialize an object, you create some sort of `OutputStream` object, then wrap it inside an `ObjectOutputStream` object. Now you need only call `writeObject()` and your object is serialized and sent to the `OutputStream` (object serialization is byte-oriented, and thus uses the `InputStream` and `OutputStream` hierarchies). To reverse the process, you wrap an `InputStream` inside an `ObjectInputStream` and call `readObject()`. What comes back is, as usual, a reference to an upcast `Object`, so you must downcast to set things straight.

A particularly clever aspect of object serialization is that it not only saves an image of your object, but it also follows all the references contained in your object and saves *those* objects, and follows all the references in each of those objects, etc. This is sometimes called the "web of objects" that a single object can be connected to, and it includes arrays of references to objects as well as member objects. If you had to maintain your own object serialization scheme, maintaining the code to follow all these links could be mind-boggling. However, Java object serialization seems to pull it off flawlessly, no doubt using an optimized algorithm that traverses the web of objects. The following example tests the serialization mechanism by making a "worm" of linked objects, each of which has a link to the next segment in the worm as well as an array of references to objects of a different class, `Data`:

```
// serialization/Worm.java
// Demonstrates object serialization
import java.io.*;
import java.util.*;

class Data implements Serializable {
  private int n;
  Data(int n) { this.n = n; }
  @Override public String toString() {
    return Integer.toString(n);
  }
}

public class Worm implements Serializable {
  private static Random rand = new Random(47);
  private Data[] d = {
    new Data(rand.nextInt(10)),
    new Data(rand.nextInt(10)),
    new Data(rand.nextInt(10))
  };
  private Worm next;
  private char c;
  // Value of i == number of segments
  public Worm(int i, char x) {
    System.out.println("Worm constructor: " + i);
    c = x;
    if(--i > 0)
      next = new Worm(i, (char)(x + 1));
  }
  public Worm() {
    System.out.println("Zero-argument constructor");
  }
  @Override public String toString() {
    StringBuilder result = new StringBuilder(":");
    result.append(c);
    result.append("(");
    for(Data dat : d)
      result.append(dat);
    result.append(")");
    if(next != null)
      result.append(next);
    return result.toString();
```

```
  }
  public static void
  main(String[] args) throws ClassNotFoundException,
    IOException {
    Worm w = new Worm(6, 'a');
    System.out.println("w = " + w);
    try(
      ObjectOutputStream out = new ObjectOutputStream(
        new FileOutputStream("worm.dat"))
    ) {
      out.writeObject("Worm storage\n");
      out.writeObject(w);
    }
    try(
      ObjectInputStream in = new ObjectInputStream(
        new FileInputStream("worm.dat"))
    ) {
      String s = (String)in.readObject();
      Worm w2 = (Worm)in.readObject();
      System.out.println(s + "w2 = " + w2);
    }
    try(
      ByteArrayOutputStream bout =
        new ByteArrayOutputStream();
      ObjectOutputStream out2 =
        new ObjectOutputStream(bout)
    ) {
      out2.writeObject("Worm storage\n");
      out2.writeObject(w);
      out2.flush();
      try(
        ObjectInputStream in2 = new ObjectInputStream(
          new ByteArrayInputStream(
            bout.toByteArray()))
      ) {
        String s = (String)in2.readObject();
        Worm w3 = (Worm)in2.readObject();
        System.out.println(s + "w3 = " + w3);
      }
    }
  }
}
```

```
/* Output:
Worm constructor: 6
Worm constructor: 5
Worm constructor: 4
Worm constructor: 3
Worm constructor: 2
Worm constructor: 1
w = :a(853):b(119):c(802):d(788):e(199):f(881)
Worm storage
w2 = :a(853):b(119):c(802):d(788):e(199):f(881)
Worm storage
w3 = :a(853):b(119):c(802):d(788):e(199):f(881)
*/
```

Each `Data` object in the array inside `Worm` is initialized with a random number. (This way, you don't suspect the compiler of keeping some kind of meta-information.) Each `Worm` segment is labeled with a `char` that's automatically generated in the process of recursively generating the linked list of `Worm`s. When you create a `Worm`, you tell the constructor how long you want it to be. To make the `next` reference, it calls the `Worm` constructor with a length of one less, etc. The final `next` reference is left as `null`, indicating the end of the `Worm`.

The point of all this was to make something reasonably complex that couldn't easily be serialized. The act of serializing, however, is simple. Once the `ObjectOutputStream` is created from some other stream, `writeObject()` serializes the object. Notice the call to `writeObject()` for a `String`, as well. You can also write all the primitive data types using the same methods as `DataOutputStream` (they share the same interface).

There are two separate code sections that look similar. The first writes and reads a file, and the second, for variety, writes and reads a `ByteArray`. You can read and write an object using serialization to any `DataInputStream` or `DataOutputStream`, including a network.

The output shows that the deserialized object really does contain all links in the original object.

Note that no constructor, not even the zero-argument constructor, is called in the process of deserializing a `Serializable` object. The entire object is restored by recovering data from the `InputStream`.

Finding the Class

You might wonder what's necessary to recover an object from its serialized state. For example, suppose you serialize an object and send it as a file or through a network to another machine. Could a program on the other machine reconstruct the object using only the contents of the file?

The best way to answer this question is (as usual) to perform an experiment. The following file goes in the subdirectory for this chapter:

```
// serialization/Alien.java
// A serializable class
import java.io.*;
public class Alien implements Serializable {}
```

The file that creates and serializes an `Alien` object goes in the same directory:

```
// serialization/FreezeAlien.java
// Create a serialized output file
import java.io.*;

public class FreezeAlien {
  public static void
  main(String[] args) throws Exception {
    try(
      ObjectOutputStream out = new ObjectOutputStream(
        new FileOutputStream("X.file"));
    ) {
      Alien quellek = new Alien();
      out.writeObject(quellek);
    }
  }
}
```

Once the program is compiled and run, it produces a file called `X.file` in the `serialization` directory. The following code is in a subdirectory called `xfiles`:

```
// serialization/xfiles/ThawAlien.java
// Recover a serialized file
// {java serialization.xfiles.ThawAlien}
// {RunFirst: FreezeAlien}
package serialization.xfiles;
import java.io.*;

public class ThawAlien {
  public static void
  main(String[] args) throws Exception {
    ObjectInputStream in = new ObjectInputStream(
      new FileInputStream(new File("X.file")));
    Object mystery = in.readObject();
    System.out.println(mystery.getClass());
  }
}
/* Output:
class Alien
*/
```

To work, the JVM must be able find the associated `.class` file.

Controlling Serialization

The default serialization mechanism is trivial to use. But what about special needs? Perhaps you have special security issues and you don't want to serialize portions of your object, or perhaps it just doesn't make sense for one subobject to be serialized if that part needs creating anew when the object is recovered.

You can control the process of serialization by implementing the `Externalizable` interface instead of the `Serializable` interface. The `Externalizable` interface extends the `Serializable` interface and adds two methods, `writeExternal()` and `readExternal()`. These are automatically called for your object during serialization and deserialization so you can perform your special operations.

The following example shows simple implementations of the `Externalizable` interface methods. Note that `Blip1` and `Blip2` are nearly identical except for a subtle difference (see if you can discover it by looking at the code):

```
// serialization/Blips.java
// Simple use of Externalizable & a pitfall
import java.io.*;

class Blip1 implements Externalizable {
  public Blip1() {
    System.out.println("Blip1 Constructor");
  }
  @Override
  public void writeExternal(ObjectOutput out)
      throws IOException {
    System.out.println("Blip1.writeExternal");
  }
  @Override
  public void readExternal(ObjectInput in)
     throws IOException, ClassNotFoundException {
    System.out.println("Blip1.readExternal");
  }
}

class Blip2 implements Externalizable {
  Blip2() {
    System.out.println("Blip2 Constructor");
  }
  @Override
  public void writeExternal(ObjectOutput out)
      throws IOException {
    System.out.println("Blip2.writeExternal");
  }
  @Override
  public void readExternal(ObjectInput in)
     throws IOException, ClassNotFoundException {
    System.out.println("Blip2.readExternal");
  }
}

public class Blips {
  public static void main(String[] args) {
    System.out.println("Constructing objects:");
    Blip1 b1 = new Blip1();
    Blip2 b2 = new Blip2();
    try(
```

```
      ObjectOutputStream o = new ObjectOutputStream(
        new FileOutputStream("Blips.serialized"))
    ) {
      System.out.println("Saving objects:");
      o.writeObject(b1);
      o.writeObject(b2);
    } catch(IOException e) {
      throw new RuntimeException(e);
    }
    // Now get them back:
    System.out.println("Recovering b1:");
    try(
      ObjectInputStream in = new ObjectInputStream(
        new FileInputStream("Blips.serialized"))
    ) {
      b1 = (Blip1)in.readObject();
    } catch(IOException | ClassNotFoundException e) {
      throw new RuntimeException(e);
    }
    // OOPS! Throws an exception:
    //- System.out.println("Recovering b2:");
    //- b2 = (Blip2)in.readObject();
  }
}
/* Output:
Constructing objects:
Blip1 Constructor
Blip2 Constructor
Saving objects:
Blip1.writeExternal
Blip2.writeExternal
Recovering b1:
Blip1 Constructor
Blip1.readExternal
*/
```

The reason that the `Blip2` object is not recovered is that trying to do so causes an exception. Can you see the difference between `Blip1` and `Blip2`? The constructor for `Blip1` is `public`, while the constructor for `Blip2` is not, and that causes the exception upon recovery. Try making `Blip2`'s constructor `public` and removing the `//-` comments to see the correct results.

When `b1` is recovered, the `Blip1` zero-argument constructor is called. This is different from recovering a `Serializable` object, where the object is constructed entirely from its stored bits, with no constructor calls. With an `Externalizable` object, all the normal default construction behavior occurs (including the initializations at the point of field definition), and *then* `readExternal()` is called. Be aware of this—in particular, the fact that all the default construction always takes place—to produce the correct behavior in your `Externalizable` objects.

Here's an example that shows what you must do to fully store and retrieve an `Externalizable` object:

```
// serialization/Blip3.java
// Reconstructing an externalizable object
import java.io.*;

public class Blip3 implements Externalizable {
  private int i;
  private String s; // No initialization
  public Blip3() {
    System.out.println("Blip3 Constructor");
    // s, i not initialized
  }
  public Blip3(String x, int a) {
    System.out.println("Blip3(String x, int a)");
    s = x;
    i = a;
    // s & i initialized only in non-zero-argument constructor.
  }
  @Override public String toString() { return s + i; }
  @Override public void writeExternal(ObjectOutput out)
  throws IOException {
    System.out.println("Blip3.writeExternal");
    // You must do this:
    out.writeObject(s);
    out.writeInt(i);
  }
  @Override public void readExternal(ObjectInput in)
  throws IOException, ClassNotFoundException {
    System.out.println("Blip3.readExternal");
    // You must do this:
    s = (String)in.readObject();
```

```
    i = in.readInt();
  }
  public static void main(String[] args) {
    System.out.println("Constructing objects:");
    Blip3 b3 = new Blip3("A String ", 47);
    System.out.println(b3);
    try(
      ObjectOutputStream o = new ObjectOutputStream(
        new FileOutputStream("Blip3.serialized"))
    ) {
      System.out.println("Saving object:");
      o.writeObject(b3);
    } catch(IOException e) {
      throw new RuntimeException(e);
    }
    // Now get it back:
    System.out.println("Recovering b3:");
    try(
      ObjectInputStream in = new ObjectInputStream(
        new FileInputStream("Blip3.serialized"))
    ) {
      b3 = (Blip3)in.readObject();
    } catch(IOException | ClassNotFoundException e) {
      throw new RuntimeException(e);
    }
    System.out.println(b3);
  }
}
/* Output:
Constructing objects:
Blip3(String x, int a)
A String 47
Saving object:
Blip3.writeExternal
Recovering b3:
Blip3 Constructor
Blip3.readExternal
A String 47
*/
```

The fields `s` and `i` are initialized only in the second constructor, but not in the zero-argument constructor. This means if you don't initialize `s` and `i` in `readExternal()`,

`s` is `null` and `i` is zero (the storage for the object gets wiped to zero in the first step of object creation). If you comment out the two lines of code following the phrases "You must do this:" and run the program, you'll see that when the object is recovered, `s` is `null` and `i` is zero.

If you inherit an `Externalizable` object, you'll typically call the base-class versions of `writeExternal()` and `readExternal()` to provide proper storage and retrieval of the base-class components.

So to make things work correctly, you must not only write the important data from the object during the `writeExternal()` method (there is no default behavior that writes any of the member objects for an `Externalizable` object), but you must also recover that data in the `readExternal()` method. This can be a bit confusing at first because the default construction behavior for an `Externalizable` object can make it seem like some kind of storage and retrieval takes place automatically. It does not.

The `transient` Keyword

When you're controlling serialization, there might be a particular subobject you don't want Java's serialization mechanism to automatically save and restore. This is commonly the case if that subobject represents sensitive information you don't want to serialize, such as a password. Even if that information is `private` in the object, once it is serialized, it's possible for someone to access it by reading a file or intercepting a network transmission.

One way to prevent sensitive parts of your object from being serialized is to implement your class as `Externalizable`, as shown previously. Then nothing is automatically serialized, and you can explicitly serialize only the necessary parts inside `writeExternal()`.

If you're working with a `Serializable` object, however, all serialization happens automatically. To control this, you can turn off serialization on a field-by-field basis using the `transient` keyword, which says, "Don't bother saving or restoring this—I'll take care of it."

For example, consider a `Logon` object that keeps information about a particular login session. Suppose that, once you verify the login, you want to store the data, but without the password. The easiest way to do this is by implementing `Serializable` and marking the `password` field as `transient`. Here's what it looks like:

```
// serialization/Logon.java
// Demonstrates the "transient" keyword
import java.util.concurrent.*;
import java.io.*;
import java.util.*;
import onjava.Nap;

public class Logon implements Serializable {
  private Date date = new Date();
  private String username;
  private transient String password;
  public Logon(String name, String pwd) {
    username = name;
    password = pwd;
  }
  @Override public String toString() {
    return "logon info: \n   username: " +
      username + "\n   date: " + date +
      "\n   password: " + password;
  }
  public static void main(String[] args) {
    Logon a = new Logon("Hulk", "myLittlePony");
    System.out.println("logon a = " + a);
    try(
      ObjectOutputStream o =
        new ObjectOutputStream(
          new FileOutputStream("Logon.dat"))
    ) {
      o.writeObject(a);
    } catch(IOException e) {
      throw new RuntimeException(e);
    }
    new Nap(1);
    // Now get them back:
    try(
      ObjectInputStream in = new ObjectInputStream(
        new FileInputStream("Logon.dat"))
    ) {
      System.out.println(
        "Recovering object at " + new Date());
      a = (Logon)in.readObject();
    } catch(IOException | ClassNotFoundException e) {
```

```
      throw new RuntimeException(e);
    }
    System.out.println("logon a = " + a);
  }
}
/* Output:
logon a = logon info:
   username: Hulk
   date: Sun Jan 24 08:49:30 MST 2021
   password: myLittlePony
Recovering object at Sun Jan 24 08:49:31 MST 2021
logon a = logon info:
   username: Hulk
   date: Sun Jan 24 08:49:30 MST 2021
   password: null
*/
```

The `date` and `username` fields are ordinary (not `transient`), and thus are automatically serialized. However, the `password` is `transient`, so it is not stored to disk; also, the serialization mechanism makes no attempt to recover it. When the object is recovered, the `password` field is `null`. Note that while `toString()` assembles a `String` object using the overloaded `+` operator, a `null` reference is automatically converted to the `String` "null."

You can also see that the `date` field is stored to and recovered from disk and not generated anew.

`Externalizable` objects do not store any of their fields by default so the `transient` keyword is for use with `Serializable` objects only.

An Alternative to `Externalizable`

If you're not keen on implementing the `Externalizable` interface, there's another approach. You can implement the `Serializable` interface and *add* (notice I say "add" and not "override" or "implement") methods called `writeObject()` and `readObject()` that are automatically called when the object is serialized and deserialized, respectively. That is, if you provide these two methods, they are used instead of the default serialization.

