Java 9.0 to 17.0 Cookbook

A Roadmap with Instructions for the Effective Implementation of Features, Codes, and Programs



Tejaswini Jog
Mandar Jog

bpb

Java 9.0 to 17.0 Cookbook

A Roadmap with Instructions for the Effective Implementation of Features, Codes, and Programs



Tejaswini Jog
Mandar Jog

bpb

Java 9.0 to 17.0 Cookbook

A Roadmap with Instructions for the Effective Implementation of Features, Codes, and Programs

Tejaswini Jog

Mandar Jog



www.bpbonline.com

Copyright © 2023 BPB Online

All rights reserved. No part of this book may be reproduced, stored in a retrieval system, or transmitted in any form or by any means, without the prior written permission of the publisher, except in the case of brief quotations embedded in critical articles or reviews.

Every effort has been made in the preparation of this book to ensure the accuracy of the information presented. However, the information contained in this book is sold without warranty, either express or implied. Neither the author, nor BPB Online or its dealers and distributors, will be held liable for any damages caused or alleged to have been caused directly or indirectly by this book.

BPB Online has endeavored to provide trademark information about all of the companies and products mentioned in this book by the appropriate use of capitals. However, BPB Online cannot guarantee the accuracy of this information.

First published: 2023

Published by BPB Online

WeWork

119 Marylebone Road

London NW1 5PU

UK | UAE | INDIA | SINGAPORE

ISBN 978-93-5551-238-3

www.bpbonline.com

Dedicated to

Our Lovely Son

OJAS

&

***Neko** (Our Pet)*

About the Authors

Tejaswini Jog currently works as a Java Consultant Trainer. Her areas of expertise are Spring and Microservices. She started her journey as Java trainer almost 15 years ago. She has a rich experience imparting training to corporate clients. Few of her clientele includes Wipro, Accenture, Citi Bank and so on. Along with training, she had also authored books like ‘Learning Spring 5.0’, ‘Reactive Programming with Java 9’ among others.

Mandar Jog is a passionate Java Trainer. With more than 20 years of experience as a trainer, Mandar is well versed with imparting training to freshers as well as experienced candidates. He is always keen to build the skill set of freshers so as to make them ready to get placed. He is also author of the books like “Java 9.0 to 13.0” and “Professional Java Interview Guide”. He also works as corporate trainer for companies like Zensar, Accenture, eClerx and others.

About the Reviewers

Paulo Lopes is a Principal Software Engineer and profound evangelist of event-driven platforms with a proven record on Java high-performance distributed-systems within Financial, Telecom, Video Game and Geo-Spatial businesses. He is an Open-source maintainer for Eclipse Vert.x with vast experience on Web and Security standards, such as FIDO2/webauthn, Oauth2 and OpenID Connect. He is also member of the W3C WebAuthn Working Group and a strong advocate for secure software development, author of reactive drivers for Redis compatible databases, and skilled Relational Database developer for PostgreSQL and MySQL.

Prashant Kumar has been working in the software industry for 11 years. Currently, he works as a Data engineer at IBM and is involved in interesting big data applications. He started as a Java developer but now works with both Java and Python, depending on the nature of work. He has also worked as consultant for enterprise content management systems. When he is not working, Prashant likes to read books and is a big football fan.

Acknowledgement

Writing a book is never an easy task for sure. It requires a lot of team work and coordination to make it happen. During this entire process of selecting a book title and getting it published, tremendous support was provided by entire team of BPB Publications. We are thankful to ENTIRE TEAM.

We would like to thank our parents and friends, who constantly encouraged us to share our knowledge to others through this book.

Preface

This book is written for everyone who is keen to know the features introduced in different versions of Java. Earlier, Java versioning was not that critical to know, as the versions were used to be released after a few years. So, once you learn one version, you can use that version without upgrading to a new concept for a longer duration. However, Oracle launched the concept of Time-Based Version Release model, because of which, after every six months, new version is released. So, understanding the new features and keeping ourselves updated is becoming one of the most important aspects if we are Java developers.

This book is totally practical oriented. Throughout this book, you will find different problem statements connected to different features in the particular version. Though the concepts are discussed, they are discussed through problem statements. Every problem statement is aligned with solution and its explanation, to help the readers grasp the concepts more practically, rather than conceptually.

This book is divided into 8 different chapters. Each of the chapter is dedicated to specific version starting from Java 9 to Java 17. The book covers almost all the features that are important to Java developers. Every feature is explained with problem statements and its solution using that feature. The details of each chapters are listed as follows.

Chapter 1 will cover the updates from Java 9.0. In this chapter, readers will get to know different important features such as private methods in interfaces, updates in stream and collection API. Along with this, it also covers the usage of JShell and ObjectInputFilter. Most importantly, the chapters provide the practical implementation of how to use the module system in Java, which is the most important feature of Java 9.