The methods must have these exact signatures:

```
private void writeObject(ObjectOutputStream stream)
throws IOException;

private void readObject(ObjectInputStream stream)
throws IOException, ClassNotFoundException
```

From a design standpoint, things get really weird here. First of all, you might think that, because these methods are not part of a base class or the `Serializable` interface, they ought to be defined in their own interface(s). But notice they are defined as `private`, which means they are called only by other members of this class. However, you don't actually call them from other members of this class, but instead the `writeObject()` and `readObject()` methods of the `ObjectOutputStream` and `ObjectInputStream` objects call your object's `writeObject()` and `readObject()` methods. (Notice my tremendous restraint in not launching into a long diatribe on using the same method names here. In a word: confusing.) You might wonder how the `ObjectOutputStream` and `ObjectInputStream` objects have access to `private` methods of your class. We can only assume this is part of the serialization magic.[208]

Anything defined in an interface is automatically `public`, so if `writeObject()` and `readObject()` must be `private`, then they can't be part of an interface. You must follow the signatures exactly so the effect is the same as if you're implementing an interface.

It would appear that when you call `ObjectOutputStream.writeObject()`, the `Serializable` object you pass it to is interrogated (using reflection, no doubt) to see if it implements its own `writeObject()`. If so, the normal serialization process is skipped and the custom `writeObject()` is called. The same situation exists for `readObject()`.

There's one other twist. Inside your `writeObject()`, you can choose to perform the default `writeObject()` action by calling `defaultWriteObject()`. Likewise, inside `readObject()` you can call `defaultReadObject()`. Here is a simple example that demonstrates how you can control the storage and retrieval of a `Serializable` object:

[208] The section "Interfaces and type information" at the end of the Reflection chapter shows how it's possible to access `private` methods from outside of the class.

```
// serialization/SerialCtl.java
// Controlling serialization by adding your own
// writeObject() and readObject() methods
import java.io.*;

public class SerialCtl implements Serializable {
  private String a;
  private transient String b;
  public SerialCtl(String aa, String bb) {
    a = "Not Transient: " + aa;
    b = "Transient: " + bb;
  }
  @Override public String toString() {
    return a + "\n" + b;
  }
  private void writeObject(ObjectOutputStream stream)
  throws IOException {
    stream.defaultWriteObject();
    stream.writeObject(b);
  }
  private void readObject(ObjectInputStream stream)
  throws IOException, ClassNotFoundException {
    stream.defaultReadObject();
    b = (String)stream.readObject();
  }
  public static void main(String[] args) {
    SerialCtl sc = new SerialCtl("Test1", "Test2");
    System.out.println("Before:\n" + sc);
    try (
      ByteArrayOutputStream buf =
        new ByteArrayOutputStream();
      ObjectOutputStream o =
        new ObjectOutputStream(buf);
    ) {
      o.writeObject(sc);
      // Now get it back:
      try (
        ObjectInputStream in =
          new ObjectInputStream(
            new ByteArrayInputStream(
              buf.toByteArray()));
      ) {
```

```
        SerialCtl sc2 = (SerialCtl)in.readObject();
        System.out.println("After:\n" + sc2);
      }
    } catch(IOException | ClassNotFoundException e) {
      throw new RuntimeException(e);
    }
  }
}
/* Output:
Before:
Not Transient: Test1
Transient: Test2
After:
Not Transient: Test1
Transient: Test2
*/
```

In this example, one `String` field is ordinary and the other is `transient`, to prove that the non-`transient` field is saved by the `defaultWriteObject()` method and the `transient` field is saved and restored explicitly. The fields are initialized inside the constructor rather than at the point of definition to prove they are not initialized by some automatic mechanism during deserialization.

If you use the default mechanism to write the non-`transient` parts of your object, you must call `defaultWriteObject()` as the first operation in `writeObject()`, and `defaultReadObject()` as the first operation in `readObject()`. These are strange method calls. It would appear, for example, that you are calling `defaultWriteObject()` for an `ObjectOutputStream` and passing it no arguments, and yet it somehow turns around and knows the reference to your object and how to write all the non-`transient` parts. Spooky.

The storage and retrieval of the `transient` objects uses more familiar code. And yet, think about what happens here. In `main()`, a `SerialCtl` object is created and serialized to an `ObjectOutputStream`. (Notice here that a buffer is used instead of a file—it's all the same to the `ObjectOutputStream`.) The serialization occurs in the line:

```
o.writeObject(sc);
```

The `writeObject()` method must be examining `sc` to see if it has its own `writeObject()` method. (Not by checking the interface—there isn't one—or the class type,

but by actually hunting for the method using reflection.) If it does, it uses that. A similar approach holds true for `readObject()`. Perhaps this was the only practical way they could solve the problem, but it's certainly strange.

Versioning

You might change the version of a serializable class (objects of the original class might be stored in a database, for example). This is supported, but you'll probably do it only in special cases, and it requires an extra depth of understanding that we will not attempt to achieve here. The JDK documents[209] cover this topic thoroughly.

Using Persistence

It's appealing to use serialization technology to store some of the state of your program to easily restore the program to the current state later. But before you can do this, some questions must be answered. What happens if you serialize two objects that both have a reference to a third object? When you restore those two objects from their serialized state, do you get only one occurrence of the third object? What if you serialize your two objects to separate files and deserialize them in different parts of your code?

Here's an example that shows the problem:

```
// serialization/MyWorld.java
import java.io.*;
import java.util.*;

class House implements Serializable {}

class Animal implements Serializable {
  private String name;
  private House preferredHouse;
  Animal(String nm, House h) {
    name = nm;
    preferredHouse = h;
  }
```

[209] http://java.oracle.com

```
  @Override public String toString() {
    return name + "[" + super.toString() +
      "], " + preferredHouse + "\n";
  }
}

public class MyWorld {
  public static void main(String[] args) {
    House house = new House();
    List<Animal> animals = new ArrayList<>();
    animals.add(
      new Animal("Bosco the dog", house));
    animals.add(
      new Animal("Ralph the hamster", house));
    animals.add(
      new Animal("Molly the cat", house));
    System.out.println("animals: " + animals);
    try(
      ByteArrayOutputStream buf1 =
        new ByteArrayOutputStream();
      ObjectOutputStream o1 =
        new ObjectOutputStream(buf1)
    ) {
      o1.writeObject(animals);
      o1.writeObject(animals); // Write a 2nd set
      // Write to a different stream:
      try(
        ByteArrayOutputStream buf2 =
          new ByteArrayOutputStream();
        ObjectOutputStream o2 =
          new ObjectOutputStream(buf2)
      ) {
        o2.writeObject(animals);
        // Now get them back:
        try(
          ObjectInputStream in1 =
            new ObjectInputStream(
              new ByteArrayInputStream(
                buf1.toByteArray()));
          ObjectInputStream in2 =
            new ObjectInputStream(
              new ByteArrayInputStream(
```

```
                buf2.toByteArray()))
        ) {
          List
            animals1 = (List)in1.readObject(),
            animals2 = (List)in1.readObject(),
            animals3 = (List)in2.readObject();
          System.out.println(
            "animals1: " + animals1);
          System.out.println(
            "animals2: " + animals2);
          System.out.println(
            "animals3: " + animals3);
        }
      }
    } catch(IOException | ClassNotFoundException e) {
      throw new RuntimeException(e);
    }
  }
}
/* Output:
animals: [Bosco the dog[Animal@19e0bfd], House@139a55
, Ralph the hamster[Animal@1db9742], House@139a55
, Molly the cat[Animal@106d69c], House@139a55
]
animals1: [Bosco the dog[Animal@1ee12a7], House@10bedb4
, Ralph the hamster[Animal@103dbd3], House@10bedb4
, Molly the cat[Animal@167cf4d], House@10bedb4
]
animals2: [Bosco the dog[Animal@1ee12a7], House@10bedb4
, Ralph the hamster[Animal@103dbd3], House@10bedb4
, Molly the cat[Animal@167cf4d], House@10bedb4
]
animals3: [Bosco the dog[Animal@a987ac], House@a3a380
, Ralph the hamster[Animal@1453f44], House@a3a380
, Molly the cat[Animal@ad8086], House@a3a380
]
*/
```

It's possible to use object serialization to and from a `byte` array as a way of doing a *deep copy* of any object that's `Serializable`. (A deep copy means you're duplicating the entire web of objects, rather than just the basic object and its references.) Object copying is covered in depth in the Appendix: Passing and Returning Objects.

`Animal` objects contain fields of type `House`. In `main()`, a `List` of these `Animals` is created and serialized, twice to one stream, then again to a separate stream. When these are deserialized and printed, you see the output shown for one run (the objects are in different memory locations for each run).

You might expect that the deserialized objects have different addresses from their originals. But notice that in `animals1` and `animals2`, the same addresses appear, including the references to the `House` object that both share. On the other hand, when `animals3` is recovered, the system has no way of knowing that the objects in this other stream are aliases of the objects in the first stream, so it makes a completely different web of objects.

As long as you're serializing everything to a single stream, you'll recover the same web of objects you wrote, with no accidental duplication of objects. You can change the state of your objects in between the time you write the first and the last, but that's your responsibility; the objects are written in their current state (and with whatever connections they have to other objects) at the time you serialize them.

The safest way to save the state of a system is to "atomically" serialize that state. If you serialize some things, do some other work, and serialize some more, etc., then you will not store the system safely. Instead, put all the objects that comprise the state of your system in a single container and write that container out in one operation. Then you can restore it with a single method call.

The following example is an imaginary computer-aided design (CAD) system that demonstrates the approach. In addition, it throws in the issue of `static` fields; if you look at the JDK documentation, you'll see that `Class` is `Serializable`, so it should be easy to store the `static` fields by serializing the `Class` object. That seems like a sensible approach, anyway.

```
// serialization/AStoreCADState.java
// Saving the state of a fictitious CAD system
import java.io.*;
import java.util.*;
import java.util.stream.*;

enum Color { RED, BLUE, GREEN }

abstract class Shape implements Serializable {
  private int xPos, yPos, dimension;
  private static Random rand = new Random(47);
  private static int counter = 0;
  public abstract void setColor(Color newColor);
  public abstract Color getColor();
  Shape(int xVal, int yVal, int dim) {
    xPos = xVal;
    yPos = yVal;
    dimension = dim;
  }
  @Override public String toString() {
    return getClass() + "Color[" + getColor() +
      "] xPos[" + xPos + "] yPos[" + yPos +
      "] dim[" + dimension + "]\n";
  }
  public static Shape randomFactory() {
    int xVal = rand.nextInt(100);
    int yVal = rand.nextInt(100);
    int dim = rand.nextInt(100);
    switch(counter++ % 3) {
      default:
      case 0: return new Circle(xVal, yVal, dim);
      case 1: return new Square(xVal, yVal, dim);
      case 2: return new Line(xVal, yVal, dim);
    }
  }
}

class Circle extends Shape {
  private static Color color = Color.RED;
  Circle(int xVal, int yVal, int dim) {
    super(xVal, yVal, dim);
  }
```

```
  @Override public void setColor(Color newColor) {
    color = newColor;
  }
  @Override public Color getColor() { return color; }
}

class Square extends Shape {
  private static Color color = Color.RED;
  Square(int xVal, int yVal, int dim) {
    super(xVal, yVal, dim);
  }
  @Override public void setColor(Color newColor) {
    color = newColor;
  }
  @Override public Color getColor() { return color; }
}

class Line extends Shape {
  private static Color color = Color.RED;
  public static void
  serializeStaticState(ObjectOutputStream os)
  throws IOException { os.writeObject(color); }
  public static void
  deserializeStaticState(ObjectInputStream os)
  throws IOException, ClassNotFoundException {
    color = (Color)os.readObject();
  }
  Line(int xVal, int yVal, int dim) {
    super(xVal, yVal, dim);
  }
  @Override public void setColor(Color newColor) {
    color = newColor;
  }
  @Override public Color getColor() { return color; }
}

public class AStoreCADState {
  public static void main(String[] args) {
    List<Class<? extends Shape>> shapeTypes =
      Arrays.asList(
        Circle.class, Square.class, Line.class);
    List<Shape> shapes = IntStream.range(0, 10)
```

```
      .mapToObj(i -> Shape.randomFactory())
      .collect(Collectors.toList());
    // Set all the static colors to GREEN:
    shapes.forEach(s -> s.setColor(Color.GREEN));
    // Save the state vector:
    try(
      ObjectOutputStream out =
        new ObjectOutputStream(
          new FileOutputStream("CADState.dat"))
    ) {
      out.writeObject(shapeTypes);
      Line.serializeStaticState(out);
      out.writeObject(shapes);
    } catch(IOException e) {
      throw new RuntimeException(e);
    }
    // Display the shapes:
    System.out.println(shapes);
  }
}
/* Output:
[class CircleColor[GREEN] xPos[58] yPos[55] dim[93]
, class SquareColor[GREEN] xPos[61] yPos[61] dim[29]
, class LineColor[GREEN] xPos[68] yPos[0] dim[22]
, class CircleColor[GREEN] xPos[7] yPos[88] dim[28]
, class SquareColor[GREEN] xPos[51] yPos[89] dim[9]
, class LineColor[GREEN] xPos[78] yPos[98] dim[61]
, class CircleColor[GREEN] xPos[20] yPos[58] dim[16]
, class SquareColor[GREEN] xPos[40] yPos[11] dim[22]
, class LineColor[GREEN] xPos[4] yPos[83] dim[6]
, class CircleColor[GREEN] xPos[75] yPos[10] dim[42]
]
*/
```

The `Shape` class `implements Serializable`, so anything that inherits `Shape` is automatically `Serializable` as well. Each `Shape` contains data, and each derived `Shape` class contains a `static` field that holds the color of all of those types of `Shapes`. (Placing a `static` field in the base class would result in only one field because `static` fields are not duplicated in derived classes.) Methods in the base class can be overridden to set the color for the various types (`static` methods are not dynamically bound, so these are normal methods). The `randomFactory()` method

creates a different `Shape` each time you call it, using random values for the `Shape` data.

`Circle` and `Square` are straightforward extensions of `Shape`; the only difference is that `Circle` initializes `color` at the point of definition and `Square` initializes it in the constructor. We'll leave the discussion of `Line` for later.

In `main()`, one `ArrayList` is used to hold the `Class` objects and the other to hold the shapes.

Recovering the objects is fairly straightforward:

```
// serialization/RecoverCADState.java
// Restoring the state of the fictitious CAD system
// {RunFirst: AStoreCADState}
import java.io.*;
import java.util.*;

public class RecoverCADState {
  @SuppressWarnings("unchecked")
  public static void main(String[] args) {
    try(
      ObjectInputStream in =
        new ObjectInputStream(
          new FileInputStream("CADState.dat"))
    ) {
      // Read in the same order they were written:
      List<Class<? extends Shape>> shapeTypes =
        (List<Class<? extends Shape>>)in.readObject();
      Line.deserializeStaticState(in);
      List<Shape> shapes =
        (List<Shape>)in.readObject();
      System.out.println(shapes);
    } catch(IOException | ClassNotFoundException e) {
      throw new RuntimeException(e);
    }
  }
}
/* Output:
[class CircleColor[RED] xPos[58] yPos[55] dim[93]
, class SquareColor[RED] xPos[61] yPos[61] dim[29]
, class LineColor[GREEN] xPos[68] yPos[0] dim[22]
, class CircleColor[RED] xPos[7] yPos[88] dim[28]
```

```
, class SquareColor[RED] xPos[51] yPos[89] dim[9]
, class LineColor[GREEN] xPos[78] yPos[98] dim[61]
, class CircleColor[RED] xPos[20] yPos[58] dim[16]
, class SquareColor[RED] xPos[40] yPos[11] dim[22]
, class LineColor[GREEN] xPos[4] yPos[83] dim[6]
, class CircleColor[RED] xPos[75] yPos[10] dim[42]
]
*/
```

The values of `xPos`, `yPos`, and `dim` were all stored and recovered successfully, but there's something wrong with the retrieval of the `static` information. It's all "3" going in, but it doesn't come out that way. `Circles` have a value of `1` (`RED`, which is the definition), and `Squares` have a value of `0` (remember, they are initialized in the constructor). It's as if the `statics` didn't get serialized at all! That's right—even though class `Class` is `Serializable`, it doesn't do what you expect. So to serialize `statics`, you must do it yourself.

This is the purpose of the `serializeStaticState()` and `deserializeStaticState()` `static` methods in `Line`. They are explicitly called as part of the storage and retrieval process. (Note that the order of writing to the serialize file and reading back from it must be maintained.) Thus to make these programs run correctly, you must:

1. Add a `serializeStaticState()` and `deserializeStaticState()` to the shapes.
2. Remove the `ArrayList shapeTypes` and all code related to it.
3. Add calls to the new serialize and deserialize static methods in the shapes.

Another issue you might think about is security because serialization also saves `private` data. For security, those fields should be marked as `transient`. But then you must design a secure way to store that information so when you do a restore, you can reset those `private` variables.

XML

An important limitation of object serialization is it is a Java-only solution: Only Java programs can deserialize such objects. A more interoperable solution is to convert

data to XML format, which allows it to be consumed by a large variety of platforms and languages.