Chapter 2 will cover the important features from Java 10. In this chapter, readers will learn how to use the local variable type inference and why to use it. Java 10 also added some additional functionalities to Collection API such as the copyOf() method and Stream API, such as creating unmodifiable collection and so on. The chapter will give complete details along with implementation for all such features.

Chapter 3 will cover the features that were introduced as a part of Java 11. Java 11 comes with modified API for reflection. Also, there are some methods

introduced in String API using which, we can work with string more precisely than earlier. The concepts such as local variable syntax for lambda expressions and process to launch the single-file source-code program is also becoming famous in the software industry.

Chapter 4 will cover the features from Java 12. Java 12 has provided some important features such as Teeing Collectors in Stream API. Moreover, there are some additional functionalities introduced to work with Strings.

Chapter 5 will cover the updates from Java13 as well as Java14. As there are not much of the changes in each of the version individually, in this chapter, both the versions are covered. Java13 introduced some additional functionalities in FileSystem and XML Parsers. In addition, Java 14 provided some modifications in switch-case, which we will be discussing in detail.

Chapter 6 will cover all the features introduced as a part of Java 15 version. The major changes in this version are the introduction to text blocks, hidden classes and added functionalities for CharSequence interface.

Chapter 7 will cover the features from Java 16 version. In Java 16, there are many features like invoking default methods from proxy, Record classes and so on. Moreover, from this version onwards, we can use pattern matching for instanceof operator. Along with this, there are some minor changes in the Stream API to make it more flexible.

Chapter 8 will cover major features from Java 17. In this version, Java introduced some new algorithms in Random Number Generators. Moreover, there is a feature now, using which, you can filter the process of deserialization. Along with this, there are the other concepts such as Vector and Reflection API for Sealed classes etc.

Code Bundle and Coloured Images

Please follow the link to download the
Code Bundle and the ***Coloured Images*** of the book:

https://rebrand.ly/6vgeust

The code bundle for the book is also hosted on GitHub at **https://github.com/bpbpublications/Java-9.0-to-17.0-Cookbook**. In case there's an update to the code, it will be updated on the existing GitHub repository.

We have code bundles from our rich catalogue of books and videos available at **https://github.com/bpbpublications**. Check them out!

Errata

We take immense pride in our work at BPB Publications and follow best practices to ensure the accuracy of our content to provide with an indulging reading experience to our subscribers. Our readers are our mirrors, and we use their inputs to reflect and improve upon human errors, if any, that may have occurred during the publishing processes involved. To let us maintain the quality and help us reach out to any readers who might be having difficulties due to any unforeseen errors, please write to us at :

errata@bpbonline.com

Your support, suggestions and feedbacks are highly appreciated by the BPB Publications' Family.

Did you know that BPB offers eBook versions of every book published, with PDF and ePub files available? You can upgrade to the eBook version at **www.bpbonline.com** and as a print book customer, you are entitled to

a discount on the eBook copy. Get in touch with us at: **business@bpbonline.com** for more details.

At **www.bpbonline.com**, you can also read a collection of free technical articles, sign up for a range of free newsletters, and receive exclusive discounts and offers on BPB books and eBooks.

Piracy

If you come across any illegal copies of our works in any form on the internet, we would be grateful if you would provide us with the location address or website name. Please contact us at **business@bpbonline.com** with a link to the material.

If you are interested in becoming an author

If there is a topic that you have expertise in, and you are interested in either writing or contributing to a book, please visit **www.bpbonline.com**. We have worked with thousands of developers and tech professionals, just like you, to help them share their insights with the global tech community. You can make a general application, apply for a specific hot topic that we are recruiting an author for, or submit your own idea.

Reviews

Please leave a review. Once you have read and used this book, why not leave a review on the site that you purchased it from? Potential readers can then see and use your unbiased opinion to make purchase decisions. We at BPB can understand what you think about our products, and our authors can see your feedback on their book. Thank you!

For more information about BPB, please visit **www.bpbonline.com**.