Because of its popularity, there are a confusing number of options for programming with XML, including the `javax.xml.*` libraries distributed with the JDK. I've chosen to use Elliotte Rusty Harold's open-source XOM library (downloads and documentation at www.xom.nu[210]) because it seems the simplest and most straightforward way to produce and modify XML using Java. In addition, XOM emphasizes XML correctness.

As an example, suppose you have `APerson` objects containing first and last names that you'd like to serialize into XML. The following `APerson` class has a `getXML()` method that uses XOM to produce the `APerson` data converted to an XML `Element` object, and a constructor that takes an `Element` and extracts the appropriate `APerson` data (notice that the XML examples are in their own subdirectory):

```
// serialization/APerson.java
// Use the XOM library to write and read XML
// nu.xom.Node comes from http://www.xom.nu
import nu.xom.*;
import java.io.*;
import java.util.*;

public class APerson {
  private String first, last;
  public APerson(String first, String last) {
    this.first = first;
    this.last = last;
  }
  // Produce an XML Element from this APerson object:
  public Element getXML() {
    Element person = new Element("person");
    Element firstName = new Element("first");
    firstName.appendChild(first);
    Element lastName = new Element("last");
    lastName.appendChild(last);
    person.appendChild(firstName);
    person.appendChild(lastName);
    return person;
  }
```

[210] http://www.xom.nu/

```
  // Constructor restores a APerson from XML:
  public APerson(Element person) {
    first = person
      .getFirstChildElement("first").getValue();
    last = person
      .getFirstChildElement("last").getValue();
  }
  @Override public String toString() {
    return first + " " + last;
  }
  // Make it human-readable:
  public static void
  format(OutputStream os, Document doc)
  throws Exception {
    Serializer serializer =
      new Serializer(os,"ISO-8859-1");
    serializer.setIndent(4);
    serializer.setMaxLength(60);
    serializer.write(doc);
    serializer.flush();
  }
  public static void
  main(String[] args) throws Exception {
    List<APerson> people = Arrays.asList(
      new APerson("Dr. Bunsen", "Honeydew"),
      new APerson("Gonzo", "The Great"),
      new APerson("Phillip J.", "Fry"));
    System.out.println(people);
    Element root = new Element("people");
    for(APerson p : people)
      root.appendChild(p.getXML());
    Document doc = new Document(root);
    format(System.out, doc);
    format(new BufferedOutputStream(
      new FileOutputStream("People.xml")), doc);
  }
}
/* Output:
[Dr. Bunsen Honeydew, Gonzo The Great, Phillip J. Fry]
<?xml version="1.0" encoding="ISO-8859-1"?>
<people>
    <person>
```

```
        <first>Dr. Bunsen</first>
        <last>Honeydew</last>
    </person>
    <person>
        <first>Gonzo</first>
        <last>The Great</last>
    </person>
    <person>
        <first>Phillip J.</first>
        <last>Fry</last>
    </person>
</people>
*/
```

The XOM methods are fairly self-explanatory and are found in the XOM documentation.

XOM also contains a `Serializer` class you see used in the `format()` method to turn the XML into a more readable form. If you just call `toXML()` you'll get everything run together, so the `Serializer` is a convenient tool.

Deserializing `APerson` objects from an XML file is also simple:

```
// serialization/People.java
// nu.xom.Node comes from http://www.xom.nu
// {RunFirst: APerson}
import nu.xom.*;
import java.io.File;
import java.util.*;

public class People extends ArrayList<APerson> {
  public People(String fileName) throws Exception  {
    Document doc =
      new Builder().build(new File(fileName));
    Elements elements =
      doc.getRootElement().getChildElements();
    for(int i = 0; i < elements.size(); i++)
      add(new APerson(elements.get(i)));
  }
  public static void
  main(String[] args) throws Exception {
    People p = new People("People.xml");
```

```
    System.out.println(p);
  }
}
/* Output:
[Dr. Bunsen Honeydew, Gonzo The Great, Phillip J. Fry]
*/
```

The `People` constructor opens and reads a file using XOM's `Builder.build()` method, and the `getChildElements()` method produces an `Elements` list (not a standard Java `List`, but an object that only has a `size()` and `get()` method—Harold did not want to force people to use a particular version of Java, but still wanted a type-safe container). Each `Element` in this list represents a `APerson` object, so it is handed to the second `APerson` constructor. Note this requires you know ahead of time the exact structure of your XML file, but this is often true with these kinds of problems. If the structure doesn't match what you expect, XOM will throw an exception. It's also possible for you to write more complex code to explore the XML document, rather than making assumptions about it, for cases when you have less concrete information about incoming XML structure.

To get these examples to compile, put the JAR files from the XOM distribution into your classpath.

This has only been a brief introduction to XML programming with Java and the XOM library; for more information see www.xom.nu[211].

[211] http://www.xom.nu/

Appendix: Benefits and Costs of Static Type Checking

> This is an edited collection of essays I've written over the years that tries to put the debate between statically-checked and dynamic languages into perspective, with two forewords describing my recent insights on the topic.

Foreword to the 2021 Edition

Since this book was originally published I have experienced two developments in the world of static type checking:

1. In the process of writing Atomic Kotlin[212], I've become aware of the impact of declaration style; in particular, the difference between "Algol style" that you see in C, C++, Java and similar languages, and what I'll call "Haskell style" which is used in languages like Scala, Kotlin, and more recently Python. Not only is this style (I think) more readable, it also produces cleaner type inference.
2. Python version 3.5 added "type hints," which is a form of optional static typing. I found that the Haskell-style type-declaration syntax had virtually no negative impact but the benefits are quite notable, including better checking and suggestions by development environments. When, for example, PyCharm sees a type declaration, it can tell me all the things I can do with that object, and this saves time. It can also tell me at edit-time that I'm using an object incorrectly, and that also saves time. The syntax is easy to add and it improves my life as a Python programmer.

I now think that my questions and grumbles about static type checking have been because of the impact of the syntax—even with the improvements demonstrated in this book, Java syntax still contains significantly more "visual noise" than I care for.

[212] https://www.atomickotlin.com/

Because of this, I thought of taking this appendix out entirely, but there were already some foreign-language publishers who had negotiated for the book as-is and I was reluctant to change it for that reason. As a compromise, I decided to add this foreword.

When reading this appendix, please remember that these not my current thoughts—they are the thoughts of previous versions of me, and have changed over time. My present position is that if the syntax is low-noise and the process is unobtrusive, I prefer the benefits of static type checking.

Foreword to the Original Edition

My primary interest in software development has always been programmer productivity. Programmer cycles are expensive, CPU cycles are cheap, and I believe we should not pay for the latter with the former.

My initial experiences with static type checking were first with assembly language (no type checking at all), followed by pre-ANSI C, which allowed many problems to slip through. Then C++ appeared and checked the type of function arguments and return values, and thus found many errors (and influenced ANSI C to adopt this approach). This error discovery made me a big fan. Time passed, and we were burdened with more and more coding overhead to support more type checking. In the meantime, I began experimenting with dynamic languages, and eventually discovered Python and its marvelous productivity. This made me start to wonder if we weren't missing something in our pursuit of static perfection. (I have also ventured into the extremes of static type checking while writing Atomic Scala[213]).

I tended to hear things like "more static checking is always better" and "you can only solve simple problems with dynamic languages; they aren't suited to real-world applications."

My goal in writing these essays was to explain the benefits of dynamic languages and to question the idea that static type checking was "free," that there are no costs associated with all those features. And that sometimes those costs hold us back.

My assumption throughout was always that people simply didn't understand, and that if I just explained harder, they would eventually get it.

[213] http://www.AtomicScala.com

Then something interesting happened. Time passed, and people created more and more large, real-world applications using languages like Python and Ruby. Dynamic languages like Groovy and Clojure appeared for the JVM, and folks began writing impressive programs with these.

After seeing what these languages could do, there were no remaining legitimate arguments to justify the idea that certain things can only be accomplished with static languages. At that point, some people started saying they just "preferred static type checking."

Only recently did it hit me:

> This is a cultural issue, an issue of belief.

That's why all my explanations kept failing. A group of people had gotten comfortable with a set of beliefs, and I was challenging those and making them uncomfortable. So they fought back, and dug in harder.

> Programming languages create cultural communities.

With this new perspective, which seems obvious in hindsight, the issues and choices now look quite different to me. Some issues can be argued with reason and experimentation. But other issues are part of the core belief system of a community, and if you join, you sign up for those and go where they take you.

Static Type Checking vs. Testing

How can we get maximal leverage on the problems we try to solve? Whenever a new tool (especially a programming language) appears, that tool provides some kind of abstraction that may or may not hide needless detail from the programmer. I have come, however, to always be on watch for a Faustian bargain, especially one that tries to convince me to ignore all the hoops I must jump through to achieve this abstraction. Perl is an excellent example of this—the immediacy of the language hides the meaningless details of writing a program, but the unreadable syntax (based, I know, on backward-compatibility with Unix tools like awk, sed and grep) is a significant price to pay.

Over the years, I've slowly come to understand the relationship of more traditional programming languages and their orientation towards static type checking. This began, long ago, with a two-month love affair with Perl, which gave me productivity through rapid turnaround (The affair was terminated because of Perl's reprehensible treatment of references and classes; only later did I see the real problems with the syntax). Issues of static-vs-dynamic type checking were not visible with Perl because you can't build projects large enough to see these issues, and the syntax obscures everything in smaller programs.

After I worked with Python[214]—a language which can build large, complex systems—I noticed that, despite an apparent carelessness about type checking, Python programs seemed to work quite well without much effort, and without the kinds of problems expected from a language that doesn't have the static type checking we've all come to "know" is the only correct way of solving the programming problem.

This became a puzzle to me: if static type checking is so important, why are people able to build big, complex Python programs (in much shorter time and with much less effort than the static counterparts) without the disaster I was so sure would ensue?

This shook my unquestioning acceptance of static type checking (acquired when moving from C to C++, where the improvement was dramatic) enough that the next time I examined the issue of checked exceptions in Java, I asked "why"? which produced a big discussion wherein I was told that if I kept advocating unchecked exceptions, cities would fall and civilization as we know it would cease to exist. In *Thinking in Java*, 3rd edition, I advocated the use of `RuntimeException` as a wrapper class to "turn off" checked exceptions. Every time I do it now, it seems right (I note that Martin Fowler came up with the same idea at roughly the same time), but I still get the occasional email that warns me I am violating all that is right and true, civilizations will fall, etc.

But deciding that checked exceptions seem like more trouble than they're worth[215] did not answer the question "why does Python work so well, when conventional wisdom says it should produce massive failures?" Python and similar dynamically-typed languages are lazy about type checking. Instead of putting the strongest possible constraints upon the type of objects, as early as possible (as C++ and Java do), languages like Ruby, SmallTalk and Python put the loosest possible constraints

[214] www.Python.org

[215] The checking, not the exception. I believe that a single, consistent error reporting mechanism is essential, although in the end error reporting might not actually be conflated with exceptions—see the Go language, for example.

on types, and evaluate types only if they have to. That is, you can send any message to any object, and the language only cares that the object can accept the message—it doesn't require that the object be a particular type, as Java and C++ do. For example, for speaking pets in Java, the code looks like this:

```
// staticchecking/petspeak/PetSpeak.java
// Speaking pets in Java
// {java staticchecking.petspeak.PetSpeak}
package staticchecking.petspeak;

interface Pet {
  void speak();
}

class Cat implements Pet {
  @Override public void speak() {
    System.out.println("meow!");
  }
}

class Dog implements Pet {
  @Override public void speak() {
    System.out.println("woof!");
  }
}

public class PetSpeak {
  static void command(Pet p) { p.speak(); }
  public static void main(String[] args) {
    Pet[] pets = { new Cat(), new Dog() };
    for(Pet pet : pets)
      command(pet);
  }
}
/* Output:
meow!
woof!
*/
```

Note that `command()` must know exactly the type of argument it's going to accept—a `Pet`—and it will accept nothing else. Thus, I must create a hierarchy of `Pet`, and inherit `Dog` and `Cat` so I can upcast them to the generic `command()` method.

For the longest time, I assumed that upcasting was an inherent part of object-oriented programming, and found the questions about upcasting from SmallTalkers and the like to be annoying. But when I started working with Python I found the following curiosity. The above code can be translated directly into Python:

```
# staticchecking/PetSpeak.py
#- Speaking pets in Python

class Pet:
    def speak(self): pass

class Cat(Pet):
    def speak(self):
       print("meow!")

class Dog(Pet):
    def speak(self):
       print("woof!")

def command(pet):
    pet.speak()

pets = [ Cat(), Dog() ] # (1)
for pet in pets: # (2)
    command(pet)

output = """
meow!
woof!
"""
```

If you've never seen Python before, you'll notice it redefines the meaning of a terse language, but in a very good way. You think C/C++ is terse? Let's throw away those curly braces. Indentation already has meaning to the human mind, so we'll use that instead to indicate scope. Argument types and return types? Let the language sort it out. During class creation, base classes are indicated in parentheses. `def` means we are creating a function or method definition. On the other hand, Python is explicit about the `this` argument (called `self` by convention) for method definitions.

Comments start with a `#` and continue to the end of the line.

The definition for the `Pet` class uses the `pass` keyword, which is similar to saying `abstract` in Java. It just means there's no definition here.

Note that `command(pet)` just says it takes some object called `pet`, but it doesn't give any information about what the type of that object must be. That's because it doesn't care, as long as you can call `speak()`, or whatever else your function or method wants to do. We'll look at this more closely in a minute.

Also, `command(pet)` is just an ordinary function, which is OK in Python. That is, Python doesn't insist you make everything an object because sometimes a function is what you want.

- **[1]** In Python, lists and maps/dictionaries/associative arrays are both so important they are built into the core of the language, so I don't need to import any special library to use them. Here, a list is created containing two new objects, of type `Cat` and `Dog`. The constructors are called, but no `new` is necessary (and now you'll go back to Java and realize that no `new` is necessary there, either. It's just a redundancy inherited from C++).
- **[2]** Iterating through a sequence is also important enough that it's a native operation in Python. The `for` selects each item from the list `pets` into the variable `pet`.

The output is the same as the Java version, and is captured in the definition of the `output` variable. Triple quoting creates a multiline string.

Python makes an excellent pseudo-coding language, with the wonderful attribute that it can actually be executed. This means you can quickly try out ideas in Python, and when you get one that works, you can rewrite it in Java, C++ or your language of choice. Or maybe you realize the problem is already solved in Python, so why bother rewriting it? (That's usually as far as I get). I've taken to giving exercise hints in Python during seminars, because then I'm not giving away the whole picture, but people can see the form I'm looking for in a solution so they can move ahead. And I'm able to verify that the form is correct. This is why Python is often called "executable pseudocode."

For this discussion, the interesting part is this: because the `command(pet)` method doesn't care about the type it's getting, I don't have to upcast. So I can rewrite the Python program without using base classes:

```
# staticchecking/NoBasePetSpeak.py
#- Speaking pets without base classes

class Cat:
    def speak(self):
       print("meow!")

class Dog:
    def speak(self):
       print("woof!")

class Bob:
    def bow(self):
       print("thank you, thank you!")
    def speak(self):
       print("Welcome to the neighborhood!")
    def drive(self):
       print("beep, beep!")

def command(pet):
    pet.speak()

pets = [ Cat(), Dog(), Bob() ]

for pet in pets:
    command(pet)

output = """
meow!
woof!
Welcome to the neighborhood!
"""
```

`command(pet)` only cares whether it can send the `speak()` message to its argument so I've removed the base class `Pet`, and even added a totally non-pet class called `Bob` which happens to have a `speak()` method, so it also works in the `command(pet)` function.

At this point, a statically-typed language sputters with rage, insisting this kind of sloppiness will cause disaster and mayhem. Clearly, at some point the "wrong" type will be used with `command()` or will otherwise slip through the system. The benefit

of simpler, clearer expression of concepts is simply not worth the danger. Even if that benefit is a productivity increase of 5 to 10 times over that of Java or C++.

What happens when such a problem occurs in a Python program—an object somehow gets where it shouldn't be? Python reports all errors as exceptions. So you do find out there's a problem, but it's virtually always at run time. "Aha!" you say, "There's your problem: you can't guarantee the correctness of your program because you don't have the necessary compile-time type checking."

This program can even be rewritten in the **Go** language, like this:

```
// staticchecking/petspeak.go
package main
import "fmt"

type Cat struct {}
func (this Cat) speak() { fmt.Printf("meow!\n")}

type Dog struct {}
func (this Dog) speak() { fmt.Printf("woof!\n")}

type Bob struct {}
func (this Bob) bow() {
  fmt.Printf("thank you, thank you!\n")
}
func (this Bob) speak() {
  fmt.Printf("Welcome to the neighborhood!\n")
}
func (this Bob) drive() {
  fmt.Printf("beep, beep!\n")
}

type Speaker interface {
  speak()
}

func command(s Speaker) { s.speak() }

// If "Speaker" is never used
// anywhere else, it can be anonymous:
func command2(s interface { speak() }) { s.speak() }
```

```
func main() {
  command(Cat{})
  command(Dog{})
  command(Bob{})
  command2(Cat{})
  command2(Dog{})
  command2(Bob{})
}

/* Output:
meow!
woof!
Welcome to the neighborhood!
meow!
woof!
Welcome to the neighborhood!
*/
```

Go has no `class` keyword, but you can create the equivalent of basic classes using the above form: what you would ordinarily define as a class, you instead define as a `struct`, within which dwell your data fields (there are none here). For each method, you start with the `func` keyword, then—in order to attach the method to your class—you put parentheses containing the object reference, which can be any identifier but I use `this` here to remind you that it's like the `this` in C++ or Java. Then you define the rest of the function as you do for any other function in Go. (Note there's also no inheritance in Go, so this form of "object-orientedness" is relatively primitive, and probably the main thing that keeps me from spending more time with the language. Composition, however, is straightforward).

The `command()` and `command2()` functions both use structural/duck typing: the exact type of the argument is unimportant as long as it contains a `speak()` method. I show two approaches here: `command()` uses an externally-defined `Speaker` interface, but if that interface is never used anywhere else, you can define it anonymously, inline, as seen in `command2()`.

`main()` demonstrates that `command()` and `command2()` are indeed indifferent to the exact type of their arguments, as long as there's a `speak()`. However, just like C++ template functions, the types are checked at compile time. (The syntax `Cat{}` creates an anonymous `Cat struct`).

Type Checking is Just One Kind of Testing

When I wrote *Thinking in C++*, 1st edition, I incorporated a very crude form of testing: I wrote a program that would automatically extract all the code from the book, using comment markers placed in the code to find the beginning and ending of each listing. It then built makefiles that would compile all the code. This way I could guarantee that all the code in my books compiled and so, I reasoned, I could say "if it's in the book, it's correct." I ignored the nagging voice that said "compiling doesn't mean it executes properly," because it was a big step to automate the code verification in the first place (as anyone who looks at programming books knows, many authors still don't put much effort into verifying code correctness). But naturally, some of the examples didn't run right, and when enough of these were reported over the years I began to realize I could no longer ignore the issue of testing. I came to believe:

> If it's not tested, it's broken.

If a program compiles in a statically typed language, it just means it has passed some tests, which means the syntax is guaranteed to be correct (Python checks some syntax at compile time, as well. It just doesn't have as many syntax constraints). But there's no guarantee of correctness just because the compiler passes your code. If your code seems to run, that's also no guarantee of correctness.

The only guarantee of correctness, regardless of whether your language is strongly or "flexibly" typed, is whether it passes all the tests that define the correctness of your program. And you have to write some of those tests yourself. These, of course, are unit tests. Once you become "test infected," you don't go back.

It's very much like going from old C to C++. Suddenly, the compiler performs many more tests for you. You can work faster. But those syntax tests only go so far. The compiler cannot know how you expect the program to behave, so you must "extend" the compiler by adding unit tests (regardless of the language you're using). If you do this, you can make sweeping changes (refactoring code or modifying design) in a rapid manner because you know that your suite of tests will back you up, and immediately fail if there's a problem—just like a compilation fails when there's a syntax problem.

But without a full set of unit tests (at the very least), you can't guarantee the correctness of a program. To claim that the static type checking constraints in C++

or Java will prevent you from writing broken programs is clearly an illusion (you know this from personal experience). In fact, what we need is

> Strong testing, not strong typing.

So this, I assert, is an aspect of why Python works. C++ tests happen at compile time (with a few minor special cases). Some Java tests happen at compile time (syntax checking), and some happen at run time (array-bounds checking, for example). Most Python tests happen at runtime rather than at compile time, but they do happen, and that's the important thing (not when). And because I can get a Python program up and running in far less time than it takes you to write the equivalent C++ or Java program, I can start running the real tests sooner: unit tests, tests of my hypothesis, tests of alternate approaches, etc. And if a Python program has adequate unit tests, it can be as robust as C++ or Java programs which also have adequate unit tests (although the tests in Python are faster to write).

Robert Martin is a long-time inhabitant of the C++ community. He's written books and articles, consulted, taught, etc. A pretty hard-core, strong-static type checking guy. Or so I would have thought, until I read this weblog entry[216]. Robert came to more or less the same conclusion I have, but he did so by becoming "test infected" first, then realizing that the compiler was just one (incomplete) form of testing, then understanding that a dynamically-typed language could be much more productive but create programs just as robust as those written in statically-typed languages, by providing adequate testing.

Of course, Martin also received the usual "how can you possibly think this?" comments. Which is the very question that led me to begin struggling with the static/dynamic typing concepts in the first place. And certainly both of us began as static type checking advocates. Sometimes it takes an earth-shaking experience—like becoming test-infected or learning a different kind of language—to cause a re-evaluation of beliefs.

How to Argue about Typing

Many people observe that type checking is a religious discussion best avoided. I often agree, having started more than my share of fires in this area. However, there

[216] http://www.artima.com/weblogs/viewpost.jsp?thread=4639

are a few issues surrounding types and type checking that capture the essential distinctions between programming languages. This understanding makes the pitfalls worth the risk, so in this section––from which I drew the closing keynote address at the 2004 Python Conference (PyCon)—I will look at various issues and arguments surrounding the concept of type, and in particular examine the phenomenon of "structural typing" a.k.a. "latent typing" (also unfortunately sometimes called "weak typing"), why the concept is powerful, and how it is expressed in different languages.

Background: When I was in junior high school in Southern California, they tried to teach me Spanish by showing us films about "E Man" (a dog, I think), and albondigas (meatballs) in la biblioteca (the library). That's about all I remember, because I reacted strongly when the teacher would speak Spanish to us, then expect us to reply with something meaningful. I was not conscious enough to see the motivation for learning this, and the effort seemed painful. The same was true for playing the guitar (it makes your fingers hurt) and any number of other potential abilities that never manifested.

So I sympathize with those who resist learning more than one language. I'll probably only ever know a few phrases of non-English languages, and it is only through experience I've discovered the incredible value of learning more than one programming language. The biggest payoff is this: I can now think about things in Java or C++ I was unable to conceive of before I learned Python. And from what I've read and the code I've seen, very few single-language users are able to conceive of these things, either. Your language really does constrain your thoughts.

Some years ago, I began noticing programming language features that seem to revolve around particular philosophies of type checking. These philosophies had as their foundation the idea that a particular way of doing things will always yield better results. In addition, the meanings of various terms tend to be given in the context of the language that one knows and likes best.

I stepped into the middle of this when I began using the term "weak typing" (which I have since observed is quite different from "weakly typed"). At the time, the term seemed to come from an authoritative source, but now I'm unable to track it down and discover its genesis. In Java, I also produced a hailstorm of protest when I asserted (in *Thinking in Java*, 3rd edition) that checked exceptions—where the compiler forces you to write code to handle the exceptions from particular calls, rather than allowing you to decide whether or not to handle the exception at that time—is an experiment in that language that largely causes more trouble than it is worth (not to say checked

exceptions aren't useful sometimes, but they get in the way more than they help). Both of these discussions seem to touch deep nerves in fundamental belief systems.

The value of what I originally called "weak typing," but which is more predominantly called "structural" or "latent" typing is fascinating, whatever we choose to call it. Basically, it's how you say "I don't care what this type is" (in certain programming constructs) "...but I still care that the type behaves correctly...". In C++, this construct is the template. Java uses the term "generic" which implies the same thing, but in practice it can only be as generic as the root class `Object`. But in Python (and SmallTalk), it's just the way you define any function, as `speak()` is defined here:

```
# staticchecking/DogsAndRobots.py

def speak(anything):
    anything.talk()

class Dog:
    def talk(self):  print("Arf!")
    def reproduce(self): pass

class Robot:
    def talk(self): print("Click!")
    def oilChange(self): pass

a = Dog()
b = Robot()

speak(a)
speak(b)

output = """
Arf!
Click!
"""
```

`speak()` doesn't care about the type of its argument. I can pass any object that supports the `talk()` method.

This example easily translates to C++[217]:

[217] You can try it out without the pain of installing a C++ compiler here.

```
// staticchecking/DogsAndRobots.cpp
#include <iostream>
using namespace std;

class Dog {
public:
  void talk() { cout << "Arf!" << endl; }
  void reproduce() {}
};

class Robot {
public:
  void talk() { cout << "Click!" << endl; }
  void oilChange() {}
};

template<class T> void speak(T speaker) {
  speaker.talk();
}

int main() {
  Dog d;
  Robot r;
  speak(d);
  speak(r);
}
/* Output:
Arf!
Click!
*/
```

Here's the same program written in Go:

```
// staticchecking/dogsandrobots.go
package main
import "fmt"

type Dog struct {}
func (this Dog) talk() { fmt.Printf("woof!\n")}
func (this Dog) reproduce() {}

type Robot struct {}
func (this Robot) talk() { fmt.Printf("Click!\n") }
func (this Robot) oilChange() {}

func speak(speaker interface { talk() }) {
  speaker.talk();
}

func main() {
  speak(Dog{})
  speak(Robot{})
}
/* Output:
woof!
Click!
*/
```

It's almost as if you're inventing a new type: "the type acceptable to `speak()`," because it certainly doesn't refer to any other type that exists in the system. You might also argue that `speak()` defines two basic categories of types: those acceptable to `speak()` and everything else. Neither of these definitions work for me. It appears there is either no type specified for `x`, or that the type constraints on `x` are weakened to allow many different types.

After dealing with terminology, this section looks at the way structural/latent typing is achieved in Python and C++, and what generics look like in Java (which doesn't support structural types).

Weakly Typed vs. Weak Typing

C++-Proxy

"C++ has strong, static type checking."

Wonk

"C/C++ is weakly typed because it has casts and unions."

Proxy

"That's a bit extreme, isn't it? How about if we say that C++ is strongly, statically typed with a hole in the type system: casts (and who really uses unions, anyway?)."

Wonk

"Nope. If a language allows an object to accept any incorrect message, it's weakly typed. Java also allows casts, but it checks them, both at compile time to see if the cast is feasible, and at run time to ensure it is correct. So Java is strongly typed, C++ is weakly typed."

Proxy

"So you're saying that if the language is strong everywhere but it has a single trap door, you're going to call the whole thing weak?"

Wonk

"Totally weak, dude."

Clearly, one of the issues revolves around definitions and how closely you stick to them. So first we have to decide "what is a type?" Here are two different ways to define type. An object can be:

1. Syntactically compatible: The object provides all the expected operations (type names, function signatures, interfaces).

   Or:

2. Semantically compatible: The object's operations all behave in the expected way (state semantics, logical axioms, proofs).

The first approach is often considered *type* (it provides a particular interface) and the second is usually called *class* (describes implementation constraints). I think this fits closely with the issue at hand: we are predominantly concerned with the operations that can be performed upon objects, and the only semantics of interest are whether those operations are safe and if the language system consistently reports unsafe operations (e.g. C++ allows old-style unchecked casts and doesn't report anything if you use them).

- Does the language apply the type constraints to the symbols, or to the objects?

You apply the type constraints to the symbols especially if you have limited run-time support (C++). It's also possible to do both, as with Java, which constrains the types and usage of all symbols at compile time, but also has enough runtime support to perform limited dynamic checks. If it can't check something statically (array bounds or null pointers, for example), it checks them dynamically.

- Are the constraints applied before the program runs(static type checking) or while the program is running (dynamic type checking)?

Again, Java does some of both. But dynamically-checked languages seem to have both more flexibility and at the same time better ability to make sure an object is properly treated. They can afford to treat the symbols casually because the objects guard themselves against improper use at runtime, rather than relying on the compiler to guard them.

- Does the language "guess" the type based on usage (type inference), or does it know via a declaration somewhere (manifest typing)?

Type inference is sometimes confused with dynamic typing, because the symbols can appear to have no type (or no fixed type). Scala is a good example of a type-inference language, and it is much nicer to use when defining types.

- If a language is statically typed, does it ever relax the type constraints a bit to allow more flexible programming? (This is sometimes called *gradual typing*).

This is my key area of interest in this section. Note that in the dog-robot example above, many people reach the conclusion that because the `anything` in the `speak()` argument list is not constrained, there is no type safety. However, strong type safety is maintained—you still cannot send an improper message to an object.

Lexicon

- **Static typing**: Types are checked at compile time.
- **Dynamic typing**: Types are checked at run time.
- **Strong typing**: You can't successfully apply an improper operation (send a bad message) to an object.
- "**Weakly typed**": You can successfully perform an improper operation on an object. There may or may not be common confusion with the following term, but people use it anyway.

- **Latent typing/Structural typing**: Type constraints are relaxed in a few specific cases to make programming more flexible and powerful.
- **Type inference**: You don't have to say what the types are, the language system figures out types based on how they are used (Scala does this).
- **Manifest typing**: You must specify the types when you create them.
- **Duck Typing**: A term apparently adopted by Ruby, to describe structural/latent typing. "If it walks like a duck and talks like a duck, it's a duck." The value of duck typing seems to be the ability to create an adapted object without creating a new adapter class (see *Adapter* in the Patterns chapter).

The phrase "Implicit typing" would seem to neatly describe what I'm talking about here: an implied type is created when you define a function, template, or generic. Unfortunately, Fortran has already co-opted this term, with its "first letter of the identifier implies Real or Integer" scheme. I'm tempted to vary it slightly to "implied typing" because it is so close to the mark, but I'm sure we'd end up with the same issue as "weakly" vs. "weak." (I've also noticed people using the term "implicitly typed").

> `la·tent`
>
> 1. *Present or potential but not evident or active. In existence but not manifest. Latent talent.*
> 2. *Psychology. Present and accessible in the unconscious mind but not consciously expressed.*
>
> *A fingerprint that is not apparent to the eye but can be made sufficiently visible, as by dusting or fuming, for use in identification.*

It's there, but not explicit. That's a pretty good word after all. Or:

> If you only need something that quacks, don't worry about checking that it's also a duck.—Andrew Dalke

Of course with latent typing we are not even interested in naming it a duck in the first place. I believe that the duck is there even if I see only scant evidence for its existence.

Templates & Latent Typing vs. Java Generics

Before templates were introduced to C++, you created generic containers using macros. For the library designer this was painful, for the client programmer somewhat less so. When templates were introduced, I explained that if C++ had a singly-rooted hierarchy (everything ultimately inherits from `Object`), these would not be as necessary because the containers could be written to hold `Object`, and therefore automatically hold everything. This is the way SmallTalk had done it and that seemed pretty successful.

Of course SmallTalk used latent typing, and so did not require the casts. When Java followed this prescription, it had static typing and so you had the benefit of containers built with a singly-rooted hierarchy. But with Java you had to cast everything up and down all the time.

The solution introduced in Java 5 is the so-called *generic*, which implies "anything" but which is actually constrained. This solution makes collections more civilized:

```
// staticchecking/drc/DogAndRobotCollections.java
// {java staticchecking.drc.DogAndRobotCollections}
package staticchecking.drc;
import java.util.*;

class Dog {
  public void talk() {
    System.out.println("Woof!");
  }
  public void reproduce() { }
}

class Robot {
  public void talk() {
    System.out.println("Click!");
  }
  public void oilChange() { }
}

public class DogAndRobotCollections {
  public static void main(String[] args) {
    List<Dog> dogList = new ArrayList<>();
    List<Robot> robotList = new ArrayList<>();
```

```
    for(int i = 0; i < 10; i++)
      dogList.add(new Dog());
    //- dogList.add(new Robot()); // Compile-time error
    for(int i = 0; i < 10; i++)
      robotList.add(new Robot());
    //- robotList.add(new Dog()); // Compile-time error
    dogList.forEach(Dog::talk);
    robotList.forEach(Robot::talk);
  }
}
/* Output:
Woof!
Woof!
Woof!
Woof!
Woof!
Woof!
Woof!
Woof!
Woof!
Woof!
Click!
Click!
Click!
Click!
Click!
Click!
Click!
Click!
Click!
Click!
*/
```

This is definitely an improvement for collections and a few other things. However, you cannot say "I don't care what type this is" for an argument, like you can in C++. So you cannot say in Java:

```
class Communicate {
  public static <T> void speak(T speaker) {
    speaker.talk();
  }
}
```

You can only say, "This thing can be no more specific than `Object`." So this compiles, because I call an `Object` method for it:

```
public class NothingButObject {
  public <T> String f(T anyObject) {
    return anyObject.toString();
  }
}
```

This turns out to work just fine when you're dealing with Java collections, which are already constrained to hold nothing more specific than `Object` (so they hold any type).

But to say something more general than that, to write true "generic code," you cannot (Java 8, however, allows "Assisted Latent Typing," described at the end of the Generics chapter). To apply the `communicate()` function to dogs and robots, you must do this:

```
// staticchecking/dr/DogsAndRobots.java
// {java staticchecking.dr.DogsAndRobots}
package staticchecking.dr;

interface Speaks { void talk(); }

class Dog implements Speaks {
  @Override public void talk() {
    System.out.println("Woof!");
  }
  public void reproduce() { }
}

class Robot implements Speaks {
  @Override public void talk() {
    System.out.println("Click!");
  }
```

```
  public void oilChange() { }
}

class Communicate {
  public static <T extends Speaks>
  void speak(T speaker) {
    speaker.talk();
  }
}

public class DogsAndRobots {
  public static void main(String[] args) {
    Dog d = new Dog();
    Robot r = new Robot();
    Communicate.speak(d);
    Communicate.speak(r);
  }
}
/* Output:
Woof!
Click!
*/
```

You must constrain the generic type accepted by the function to conform to a class or interface which contains the operations you are calling inside that function.

But of course this is no better, and actually more confusing, than simply using the interface alone as the argument:

```
class Communicate {
  public static void speak(Speaks speaker) {
    speaker.talk();
  }
}
```

What I was really hoping for in Java generics was true latent typing:

```
<T> void f(T objectOfAnyType) {
  objectOfAnyType.anyOperation();
}
```

What I got was cleanup of templates and a few other features, but none of the power of true generic (latent) types (until, as mentioned earlier, Java 8). The primary

argument against latent types in Java is that "it isn't type safe," which I hope to show is incorrect later in this appendix.

C++ templates always allowed true generic code. Their downfall was that early compilers produced terrible error messages for template code (they've gotten better since), but many people came to the conclusion that templates were therefore bad.

Another argument against C++ templates is that they cause code bloat because a copy of the template is laid down each time you instantiate it for a new type. However, code bloat of this kind generally results from a misunderstanding of the feature; in particular, putting all your code inside the template instead of inheriting or using composition. The base class or member objects hold the code that doesn't need to be templatized (or reproduced).

It's very powerful to say "it doesn't matter what type this object is, as long as I can perform these operations on it." This is the foundation, for example, of the C++ *Standard Template Library* (STL) algorithms. The algorithms work on containers, which themselves are designed to be as unconstraining as possible so they can hold any kind of object. It's as if there's an "implied singly-rooted hierarchy" in C++ (with a very simple root class).

Flexibility vs. (Perceived) Safety

Java Wonk
: Static type checking is necessary for safety.

Me But Java does some checking at run-time. That's why you need `ClassCastException`.

Wonk
: OK, I'll say this, then: For safety, as much static type checking as possible is always desirable.

Me Isn't it possible to occlude the meaning of the code if you have too much type-checking ceremony around it? And we still haven't made a dent in the "20 working lines of code per day per programmer" statistic.

Wonk
: You're just complaining about finger typing. I have tools like IntelliJ that do a lot of typing for me, so I get the best of both worlds: reduced finger typing and greater type checking.

Me Yes, but code is read much more than it is written. By making the reader do more work you're slowing down the development process.

Wonk

On the contrary, this makes the code more explicit and thus easier to understand.

Me I find Python code much faster to read precisely because it's shorter and more straightforward, and thus it seems much easier to understand than when I'm constantly interrupted by all the ceremony of Java.

Wonk

I don't know Python so I don't believe you. And I'm positive I want the guaranteed safety provided by Java's static type checking.

Me What do you mean, "guaranteed?" With the failure rate of software development projects at least 50%, and possibly as high as 80%?

Wonk

That's not verifiable. And Java has improved the success rate over C++. It's sure made my life easier.

Me True, few are willing to admit their failures so we can't get an exact number except to say it's very high, and very expensive. How can you argue this is a sure thing when the failure rates continue to be so high? It does seem that Java improves the success of projects over C++, but how can you say that anything is working when everything is mostly failing?

Wonk

But you want to reduce the amount of type checking. That makes it worse.

Me No, the amount of type checking is at least the same. It's just when it happens that changes.

Wonk

If you make it happen at run-time instead of at compile time, things can slip through the cracks.

Me Let's look at an example where run-time checking happens in Java. Pre-Java 5 collections allow you to put any object in, then you must cast it back to the desired type when you pull it out. The correctness of the cast is checked at runtime.

Wonk

Yes, very bad. You can put the wrong type into a collection. I could put a `Dog` into a collection of `Cat`. Hey, I think that's an example in your book.

Me Yes, that would be a problem, but you'd get a runtime exception when it happened, so you'd find out about it.

Wonk

But with static typing like we get with Java generics it won't happen at all. That's a big worry solved.

Me This happens to you a lot?

Wonk

Well … no. Actually, I don't remember the last time it happened. But I'm sure the problem is just lurking, waiting to crash an important program.

Me As long as you find out when it happens, you still find out. And you can do that through unit testing.

Wonk

Static checking is better than unit testing because I don't have to write the tests.

Me You eventually have to write some tests. There's always a line where it goes beyond language syntax and becomes the semantics of your particular program. Without your own tests you cannot have a verifiably correct program.

Wonk

Unit tests are good, yes.

Me So what I'm suggesting is that by acknowledging that you have to write your own tests anyway—and thus perform dynamic checking of your application—you can consider moving the line and making things a bit more dynamic. I think the benefit you get in easier-to-write, easier-to-read code will far outweigh the loss of a bit of the static type checking. Keep as much of it as possible, but move a little bit into the runtime arena.

Wonk

Static type checking is too important. It's what Java is about. More static type checking is always better.

Is Latent/Structural Typing Necessary?

In the end we have to ask whether Latent/Structural typing is an essential feature. What does it really do?

It allows you to cut across class hierarchies, calling methods that are not part of a common interface.

Classes only share the methods in their common base class. C++ has no ultimate `Object` base class, so the need for latent typing is more immediately obvious. Java has a singly-rooted hierarchy and `className` can only call `Object` methods in `T`. They are "generic," as long as you don't step out of the bounds of what's already generic (`Object`). This works fine for collections and any other code which only needs the fundamental operations of `Object`.

However, it's still possible to have disjoint hierarchies in Java, with common methods but no common interfaces. The original authors of two separate class hierarchies cannot be expected to predict every commonality that might occur, nor to anticipate you might write a single piece of code that is applied to both of them.

This is more likely to happen in existing hierarchies, ones that are handed to you. To perform common operations across multiple disjoint hierarchies, you are forced to write duplicate code. Thinking this is OK is effectively saying "it's OK to duplicate code," although it may seem like it's just a little bit and not very often.

But what if this is another feature you haven't been using because you didn't know about it? Like objects for a procedural programmer, for example. Or aspects, which can greatly simplified some architectures. C++ programmers benefit by using templates for more than just container classes (although template metaprogramming seems beyond what most people do).

Even if you do argue that all classes can be shoehorned into a hierarchy to achieve a common interface, there is also the argument of the maintenance overhead that results from the extra code. As Java programmers we've been like frogs in saucepans (a myth—apparently they do jump out when the water gets too hot), adjusting to more and more code to do basic things. Over time, Java added syntax to reduce some of that code, and this is an admirable development. It still requires more finger-typing than may be necessary. The counter-argument is that we have tools like IntelliJ to do the typing for us, but the overhead of reading and maintaining the code—which is where most of the money ends up going—still increases.

Consider the opposite view. At the Software Development Conference 2004, James Hobart pointed out the distinction between "probability" and "possibility." Here, it is arguably more probable that C++ will have disjoint class hierarchies than will Java. Java generics seem to work adequately for collections. It's possible you might encounter disjoint class hierarchies that you'd like a single piece of code to cut across. But perhaps the probability is not that high it will happen very often in practice. This

Hobart called "going down the happy path." In Java, we have no choice but to take this path, but in other languages we may see through experience what is lost by doing so.

We're Really Talking about Testing

In the end, these are all just different kinds of tests. We can consider them on a spectrum:

- Testing by the compiler (static type checking)
- Testing by the run-time system (dynamic type checking + more)
- Unit testing of your classes
- Conformance testing of your program

The compiler and run-time system can only know so much. There can only be a certain set of tests general enough they can be applied to every program. At some point you must begin writing your own tests to test the semantics of your particular program, to ensure its correctness.

What we are talking about, then, is not whether tests are critical (they are), but when and how these tests take place. More importantly, about balancing the expressiveness and clarity of a language with the place in the spectrum where these tests occur. My opinion: programs are prose, and must be readable. Static checking is good if it doesn't impact the clarity, expressiveness, and productivity of the language. If those are impacted, the checking should be moved to reduce that impact.

This opinion is based on my belief and experience that you cannot rely on static checking to verify program correctness, and therefore it is unreasonable to obscure the meaning of your program with high-ceremony syntax under the illusion it is somehow safer. For full safety, you'll always need run-time tests. Given this, to me it makes no sense to make your life harder by obscuring the code because of this illusion.

Determinism

Albert Einstein was very disturbed by the Heisenberg uncertainty principle (you can't know both position and momentum of small particles). He believed that "God

does not play dice with the universe" and postulated a "hidden variable" theory that, when discovered, would allow us to completely determine everything.

Bell's theorem showed there is no hidden variable theory.

Steven Hawking describes how you can have higher determinism nearer to an event (in space-time) than further away. The effect of the uncertainty principle is cumulative in space-time.

The Greeks believed there was fate and free will together, which people have always found confusing because we demand one or the other exclusively. Perhaps Hawking's approach explains the dichotomy.

It may be possible to create programs that are provably correct, but my guess is that these programs will fill only a small fraction of the general solution space.

Static type checking is only one kind of testing, and very limited in its effect—as you move out in space time from the point of compilation, the "software uncertainty principle" adds more and more noise to the results. Only by testing throughout will you get a program that is as correct as you are able to describe it.

The Cost of Productivity

Some very interesting analyses have come out of my discussions of latent typing, especially those by an anonymous blogger named "Pixel." Although there are places where Pixel misunderstood what I was saying (or more likely I didn't say it clearly enough), in general he or she states the "Java case" reasonably well.

That case includes the regular argument that people are just as productive in Java as they could be in Python, an argument which is almost universally made in the abstract, without any direct experience in both languages. I have regular experience in both languages, and the result is always: To get a lot done, use Python. The testing doesn't come, as Pixel argues, with special hand-testing, but by using the actual data that you're trying to manipulate. You just get there faster, and start finding the real problems, using real data, faster. I have this experience over and over.

And the kinds of problems I solve are only theoretically solvable using Java. It would take at least 5-10 times longer to do it, and that assumes you wouldn't give up or get lost first. For example, you can translate something from Python, but it's much

easier with the Python design as a road map, and the resulting code would still be much bigger, messier, and (a big point of argument because people regularly claim that the more verbose Java is easier to maintain because it's so much more explicit) harder to maintain.

In the follow up, Pixel ends by asking "How much value do you put on checking as much as possible at compile time?" This oversimplifies the issue. If the type checking came at no cost, the answer is as easy and obvious as Pixel implies with this question. I've generally found that most folks who argue that static type checking must be preserved and increased at all costs have no experience of a down side. And if you don't see any down side, you think arguing against it is obviously crazy.

I know because I began firmly in the static type checking camp, having seen the benefits of moving from pre-ANSI C to C++. C++ static type checking found lots of errors that (especially pre-ANSI) C didn't, so it was a clear win. And the extension of this philosophy to Java was thus obvious. But that was my sole dimension of experience. It was only when I began using a more dynamic, more succinct language (Python) that I got a new dimension.

The experience most people have goes something like this: they have had enough exposure to Python (or substitute your dynamic language of choice) to keep it in the back of their mind. A problem arises that might require a one shot solution. Knowing how much effort Java is, they think this is a good place to try Python because it might save some effort. Then they have the watershed experience of solving the problem in far less time than it would take in Java. The next time a similar problem arises, they reach for Python more quickly, until they start using Python as a preferred tool, only using Java when there's no other choice.

During this process, the experience of being dramatically more productive in Python repeats itself over and over. It's not a philosophical argument (because the answer is "obviously" that the statically typed language is better), it's direct experience. The correctness and quality of the resulting programs is consistently very high.

Then you reach the intellectual vs. experiential crisis: you "know" that Java is "better," but your experience is that Python is "better." How can this be, when all the arguments you "know" to be true clearly show that a static typing language must be "better." At that point you start trying to resolve this crisis by seriously questioning your preconceptions. But the only arguments you hear back are the ones you yourself (I myself) previously "knew" were true, and have been shown from experience to be

uncertain.

In my experience there's a balance between the value of static typing and the resulting impact it makes on your productivity. The argument that "static is obviously better" is generally made by folks who haven't had the experience of more productivity in an alternative language. When you have this experience, you see that the overhead of static typing isn't always beneficial, because sometimes it slows you down enough it ends up having a big impact on productivity.

I can't quantify this. I haven't been able to come up with a from-first-principles mathematical proof, probably because it depends on human factors, like how much time it takes to remember how to open a file and put the try block in the right places and remember how to read lines, then remember what you were really trying to accomplish by reading that file (although this experience is greatly improved in Java 8). In Python, I can process each line in a file by saying:

```
for line in open("FileName.txt"):
   # Process line
```

(Python 3 adds a context manager to automatically close the file, not shown here). I didn't have to look that up, or to even think about it, because it's so natural. I always had to look up how to open files and read lines in Java. You can argue that Java wasn't intended to do text processing and I'd agree with you, but unfortunately it seems like Java is mostly used on servers where a very common task is to process text.

There are studies about how much time it takes to recover from interruptions, but I can't make any direct connection other than to say it seems to me this must have a big influence, both when writing the code (yes, I know that much of it is automated with IntelliJ and similar editors) and reading the code.

Java is what it is. Demonstrating that Java generics don't support latent typing was primarily meant to get people to show me how Java generics really do support latent typing in the event I had misunderstood something. They didn't, and I didn't, so that's the case.

In the end, I've realized that the requirement that you must use interfaces everywhere really isn't that big of a deal. In Python, which is a succinct language, it would really stick out, but Java is unashamedly verbose, and many people seem to like the verbosity and feel it is a benefit. Requiring a few extra interfaces here and there

is not much of an impact on the resulting Java code, and is indeed in keeping with the Java language philosophy (it would probably surprise people if latent typing did have direct support in the language).

So I understand that Java generics are really just autocasting for nicer use of containers, and that's good and useful but you can only take Java generics so far.

It is possible to do latent typing in Java, but you must really want to do it because it requires more effort. Let's revisit the dogs and robots program one more time, this time using reflection to produce a method that implements latent typing:

```
// staticchecking/latent/Latent.java
// {java staticchecking.latent.Latent}
package staticchecking.latent;
import java.lang.reflect.*;

class Dog {
  public void talk() {
    System.out.println("Woof!");
  }
  public void reproduce() {}
}

class Robot {
  public void talk() {
    System.out.println("Click!");
  }
  public void oilChange() {}
}

class Mime {
  public void walkAgainstTheWind() {}
  @Override
  public String toString() { return "Mime"; }
}

class Communicate {
  public static void speak(Object speaker) {
    try {
     Class<? extends Object> spkr =
       speaker.getClass();
     Method talk =
```

```
        spkr.getMethod("talk", (Class[])null);
      talk.invoke(speaker, new Object[]{});
    } catch(NoSuchMethodException e) {
      System.out.println(
        speaker + " cannot talk");
    } catch(IllegalAccessException e) {
      System.out.println(
        speaker + " IllegalAccessException");
    } catch(InvocationTargetException e) {
      System.out.println(
        speaker + " InvocationTargetException");
    }
  }
}

public class Latent {
  public static void main(String[] args) {
    Communicate.speak(new Dog());
    Communicate.speak(new Robot());
    Communicate.speak(new Mime());
  }
}
/* Output:
Woof!
Click!
Mime cannot talk
*/
```

I can thus call `speak()` on anything, and it will only invoke `talk()` on objects that actually have a `talk()` method.

Static vs. Dynamic

I received this letter from a reader. I have edited it a bit and also "corrected" the terminology, as described after the letter.

I'm a professor at Trinity University teaching a course in programming languages. Static typing is something that we have talked about a fair bit and one of my students came across your web page which we used quite nicely to fuel a discussion in one class.

Personally I lean toward static typing and I was wondering if you can comment on two things for me.

The first point is my general reason for liking static typing. Basically, static typing provides a proof that some aspect of the program works. Granted, the type system doesn't prove everything is OK and testing is still needed, but at least some aspect is proven correct. Why this seems to matter to me is that complete testing requires the number of tests grow at least exponentially with code size. Anything less will leave many paths through the code untested. No matter how productive one can be in a certain language, writing an exponential number of test cases will overwhelm that. Given this, isn't it worth the reliability to have some aspects proven correct by the compiler?

The more interesting question I have is what you think about ML and its derivatives in regard to the strong testing vs. static typing argument. These languages have the advantage of dynamically-typed languages in that they rarely require the user to specify types, however, they infer the types for all expressions and are statically typed. In this regard, their typing system is far safer than a language where dynamic checks are required (Java) or where users can coerce the type system into errors (C/C++).

The comparison of ML to Scheme came immediately to my mind when I read your article on strong testing vs. strong typing because Scheme does all of its type checking dynamically and gives the user complete flexibility when writing code. However, in my experience, it is very hard to write large programs in and the fact that Scheme accepts a chunk of code doesn't mean it is anywhere close to being correct. ML is the opposite in that is does static type checking on everything even without the user specifying the type of anything. When I get my ML code to compile, it works the majority of the time and one or two simple tests will uncover any bugs that remain.

With Microsoft releasing F# into the .NET framework, it is likely that many more programmers will be exposed to the ML style of coding. I just have to wonder if you would feel that having strong static typing without the coder specifying the types leads to the best of both worlds.

First, I'm at least partially responsible for the mix-up in terminology that has propagated around this, and I have not had the time nor inclination to go back into previous articles and fix this up. It also seems that things are not as clearly defined in the computer world around these terms. So to try to help rectify the situation:

`Weak typing`: Allows incorrect messages to be sent to objects. C and C++ allow you to successfully cast to the wrong type, and are thus viewed as having weak typing (although Stroustrup once said that "C++ was strongly typed with a couple of holes in the type mechanism"). I have argued in the past that "weakly typed" (a term I originally used to mean "latent typing") was distinct from "weak typing," but I think the terminology differences are far too subtle and not worth arguing over. In addition, the idea of latent typing confuses the issue; C++ (static typing, arguably weak typing in isolated spots), Python and Ruby (dynamically typed) all support structural/latent typing (Ruby calls it "duck typing").

`Dynamic typing`: Type checking still happens, and it can be strong typing, but it happens at runtime rather than at compile-time. A strongly-typed dynamic language still only allows you to send valid messages to objects.

To clarify an important point: I'm not against static type checking. The problems as I see them are that:

1. There is an illusion that static type checking can solve all of your problems, followed by the conclusion that more static type checking is always better.
2. Additional forms of static type checking are often added to a language without regard to the actual cost. In extreme cases you spend all your time arguing with the compiler.

In general, my attitude is that static typing is desirable as long as it doesn't cost you too much. As you point out, type inference as found in ML and Scala gives you static type checking without requiring that you give extra information that the compiler can figure out itself.

I do seek the "best of all worlds," as you suggest. The question is whether a particular implementation of static type checking is helping more than it is impeding. I am often accused of just complaining about "finger typing," but what I observe is much more than just the extra carpal assaults (much of which, it has been pointed out, can be automatically generated by tools such as IntelliJ).

The real issue is the limits of the human mind to manage complexity. The hard boundary is the famous number "seven plus or minus two," the number of things that we can hold in our mind at any one moment. I assert that all progress in computer programming comes by improving the mental model to make it simpler for us to manipulate the essential concepts. The reason I find Python so fascinating

is it seems to regularly grasp and incorporate new perspectives on "simpler" and "essential concepts." Much of my fascination is in how this seems to happen, and I currently have little or no sense of the secret behind it. The language design seems to incorporate the psychology of computer programming.

Despite this, I've had some leanings back in the direction of static type checking. As you point out, the goal is to create solid components—the question is how to accomplish that? In a dynamic language you have the flexibility to do rapid experimentation which is highly productive, but to ensure that your code is airtight you must be both proficient and diligent at unit testing. In a language that leans towards static type checking, the compiler will ensure that certain things will not slip through the cracks, and this is helpful, although the resulting language will typically make you work harder for a desired result, and the reader must also work harder to understand what you've done. I think the impact of this is much greater than we imagine.

In addition, I think statically typed languages only give the illusion of program correctness. In fact, they can only go so far in determining the correctness of a program, by checking the syntax. But I think such languages encourage people to think everything is OK, when in fact the requirement for developer testing is just as important. I also suspect that the extra effort required to run the gauntlet of the compiler saps some of the energy required for developer testing. And you bring up an interesting question in suggesting that a dynamically-typed language may require more unit testing than a statically typed language. Of this I am not convinced; I suspect the amount may be roughly the same and if I am correct it implies that the extra effort required to jump through the static type-checking hoops may be less fruitful than we might believe.

What is the best of both worlds? In my own experience, it's very helpful to create models in a dynamic language, because there is a very low barrier to redesigning as you learn. You're able to quickly try out your ideas to see how they work with actual data, to get some real feedback about the veracity of the model, and change the model rapidly to conform to your new understanding.

I think this approach has great benefits over simply modeling with UML. Those produce a model that is a fantasy and only after some complex transformations do you have something that expresses and tests your ideas. My experience with complex transformations of this kind is that they weigh you down and discourage you from

experimenting and making changes. With a dynamic language, on the other hand, the model becomes the code and vice-versa, and so you're able to experiment without inertia. I think this lightness is very important, because it is far closer to the way our brains work.

Once you've worked out and tested a model using a dynamic language, is it then worth transforming it into a statically-typed language? My experience with Python has not compelled me to do this for two reasons:

1. Once I get something working in Python, it seems to work pretty well. The benefits of transforming the model into a statically-typed language at that point are few. Others' experience with significant Python programs seems to support this—large Python programs seem surprisingly bug-free.
2. I usually find that any program that is used regularly will be changed. It is much easier to change the program in Python than it is in a statically typed language, so I have further incentive to leave it in Python.

I certainly don't mean to imply it never makes sense to transform a debugged Python model into Java, just that I haven't been compelled to. I could easily see many business situations where:

1. You begin by "sketching out" a preliminary model, using "UML lite" and/or CRC cards.
2. You implement this model in Python or Ruby, whichever language the developers are comfortable with. At this point you leave the paper model behind, and the code becomes your model. Python is often described as "executable pseudocode" and this becomes very helpful during modeling and experimentation.
3. You experiment and evolve this "live" model until it seems like you've worked out the kinks and it will do the trick.
4. You then translate into Java or C++ or some other language that the project constraints dictate.

By developing the model in a language that encourages change, my experience is that you end up with a better model, and this produces a distinct benefit when that model is translated to your implementation language.

In the end, my primary interest is in productivity and scalability. Visual Basic is a very productive language for small projects (and there are lots of those, so it solves

lots of problems) but it doesn't scale well. Perl also falls into this category, although its apparent successor, Ruby, seems to implement the object paradigm reasonably well and seems to scale to larger projects. Python has been used on numerous large projects and, despite its lack of static type checking, the results appear to have very low bug counts.

This last point is a major puzzle—we believe that static type checking prevents bugs, and yet a dynamically-typed language produces very good results anyway. As I have tried to delve more deeply into this mystery, many of my preconceptions—the major one being that static type checking is essential—have been challenged. An initial response to this is often to simply deny it's the case, but once you begin denying evidence your theories rapidly become nothing more than fantasies. In my own experience it can be very hard to put my finger on exactly why Python works so well. However, in trying to do so I have discovered many things and gained greater understanding about other languages.

My guess is that Python allows me to think more clearly about concepts of the problem I'm trying to solve. It is less distracting because it doesn't force me to think so much about rules imposed by the language—rules that are basically arbitrary when I'm trying to produce an effective model of my problem space. By getting out of the way, Python and similar dynamic languages allow me to spend more of my brain's "seven plus or minus two" items on the problem itself, and less on the details of the language implementation. I have had this experience more than once, for example when going from C++ to Java where I no longer had to worry about `operator=` and the copy-constructor. Having Java take care of more things for you definitely seems to improve programmer productivity, and I've had the same experience with Python.

In the end, I can still see value in taking a model that was evolved using Python or Ruby and reimplementing it in Java. Much of this value is in fitting into an existing Java development environment, but it also seems likely you might discover some bugs because of static type checking. It would be interesting to hear experiences and bug counts from people who have done this experiment.

You asked a question about whether type inference a la ML might give us the best of both worlds. I like type inference because it reduces the overhead on the programmer, but it is also a static typing mechanism (it doesn't work unless there's enough context to infer the static type). Also, it only touches on the different benefits offered by a dynamic language.

We need help to create accurate programs, and it's pretty hard to argue against static typing. The type system is the water where our object-oriented fish swim, and allowing holes and back doors seems to say "these things are true about a type, but you can only rely on it if people know what they are doing and are behaving well," which is not very reassuring. Static typing actually helps us think about our object models by ensuring their proper behavior.

The question is not whether type checking is a good thing. I'd say it unequivocally is. The question is, when does the type checking occur, and how much does static checking cost vs. dynamic checking. In C++, effectively all the type checking is static. In Java, it is both static and dynamic (I think any language that tries to achieve thorough type checking will require some dynamic type checking), and in Python and other dynamic languages it is predominantly dynamic. As you note, in most dynamic languages not all execution paths are tested by the compiler, and this can be somewhat of a problem, but how bad is it? I suspect that if we applied the same unit tests to both a Java program and its equivalent Python version we would exercise all the execution paths. I think that the unit tests are at a high enough level in both cases that you end up with the same number. Put another way, I don't think you need extra unit tests to produce syntax checking, but that the syntax checking will fall out naturally when the unit tests exercise the class interfaces. I don't have direct evidence for this other than not having to do this extra work myself.

One of the things that Java generics accomplishes is the static type checking of collection classes. This prevents `ClassCastExceptions` at runtime, which is somewhat useful, but if that were the only reason for Java generics it wouldn't be worth the complexity of the syntax (fortunately, with some effort you can create generic code, as I showed earlier). The resulting `ClassCastExceptions` don't happen that often, and are not difficult to find when they do. This is a case where dynamic type checking is adequate.

I have also pointed out that checked exceptions do not scale well, and easily get in the way even for small programs. The fact that no language designed since Java has duplicated this experiment makes, I think, a strong argument against it. However, I have found that the Java 1.4 `RuntimeException(Throwable cause)` constructor eliminates most of the complaints I have about checked exceptions—if they get in the way, I can turn them off with only a minor amount of coding. Thus, I can choose to leave them on or turn them off.

These are a couple of examples where too much static type checking, no matter how well-intentioned, gets in the way of both the creation and the examination of code. Despite that, I actually like static type checking, appropriately used. Python 3 even adds an optional type annotation mechanism, enforced by external static checking tools. But I resist any implications that "all static checking is good, so more is always better." If you think about this you can easily imagine it taken to extremes, and in a number of cases Java has done exactly that, without doing a cost-benefit analysis of the results. It is the illusion that there is no cost to static type checking which I argue against.

Appendix: The Positive Legacy of C++ and Java

> In various discussions there are assertions that C++ was a poorly-designed language. I think it's helpful to understand both C++ and Java language choices to see the bigger perspective.

That said, I hardly use C++ anymore. When I do, it's either examining legacy code, or to write performance-critical sections, typically as small as possible to be called from other programs written in other languages.

Because I was on the C++ Standards Committee for its first 8 years, I saw these decisions being made. They were all extremely carefully considered, far more so than many of the decisions made in Java.

However, as people have rightly pointed out, the resulting language was complicated and painful to use and full of weird rules I forget as soon as I'm away from it for a little while—and I figured out those rules from first principles while I wrote books, not just by memorizing them.

To understand how the language can be both unpleasant and complicated, and well designed at the same time, you must keep in mind the primary design decision upon which everything in C++ hung: compatibility with C. Bjarne Stroustrup (the language's original creator) decided—and correctly so, it would appear—that the way to get the masses of C programmers to move to objects was to make the move transparent: to allow them to compile their C code unchanged under C++. This was a huge constraint, and has always been C++'s greatest strength … and its bane. It's what made C++ as successful as it was, and as complex as it is.

It also fooled the Java designers who didn't understand C++ well enough. For example, they thought operator overloading was too hard for programmers to use properly. Which is basically true in C++, because C++ has both stack allocation and heap allocation and you must overload your operators to handle all situations and not

cause memory leaks. Difficult indeed. Java, however, has a single storage allocation mechanism and a garbage collector, which makes operator overloading trivial—as was shown in C# (but had already been shown in Python, which predated Java). But for many years, the party line from the Java team was "Operator overloading is too complicated." This and many other decisions where someone clearly didn't do their homework is why I have a reputation for disdaining many of the choices made by Gosling and the Java team. (Java 7 and 8 have included far better decisions, for some reason. But the backwards-compatibility constraint will always prevent the really great improvements. The language can never be what it might have been.)

There are plenty of other examples. Primitives "had to be included for efficiency." The right answer is to stay true to "everything is an object" and provide a trap door to do lower-level activities when efficiency was required (this would also have allowed for the hotspot technologies to transparently make things more efficient, as they eventually did). Oh, and the fact you can't use the floating point processor directly to calculate transcendental functions (it's done in software instead). I've written about issues like this as much as I can stand, and the answer I hear has always been some tautological reply to the effect that "this is the Java way."

When I wrote about how badly generics were designed, I got the same response, along with "we must be backward compatible with previous decisions made in Java" (Even though they were bad decisions). Lately more and more people have gained enough experience with generics to see they really are very hard to use—indeed, C++ templates are much more powerful and consistent (and much easier to use now that compiler error messages are tolerable). People have even been taking reification seriously—something that would be helpful but won't put that much of a dent in a design that is crippled by rigid constraints.

The list goes on to the point where it's just tedious. Does this mean Java was a failure? Absolutely not. Java brought the mainstream of programmers into the world of garbage collection, virtual machines and a consistent error handling model. With all its flaws, it moved us up a level, to the point where we are now ready for higher-level languages.

At one point, C++ was the leading language and people thought it would always be so. Many think the same about Java, but Java has made it even easier to replace itself, because of the JVM. It's now possible for someone to create a new language and have it run as efficiently as Java in short order. Previously, getting a correct and efficient

compiler took most of the development time for a new language.

And we are seeing this happen—both with higher-level static languages like Scala, and with dynamic languages, both new and ports, like Groovy, Clojure, JRuby and Jython. This is the future, and the transition is much smoother because you can easily use these new languages in conjunction with existing Java code, and you can rewrite bottlenecks in Java if necessary.

At this writing Java is the number one programming language in the world. However, Java will eventually diminish, just as C++ did, relegated to special cases (or perhaps just to support legacy code because it doesn't have the same connection to hardware as C++ does). But the unintentional benefit, the true accidental brilliance of Java is that it has created a very smooth path for its own replacements, even if Java itself has reached the point where it can no longer evolve. All future languages should learn from this: either create a culture where you can be refactored (as Python and Ruby have done) or allow competitive species to thrive.

Appendix: Becoming a Programmer

A mashup of blog posts I wrote in 2003, 2006, 2007 and 2009.

How I Got Started in Programming

This was a rather long and circuitous path. In freshman algebra in high school (1971), the extraordinarily weird teacher had a thing for computers and managed to get an ASR-33 teletype with a 300-baud acoustic phone coupler (which I learned to whistle at and get a response) along with accounts on the HP-1000 computer that the high school district used. We were able to create and run BASIC programs and save them on punch tape. I was fascinated and went home to write programs in the evenings that I would bring back and key in whenever I could. I invented HOSRAC.BAS which was a horse-racing simulation, using asterisks to represent the horses as they moved (this was on paper printout, so it took a little imagination).

My friend Daniel (the same one who designs my book covers now) had a brother who was making money for a time by providing pinball machines to bars and restaurants. He had a slot machine where I got all of that out of my system, can't stand the things now, and also one of the very first Pong games (and I now almost never play computer games. Perhaps I'm a humorless sort, but it seems like programming is far more intriguing and challenging than playing a computer game).

I got involved in photography and journalism later in high school, and majored in journalism during my first year of college. I decided I had learned enough about that from school, and changed to physics. Several colleges later I completed a physics degree at UC Irvine and added enough engineering classes to make it a double major if I had chosen a particular field of engineering, but I was trying to be broad, so my undergraduate degree is in "applied physics." As an undergraduate I took a smattering of computer programming classes here and there, enough for entertainment but

nothing that provided any depth. I think those classes helped add to my foundation by regularly trickling in little bits of information, but I had none of the depth or perspective necessary to really understand anything. I didn't make any distinction between the computer and the compiler or interpreter (and only had a vague sense of compiler vs. interpreter); it was all the infallible computer to me and the idea that a bug could exist in the language or operating system was so purely theoretical I never considered the possibility.

I went to Cal Poly San Luis Obispo for graduate school because (A) I really liked the area (B) they accepted me and even gave me a teaching job and a fellowship, but most important (C) I couldn't imagine working at a job longer than one summer. I was definitely not ready to join the working world.

As a physics student I studied solar power systems, which were big at the time (California gave a tax credit if you put a solar system on your house or business, and many business sprang up because of that), and Cal Poly seemed to promise these things in their engineering department. However, to get a degree in Solar Engineering would have taken many years because the necessary classes were not offered often enough. As a result, I took whatever graduate engineering classes were offered, and got a broad education (as was my wont) including mechanical, solar, electrical and electronic engineering. The class I taught was an introduction to electrical engineering for non-Electrical Engineering majors. The graduate engineering classes that were offered most often were in computer engineering, so I ended up with a degree in that. I was also taking art courses, a couple of dance classes, and a few computer science courses (Pascal and data structures), but in computer engineering I finally struggled through the steps of how a processor worked, and I carried that with me ever since. That was really the foundation of my knowledge of computers.

When I did start working, it was as a computer engineer, with a fair amount of hardware and relatively simple and low-level programming. I began to teach myself C because it seemed the ideal embedded systems language, and slowly started to understand more about programming languages. At one company we actually built our compiler from sources, which was a bit of an eye-opener for me (A compiler is just another piece of software! Imagine that).

When I went to work at the University of Washington School of Oceanography (for Tom Keffer, who later founded Rogue Wave), we decided to use C++. I had only one book to learn from (Stroustrup's; not a beginner text) and ended up having to

struggle through and understand language features by examining the intermediate C code that was generated by the C++ preprocessor. Painful, but intensely educational. I have used that experience ever since, because it gave me the ability to dissect a language and see it for what it is. I suppose that is also where I began to learn critical thinking.

So the concepts didn't come all at once. Instead they trickled in over time, and everything I know has taken time to assimilate. If I seem to get a new concept easily now, it's only because it's a variation on the accumulation of concepts I already know. In the Cal Poly computer science graduate program (which would take students who had non-CSci undergrad degrees), the students used to say that it took a year of being confused about computers before anything at all started to make sense (and they were in an immersion program). People often have unrealistic expectations of themselves when they try to learn about computers—usually they want a high-paying job within a few weeks, and they've heard that computer programming pays the big bucks. But the best learning process starts with someone being interested in computers, and learning more and more as time passes, and generally teaching themselves. That's primarily what I have done; even though I had a strong foundation via computer engineering, I didn't have many programming courses, but instead am self-taught. And I'm constantly learning new things. In this business, learning new things is a large part of the profession.

A Career in Computing

I regularly receive requests for career advice, and I've tried to capture the answers here.

The question that people ask is usually the wrong one: "should I learn C++ or Java?" In this essay, I shall try to lay out my view of the true issues involved in choosing a career in computing.

Note I am not talking here to the people who already know it is their calling. You're going to do it regardless of what anyone says, because it's in your blood and you can't get away from it. You know the answer already: C++ AND Java AND shell scripting AND Python AND a host of other languages and technologies that you'll learn as a matter of course. You already know several of these languages, even if you're only 14.

The person who asks me this question may be coming from another career. Or perhaps they are coming from a field like web development and they've figured out that HTML is only kind of like programming, and they'd like to try building something more substantial. But I especially hope that, if you are asking this question, you've realized that to be successful in computing, you must teach yourself how to learn, and never stop learning.

The more I do this, the more it seems to me that software is more akin to writing[218] than anything else. And we haven't figured out what makes a good writer, we only know when we like what someone writes. This is not some kind of engineering where all we have to do is put something in one end and turn the crank. It is tempting to think of software as deterministic—that's what we want it to be, and that's the reason we keep coming up with tools to help us produce the behavior we desire. But my experience keeps indicating the opposite, that it is more about people than processes, and the fact it runs on a deterministic machine becomes less and less of an influence, just like the Heisenberg principle doesn't affect things on a human scale.

My father built custom homes, and in my youth I would occasionally work for him, mostly doing grunt labor and sometimes hanging sheet rock. He and his lead carpenter would tell me they gave me these jobs for my own good—so I wouldn't go into the business. It worked.

So I can also use the analogy that building software is like building a house. We don't refer to everyone who works on a house as if they were exactly the same. There are concrete masons, roofers, plumbers, electricians, sheet rockers, plasterers, tile setters, laborers, rough carpenters, finish carpenters, and of course, general contractors. Each of these requires a different set of skills, which requires a different amount of time and effort to acquire. House-building is also subject to boom and bust cycles, like programming. To get in quick, you might take a job as a laborer or a sheet rocker, where you can start getting paid without much of a learning curve. As long as demand is strong, you have steady work, and your pay might even go up if there aren't enough people to do the work. But as soon as there's a downturn, carpenters and even the general contractor can hang the sheet rock themselves.

When the Internet was first booming, all you had to do was spend some time learning HTML and you could get a job and earn some pretty good money. When things turn down, however, it rapidly becomes clear there is a hierarchy of desirable skills,

[218] http://www.artima.com/weblogs/viewpost.jsp?thread=255898

and the HTML programmers (like the laborers and sheet rockers) go first, while the highly-skilled code smiths and carpenters are retained.

What I'm trying to say here is: Don't go into this business unless you are ready to commit to lifelong learning. Sometimes it seems like programming is a well-paying, reliable job—but the only way you can make sure of this is if you are always making yourself more valuable.

Of course you can find exceptions. There are always those people who learn one language and are just competent enough and perhaps savvy enough to stay employed without doing much to expand their abilities. But they are surviving by luck, and they are ultimately vulnerable. To make yourself less vulnerable, you must continuously improve your abilities, by reading, going to user groups, conferences, and seminars. The more depth you have in this field, the more valuable you will be, which means you have more stable job prospects and can command higher salaries.

Another approach is to look at the field in general, and find a place where you already have talents. For example, my brother is interested in software, and dabbles with it, but his business is in installing computers, fixing them and upgrading them. He's always been meticulous, so when he installs or fixes your computer you know it is in excellent shape when he's done; not just the software, but all the way down to the cables, which are bundled neat and out of the way. He's always had more work than he could do, and he never noticed the dot-com bust. And needless to say, his work cannot be offshored.

I stayed in college a long time, and managed to get by in various ways. I even began a Ph.D. program at UCLA, which was mercifully cut short—I say mercifully because I no longer loved being in college, and the reason I stayed in college for so long was because I enjoyed it so much. But what I enjoyed was typically the off-track stuff. Art and dance classes, working on the college newspaper, and the handful of computer programming classes I took (which were off-track because I was a physics undergrad and a computer engineering graduate student). Although I was far from exceptional academically (a delightful irony is that many colleges that would not have accepted me as a student now use my books in their courses), I really enjoyed the life of the college student, and had I finished a Ph.D. I probably would have taken the easy path and ended up a professor.

But as it turns out, some of the greatest value I got from college was from those same off-track courses, the ones that expanded my mind beyond "stuff we already know."

I think this is especially true in computing because you are always programming to support some other goal, and the more you know about that goal the better you'll perform (I've encountered some European graduate programs that require the study of computing in combination with some other specialty, and you build your thesis by solving a domain-specific problem in that other specialty).

I also think knowing more than just programming vastly improves your problem-solving skills (just as knowing more than one programming language vastly improves your programming skills). On multiple occasions I have encountered people, trained only in computer science, who seem more limited in their thinking than those from some other background, like math or physics, which requires more rigorous thinking and is less prone to "it works for me" solutions.

In one session at a conference I organized, one of the topics was to come up with a list of features for the ideal job candidate:

- Learning as a lifestyle. For example, learn more than one language; nothing opens your eyes more to the strengths and limitations of a language than learning another one.
- Know where and how to get new knowledge.
- Study prior art.
- We are tool users.
- Learn to do the simplest thing.
- Understand the business (Read magazines. Start with *Fast Company*, which has very short and interesting articles. Then you'll know whether to read others)
- You are personally responsible for errors. "It works for me" is not an acceptable strategy. Find your own bugs.
- Become a leader: someone who communicates and inspires.
- Who are you serving?
- There is no right answer … and always a better way. Show and discuss your code, without emotional attachment. You are not your code.
- It's an asymptotic journey towards perfection.

Take whatever risks you can—the best risks are the scary ones, but in trying you will feel more alive than you can imagine. It's best if you don't plan for a particular outcome, because you often miss the true possibilities if you're too attached to a

result. My best adventures are ones that start with "lets do a little experiment and see where it takes us."

Some people are disappointed by this answer, and reply "yes, that's all very interesting and useful. But really, what should I learn? C++ or Java?" I'll fend these off by repeating here: I know it seems like all the ones and zeroes should make everything deterministic, so such questions should have a simple answer, but they don't. It's not about making one choice and being done with it. It's about continuous learning and sometimes, bold choices. Trust me, your life will be more exciting this way.

Further Reading

I found all these to be stimulating takes on the subject:

- Teach Yourself Programming In Ten Years[219], by Peter Norvig.
- How To Be A Programmer[220], by Robert Read.
- A speech by Steve Jobs[221] to inspire a group of graduating college students.
- Kathy Sierra: Does College Matter[222]?
- Paul Graham on College[223].
- Joel Spolsky: Advice for Computer Science College Students[224].
- James Shore: Five Design Skills Every Programmer Should Have[225].
- Steve Yegge: The Truth About Interviewing[226].

The Mythical 5%

> A commencement address I gave for Neumont University, a school in Salt Lake City dedicated to teaching computer programming.

[219] http://norvig.com/21-days.html
[220] http://samizdat.mines.edu/howto/HowToBeAProgrammer.html
[221] http://news.stanford.edu/news/2005/june15/jobs-061505.html
[222] http://headrush.typepad.com/creating_passionate_users/2005/07/does_college_ma.html
[223] http://www.paulgraham.com/college.html
[224] http://www.joelonsoftware.com/articles/CollegeAdvice.html
[225] http://www.jamesshore.com/Blog/Five-Design-Skills.html
[226] http://steve-yegge.blogspot.com/2006/03/truth-about-interviewing.html

So here you are, about to be unleashed upon the world. This is a lot like taking someone and teaching them all the parts of English speech, then saying "go out and write stories," but it's probably all any school can do. And a more theoretical background wouldn't help; this is not to say theory is bad, because you'll need some eventually. But there's a recent movement that preaches "practice before theory" and I, apparently like Neumont, find that knowing how to do something is very helpful when trying to understand the theory behind it—for me, the theory takes a lot longer to grasp and it's very useful to be functional while my brain is catching up.

There's only so much you can learn in school, and it's only tenuously connected with what you end up doing in your work. So I'm going to try to soften the blow a little, and arm you with a few insights to help when you crash into the actual world of software development.

You've come from very structured learning. We even call it a science. It promises there is the same kind of structure and activities in the world. More importantly, you've come to expect a certain high level of success, similar to what you've experienced with your projects and assignments. But the world does not behave.

The statistics are sobering: 50-80% of programming projects fail. These numbers are so broad because people don't brag about their failures, so we have to guess. In any event, this makes the world sound pretty unreliable. Engineering gets better results, mostly because it must. Bad software usually just annoys people but bad engineering can kill.

An even more fascinating metric is this: 5% of programmers are 20x more productive than the other 95%. If this were a science, like it claims, we could figure out how to get everyone to the same level.

Let's say this follows the 80-20 rule. Roughly 80% of programmers don't read books, don't go to conferences, don't continue learning, don't do anything but what they covered in college. Maybe they've gotten a job in a big company where they can do the same thing over and over. The other 20% struggle with their profession: they read, try to learn things, listen to podcasts, go to user group meetings and sometimes a conference. 80% of this 20% are not very successful yet; they're still beginning, still trying. The other 20% of this 20%—that's about 5% of the whole who are 20x more productive.

So how do you become one of these mythical 5%?

These people are not those who can remember all the moves and have fingers that fly over the keyboard erupting system commands. In my experience those in the 5% must struggle to get there, and struggle to stay there, and it's the process of continuous learning that makes the difference.

Because of what I do, I've met more than my share of these people. They read a lot, and are always ready to tackle a new concept if it looks worthwhile. I think if they do go to conferences they're very selective about it. Most of their time is spent being productive, figuring things out.

The big issue is knowing that you're going after that 20x productivity increase. Which means getting leverage on everything you do. Never just "bashing something out," but using the best tools, techniques, and ideas at your disposal. Always doing your best.

And deeper than that, understanding that the leverage point doesn't always come from the obvious thing, or from what we've been told, or the commonly-held belief about what works. Being able to analyze and understand a situation and discover the hinge points of a problem is essential; this takes a clear mind and detached perspective. For example, sometimes the choice of programming language makes a huge difference, but often, it's relatively unimportant. Regardless, people will still spend all their time on one decision while something else might actually have a far greater influence. Architectural decisions, for example.

So you must learn continuously and teach yourself new technologies, but it's not that simple. It's definitely good to learn more about programming, but you can't *just* learn more about programming. For example here are two of the biggest pain points:

1. Code is read much more than it is written. If people can't read your story, they can't improve it or fix it. Unreadable code has a real cost, and we call it "technical debt."
2. Code reviews are the most effective ways to find software defects, and yet we usually "don't have time for them."

Coming from the world of ones and zeroes we'd like to believe that things are deterministic, that we can provide a set of inputs and get the same outputs every time. I know because I believed this for a long time, and it took some hard knocks when I was a physics undergraduate to begin—only to begin—to wake me up. Many years later I was in a workshop and one of the other attendees told me what I had been learning all those years; she said:

> "All models are wrong. Some are useful."

We are in a young business. Primitive, really—we don't know much about what works, and we keep thinking we've found the silver bullet that solves all problems. As a result, we go through these multi-year boom and bust cycles as new ideas come in, take off, exceed their grasp, then run out of steam. But some ideas seem to have staying power. For example, many of the ideas in agile methodologies seem to be making some real impacts in productivity and quality. This is because they focus more on the issues of people working together and less on technologies.

A man I've learned much from, Gerald Weinberg, wrote his first couple of books on the technology of programming. Then he switched, and wrote or coauthored 50 more on the process of programming, and he is most famous for saying "no matter what they tell you, it's always a people problem."

Usually the things that make or break a project are process and people issues. The way you work on a day-to-day basis. Who your architects are, who your managers are, and who you are working with on the programming team. How you communicate, and most importantly how you solve process and people problems when they come up. The fastest way to get stuck is to think it's all about technology and to believe you can ram your way through the other things. Those other things are the most likely ones to stop you cold.

In my first jobs, I saw lots of managers making stupid decisions, and so, logically, I came to the conclusion that managers and management was stupid. It's a commonly held belief in our profession: if you're not smart enough to deal with the technology, you go into management. Over time I very slowly learned that the task of management wasn't stupid, it's just very, very hard. That's why all those stupid decisions are still made; management is much harder than technology because it involves virtually no deterministic factors. It's all guesswork, so if you don't have good intuition you'll probably make stupid decisions. Napoleon wanted lucky generals rather than smart ones.

Here's an example: some companies have adopted a policy where, at the end of a predetermined period, each team gets evaluated and the bottom 10% or 20% are fired. In response to this policy, a smart manager who has a good team hires extra people who can be thrown overboard without damaging the team. It's not a good policy; in fact it's abusive and eats away at company morale from within. It's one of the things

you probably didn't learn here, and yet the kind of thing you must know, even if it seems to have nothing directly to do with programming.

Here's another example: People will ask you the shortest possible time it will take to finish a particular task. You'll do your best to guess what that is, and they'll assume you can actually do it. What you must tell them for an estimate like this, and for all your estimates, is there's a 0% probability you actually get it done in that period of time, that such a guess is only the beginning of the probability curve. Each guess must be accompanied by such a probability curve, so all the probabilities combined produce a real curve indicating when the project might likely be done. You can learn more about this by reading a small book called Waltzing with Bears[227].

You must pay attention to economics and business, both of which are far-from-exact sciences. Listen to books and lectures on tape while you commute. Understanding the underlying business issues may allow you to detect the fortunes of the company you're working for and take action early. When I first started working I looked askance at people who paid attention to business issues—that was suit stuff, not real technology. But those people were the smart ones.

A great source of information is podcasts. Anyone can produce these so many of them are bad, but there are some real gems out there, podcasts that specifically cover topics in our profession. I learn a lot from these, and they help me stay current.

Here's another thing you probably didn't learn here. Both the world of business and the world of programming is legendary for flailing about with fads that promise to get things done better. The easy fads are patently ridiculous, or become so in short order. The harder ones to spot contain some reason or truth that prevents you from quickly dismissing them. Sometimes you must pick out the good stuff and throw the rest away, and to do this you must exercise critical thinking.

I saw a grocery bag that said "studies show children and teens who eat dinner with their families at least 5 times a week are 50% less likely to use alcohol." The bag's implied conclusion was that eating dinner together prevents alcoholism. Is eating dinner together the dominant factor? Or is there something else that causes both eating dinner together and reduced alcoholism?

Here are some more things which many people have seriously believed:

- Companies don't have to make a profit anymore. It's the new economy.

[227] http://www.systemsguild.com/waltzing.htm

- Real estate always goes up, even if salaries don't.
- A university must be a traditional campus and not an office building.

It's even harder when you come from the world of ones and zeros where we really, really want to believe that everything can be deterministic. It's harder than that when you understand that adding people into the mix and scaling up a system changes the dominant factors, while everyone around you still believes it should all be deterministic.

There's a book that uses studies to debunk beliefs about managing people and projects; it happens to be software-based but the thinking could be applied almost everywhere. This book is called Peopleware [228]; it's small and fun to read and I recommend it to almost everyone. Alas, it really isn't a book of answers, it just firmly destroys many closely-held ideas about how people work in business situations that involve programming. The best thing is it reminds you how easy it is to take a wrong idea and build a bad world around it. It helps you ask questions.

So when you go into your new job with your head filled with technical knowledge from the last couple of years, and you add what I've told you today, you may be tempted to assume that everyone at the company has foolishly gotten themselves trapped with bad ideas. But there's one more very important maxim from Gerald Weinberg which doesn't really answer anything as much as it gives you a way to understand what happens. He says: "Things are the way they are because they got that way … one logical step at a time." It's the legendary frog in the saucepan (Another myth; They actually jump out). So from your fresh new perspective things might look ridiculous, but remember that each decision on the way was made by someone weighing the issues and making what seemed like the best choice at the time. This viewpoint doesn't solve the problem but it can make you more compassionate about people who are stuck there.

You'll need to make many mistakes to figure things out. So be humble, and keep asking questions.

Writing Software Is Like … Writing

I finally figured out the right analogy for software development. Alas, the target audience for this analogy won't be happy with it.

[228] https://en.wikipedia.org/wiki/Peopleware:_Productive_Projects_and_Teams

Why do we need an analogy? We know what we do. We program computers, with all that entails. And we know what that means, because we do it.

But to the stakeholders—managers, CEOs, customers, shareholders, etc.—software development is a mystery. They don't want to know everything about it, but they want to know enough to predict the behavior of software development, at least approximately.

So stakeholders need an abstraction. An analogy. But for the analogy to be useful, it must hide the things that aren't important, and show the things that are. We've been flailing about with this problem for a long time, but we've always been putting it in *our* terms, and it all makes sense to us, so we can't differentiate between a useful analogy and one that is less than helpful.

Mathematicians and engineers were the original programmers, so naturally we tried making it a science, then engineering. Mostly we discovered that no matter how much we want software to be like mathematical proofs or bridge-building, it isn't.

The stakeholders, trying to follow our analogies, asked questions that were important to them. "If programming is like math, why are programs always broken? Math is right or wrong, software is just broken." And later, after we gave up on the science analogy, "If programmers are like engineers, I can replace one engineer with another and get similar results, right?"

This latter has been a huge source of consternation among stakeholders. By and large, engineers have similar productivity levels. And the results produced are verifiable. There's a lot of consistency in engineering, and if we call it "software engineering," then there should be similar consistency in software.

The two typical approaches to this problem have been either big denial ("ignore the differences and pretend all software development is the same") or little denial ("The differences are accidental. We can force consistency").

Big denial just doesn't work. But little denial has produced repeated attempts at "standardization of software engineers," the most notable of which is certification. If only we had a certification process, the argument goes, software engineers would be like real engineers, and they'd all be consistent.

Fortunately certification has never gotten very far, because programmers could never be bothered to take such a thing seriously, and employers want to hire the best programmers without regard to whether they have any particular degrees

or credentials. And the certification programs that *do* exist are always done for money, and that seems to inevitably flatten the curve. I don't know anyone, for example, that takes the basic Sun Java Certification seriously. The more advanced Java certifications seem interesting, but they also appear much more like workshops and less like tests to me.

At one point I ridiculed this attempt to make all programmers identical cogs in a machine by reducing our activity to its simplest behavior in *Programming as Typing* (the following section in this appendix).

So we're not scientists, and we're not engineers. How do we describe what we do to non-programmers in a way that makes sense to them? In particular, in a way that explains why there's such a big difference in programmers, in programming projects, and in the success and failure of projects?

Here's my proposal. I think it explains everything. It is very unsatisfying to stakeholders that want completely predictable behavior, and who want to replace one programmer with another and get identical results. (That's still not going to happen. The only compensation for the unpredictability is approaches like the Agile methods, which increases the bandwidth of communication with the stakeholders).

> We're writers.

Most people can put words together into sentences. They can communicate adequately without being great writers. Most programmers can write some kind of program. It might not be very good, but most companies don't really need it to be very good. Most companies only need basic programming skills. A college degree in computer "science" from anywhere is good enough, and the job is just a job. It doesn't require much in the way of continuing education, conferences, workshops, or someone who is so interested in the craft of programming they are always trying to learn more.

Such people can write, but it's just basic writing. They are not essayists or novelists—and keep in mind there are lots of published articles and novels that are not particularly well-written or worth reading. Obviously such things seem to sell well enough to make the effort and risk worthwhile all around.

But someone who dedicates themselves to writing, who goes through the struggle of figuring it out and discovering their own place in the world—this is a very different

kind of writer (of prose or programs) than the average. This person can produce more functionality faster, and the results are clearer and deeper than ordinary code.

Writing a novel is a very impressive feat. Doing something that might be worth publishing—that's an even greater feat. But the vast majority of published novels aren't worth reading. Only a small fraction of writers create something really worthwhile, and no one, really, knows how they do it. Each good novelist comes to their art in their own way. And what about nonfiction? Every year there are about 5000 novels published, and about 50,000 nonfiction books. Most of those nonfiction books are merely functional, not great reading. But they contain useful information and enough people buy them to make it all worthwhile (to the publisher, at least).

This answers one of the biggest questions—why you can't replace a programmer with just any other programmer and get similar results. It also suggests re-evaluating the kind of project you're creating when you decide who your team should be, and how it will run. The creation of mysteries and young adult fiction and so-called "bodice rippers" and the vast sea of nonfiction books all have their own particular structure and constraints (you'd be surprised at how rigid and controlling publishers are about these things, as if they are manufacturing some kind of basic commodity—"the murder must happen in the first ten pages," etc.). None of these are the mass-market bestsellers ("killer apps") that are sold because of the author's voice and style. The mass-market bestsellers usually don't coincide with great writers because most people don't have the patience to read these meta-craftsmen, just as most programmers don't read the source code for compilers.

Although stakeholders won't necessarily understand the intricate details of the writing and publishing process, they typically understand there are different types of writing, and that the craft of writing is a weird, unfathomable and artistic process which can't guarantee results. So even though "software is writing" is not necessarily going to increase the predictability of what we do, it may at least help non-programmers to understand its unpredictability.

Programming as Typing

> How can we see, through new eyes, the problems we face when building software?

(This started as a reply to John Camara, then took on a life of its own.)

> I believe by now it is safe to assume that the majority of software developers and managers of software projects have come to accept the importance of testing so I feel it's unnecessary to comment on testing.

I've learned not to trust lots of noise about something. It usually doesn't correlate to reality. A friend works for a company that's just now learning about unit testing, and apparently testing in general. I've consulted with companies where testing is still a new thing. My guess is that the majority of software developers have not yet accepted the importance of testing, and it's only the noisemakers on the leading edge who are learning and talking about it, and thus giving the impression it's now well accepted.

Another example of noise vs. reality: people are always saying there are still more COBOL and FORTRAN programmers out there than any other kind. But when was the last time you saw one, much less talked to one? By that metric, they don't exist. But apparently there are lots of them.

> As important as it may be to have a second pair of eyes on a problem I feel it's not the most important benefit of code reviews. I feel that code reviews provide a means of mentoring each other.

Yes, it's one thing to show examples and talk about how you would, in theory, use a particular language feature properly. But when someone actually has a vested interest in a piece of code, that code becomes real and important. (I create toy examples in books out of necessity only).

I think the more abstract the concept, the more important it is to work with a project that people are actually trying to build, to take it out of the realm of ideas. For example, when teaching OO design (which is more abstract than programming), I encourage people to bring their own designs, so we can work on them together. This makes better use of the training or consulting time, because people can actually gain forward motion on their projects and so have more of a vested interest in the seminar.

> This form of mentoring is likely to be the only form of mentoring that the majority of developers experience these days. After all, mentoring has

> lost most, if not all, priority in these sad times of ever decreasing costs at all costs. We have simply forgotten how important it is to pass collective experiences from generation to generation.

This may come from (perceived) efficiency considerations. Mentoring on a regular basis may appear to be just a cost for a project—interference with getting things done. Whereas carving out a week for training between projects is a discrete chunk of time, you do it and you're done, then people can get back to slinging code as fast as they can. Or any number of other scenarios.

I think the problem is that while many programmers understand that programming happens in the mind, and the code itself is just an artifact of the process, outside the field, the code looks like it's what you're doing. (An understandable perception because the code is the core deliverable). So if it's about the code and not the mental process behind the code, it makes sense you would do whatever you can to produce the code as fast and as cheaply as possible, and to discard anything that appears to hinder the creation of code. From this follows the logical thought that coding is typing. If this is true, you want to see people typing all the time. Also, 10 people can type faster than one person so if you can hire ten offshore programmers cheaper than one US programmer, you're going to get a lot more typing done, so you'll get big economic leverage.

In reality, study after study indicates that success comes from who you hire. This suggests that programming is not a mass-production activity where programmers are replaceable components.

I think a good analogy is writing a novel. Suppose you want to create a Stephen King novel (I'm not much of a fan, but this is exactly the kind of book that publishers stay up nights trying to figure out how to mass produce). You can say, "A book is made up of words, and words are created by typing, so to create a book we must get a bunch of people typing. The more people we can get typing, the faster we'll create a book. And the cheaper the typist, the cheaper it is to create books."

It's hard to argue with that logic. After all, a book is made up of words. And words are created by typing, etc. But anyone who reads novels knows there must be a fundamental flaw in the logic, because there are authors whom you like and others whom you can't stand. The choice of words and the structure of the book makes the difference, and that is based on the person writing the book. We know you can't

replace one author with 10 lesser writers and get anything like what the author could produce, or anything you'd want to read.

Another example is a house. Like software, it's comprised of subsystems that fit together. Like software, you have a bunch of people working on it, and it's even true that some of those people are replaceable. It doesn't much matter who is nailing up the wallboard. But you really notice the design of the house, and you notice how well it was put together, and those things are determined by the architect and the builder.

I've been struggling with this general problem for a long time. That is, the "logical" arguments that are very hard to refute, like "software is created by typing." True on the surface, but not really the essence of the issue. But if you keep the argument on those terms, you can't really get anywhere, because the logic is irrefutable. Even if that logic completely misses the real issue.

This is probably why I keep fighting with the static-dynamic language debate, because it has the same feel to me. You can come up with all kinds of reasons that static checking is a good thing, until you have an experience programming in a dynamic language where you are vastly more productive than with a static language. But that experience defies the logic used to back up the reasoning behind static languages (*I believe now it is predominantly cultural rather than logical/rational*).

Here's another one. I think "details matter," and that environmental noise really does wear you down (studies show that noise makes you tired). What I'm talking about here is visual and complexity noise. So I was disappointed when, for example, Ruby turned out to have `begin` and `end` statements, and it uses `new` to create objects. These are all noise artifacts from previous languages, required to support their compilers. If your language creates all objects on the heap, you don't need to say `new` to distinguish between heap and stack objects (like you do in C++, which was mindlessly mimicked by Java). And everyone always indents their code, so you can use indentation to establish scope. Besides the fact that I'm justifying the design minimalism of Python here, when I put these ideas out I will probably get many perfectly reasonable rationalizations about why this is the best way of doing things. And without questioning the fundamental principles upon which those arguments are founded, those arguments are pretty airtight, even if they really come down to "I'm used to that and I don't want to think differently about it."

Java has always required a lot of extra typing. But the fact that Eclipse, NetBeans,

IntelliJ Idea, and other IDEs generate code for you seems to justify enormous amounts of visual noise, and for those in the midst of it, that's OK, and even desirable. "It's clearer because it's more explicit" (Python even has a maxim: "Explicit is better than implicit"). This is even taken to extremes with the idea, supported by a surprising number of folks, that every class should have an associated interface, which to my mind makes the code far more complicated and confusing. Which costs money, because everyone who works with that code must wade through all those extra layers of complication.

All of this detail is expensive, even with a tool that generates a lot of code for you. But if you're in the middle of it, it's all you see and it makes sense because it seems to work. And of course, if you compare one Java project to another, you aren't questioning the cost of using the language.

In contrast, when I teach OO design, my favorite approach is to (A) work on a project that the client is actually working on and (B) move quickly through the design process and model the result in a dynamic language (I know Python best). In most cases, the client doesn't know Python, but that doesn't matter. We still very quickly get a model of the system up and running, and in the process we discover problems that our initial design pass didn't see. So because of the speed and agility of a dynamic language, design issues appear early and quickly and we can refine the design before recasting it in a heavyweight language.

And I would argue that if the initial code is done in the heavyweight language instead, then:

- There is resistance to putting the design into code because it is much more work intensive—it isn't a lightweight activity.
- There is resistance to making changes to the design for the same reason.

And yet, I will probably get any number of perfectly reasonable arguments to the effect that this approach doesn't make sense. I usually find that these arguments are not based on experience, but on logic that follows from fundamental assumptions about the world of programming.

It may not even be possible to prove things logically when it comes to programming. So many of the conclusions that we draw this way appear wrong. This is what I like about the book *Peopleware*, and also *Software Conflict 2.0*. These books point out places where we operate based on what seems perfectly logical, and yet is wrong

(one of my favorite studies in *Peopleware* shows that, of all forms of estimation, the most productive approach is when no estimate at all is made).

The story I heard about Greek natural philosophers (what we call physicists today) is that they were more interested in the arguments about how something worked than they were about how that thing actually worked. So they didn't drop small and large stones to find out if one fell faster than they other, they argued based on their assumptions.

It seems to me that we're in the same situation when we try to argue about programming. A large part of the Enlightenment came from the move to the scientific method, which seems like a small, simple step but turned out to be very big, with very big impact. To wit, you can argue about how you think something will happen, but then you must go out and actually do the experiment. If the experiment disagrees with your argument, you must change your argument and try another experiment.

The key is in doing the experiment, and in paying attention to the results, rather than starting with belief and trying to wrestle the world into line with that belief. Even after some 500 years, human society is still trying to come to terms with the age of reason.

I think the essence of what the agilists are doing is a perfect analogy to the discovery of the scientific method. Instead of making stuff up—and if you look back at all the "solutions" we've invented to solve software complexity problems, that's primarily what they are—you do an experiment and see what happens. And if the experiment denies the arguments you've used in the past, you can't discard the results of the experiment. You have to change something about your argument.

Of course, you aren't forced to change your argument. But even if it doesn't happen overnight, those that look at the experiments and realize that something is different than the way they thought it was, those people will move past you and forge into new territory. Territory that your company cannot enter if they refuse to change their ideas.

Do What You Love

"In 1960, a researcher interviewed 1500 business-school students and classified them in two categories: those who were in it for the money—1245 of them—and those who

were going to use the degree to do something they cared deeply about—the other 255 people. Twenty years later, the researcher checked on the graduates and found that 101 of them were millionaires—and all but one of those millionaires came from the 255 people who had pursued what they loved to do!

"Now, you may think that your passion for Icelandic poetry of the baroque period, or butterfly collecting, or golf—or social justice—might consign you to a permanent separation between what you love and what you do for a living, but it isn't necessarily so. Vladimir Nabokov, one of the greatest novelists of this century, was far more passionate about butterfly collecting than writing. His first college teaching job, in fact, was in lepidoptery. Research on more than 400,000 Americans over the past 40 years indicates that pursuing your passions—even in small doses, here and there each day—helps you make the most of your current capabilities and encourages you to develop new ones."—From *The Other 90%* by Robert K. Cooper, Three Rivers Press 2001.

Also, see Po Bronson's book *What Should I Do With My Life?* for more exploration of these ideas.