Table of Contents

1. Java 9 – Turning the Wheels
- Introduction
- Structure
- Objectives
- Installation
- Private method interfaces
- Stream API update
- Collection API update
- Updates in the Arrays class
- Improved optional interface
- ObjectInputFilter interface
- Flow API
- Updates in CompletableFuture
- Java Platform Module System
- Conclusion
- Key terms
- Questions

2. Java 10 – Crack of a Dawn
- Introduction
- Structure
- Objectives
- Installation
- Local Variable Type Inference
- Collection API enhancements
- Stream API enhancement
- Optional Interface Enhancement
- Application Class Data Sharing
- Conclusion
- Questions
- Key Terms

3. Java 11 – Crack of a Dawn
- Introduction

Structure
Objectives
Installation
Nest-based access control
Updates in the reflection API
String API updates
Updates in the reflective access of the nested class
Local-variable syntax for lambda parameters
HttpClient (Standard)
Launch single-file source-code program
Conclusion
Questions
Keywords

4. Java 12 – Performance is the Key
Introduction
Structure
Objectives
Installation
Collectors.teeing()
String API updates
Updates in NIO
Updates in the CompletableFuture interface
Compact Number Formatting
Conclusion
Keywords
Questions

5. Java 13 and 14 – Friends Forever
Introduction
Structure
Objectives
Part I: What's new in Java 13?
Installation
Updates in java.nio.file.FileSystems
Updates in XML Parsers
Part II: Updates in Java 14
Installation

Updates in Switch-Case
Better Approach of the NullPointerException description
Using pre Java14 compiler
Using the Java 14 compiler
Plural Support in the Compact Number Format
Conclusion
Key terms
Questions

6. Java 15 – I am 25 Years Old
Introduction
Structure
Objectives
Installation
Using text blocks
Hidden classes
Invoking the static method
Invoking the non-static/instance method
Improvement in the CharSequence interface
Conclusion
Key terms
Questions

7. Java 16 – Turning the Wheels
Introduction
Structure
Objectives
Installation
Default method invocation from Proxy
Improved Date - Time API
Modified Stream API
Pattern matching in the instanceof operator
Record classes
Conclusion
Keywords
Questions

8. Java 17 – Journey is Not Over Yet

Introduction
Structure
Objectives
Installation
Improvements in the Random Number Generator Algorithm
Deserialization filtering
Modified switch case
Reflection API for a sealed class
Vector API
Terminologies in the Vector API
Subclasses of the Vector
Conclusion
Keywords
Questions

Index

CHAPTER 1

Java 9 – Turning the Wheels

Introduction

When you explore a new path; the journey is always exciting and sometimes full of surprises. Over the years, when we all walked on the banks of a well-designed and so-called structured Java, we hardly thought about restructuring the Java language itself. We are all amazed at how Java keeps updating the API to create flexible, maintainable and modular applications. The developers like you and I were more than happy to code in the traditional Java approach. But then everything has to be better, it needs to be improved. The Greek philosopher Heraclitus had once said "*Change is the only constant in life.*" That is true with programming languages as well. Every programming language keeps on updating itself with some new features that will make the programming simpler to cater to the business needs. With that goal in mind, many of you already know how Java 8 completely changed the way you code by introducing functional interfaces, lambdas, and so on. Java 8 was said to be one of the major changes in the API after JDK 5.0. There were many improvements in the API as well, which were appreciated by the software industry. But at the same time, Java API developers knew that there is a lot of work to do in terms of reliability, maintainability along with improving the performance. Keeping that focus in mind, Java 9 was released on Sept 22, 2017.

Similar to Java 8, Java 9 was also said to be one of the major developments as far API updates are considered. It has more than 150 new features which include the module system as well. The module system is a paradigm shift in Java, as it changed the traditional approach of package-centric programming to modules. Apart from this, there are a few updates in the APIs like stream, Optional, concurrency, and so on. In this chapter, we will learn many of these concepts through problem statements and their solutions.

Structure

In this chapter, we will cover the following concepts:

- Private method interfaces
- Stream API updates
- Collection API updates
- Updates in Arrays
- Improved Optional interface
- REPL in Java 9 (JShell)
- ObjectInputFilter interface
- Flow API
- Updates in CompletableFuture
- Java Module System

Objectives

After studying this unit, you will be able to learn the different features of Java 9. We will solve a few problems using which we will learn different concepts such as interface private methods, updates in the stream API, the Flow API, and so on.

Installation

Before you start reading this chapter, download Java SE Development Kit Version 9 from following resource: **https://www.oracle.com/java/technologies/javase/javase9-archive-downloads.html.**

Private method interfaces

Java 9 introduced an interesting feature wherein you can add private methods to an interface. This makes the behavior of interfaces more robust and maintainable.

Problem

We are using interfaces for our application. We are using multiple default methods in that interface. All these methods use the code to connect to resources which we want to utilize for our business logic. Is there any other way to reduce the redundant code that will occur across all the default methods?

Solution

To achieve this, follow the given steps:

1. Create an interface with the name **`PrivateMethodInterface`**.
2. Add at least 2 default methods.
3. Add a private method containing the common code to be shared by 2 default methods declared in an interface.
4. Create an application class with public static void main to invoke these methods.

`Listing 1-1` shows the declaration of **`PrivateMethodInterface.java:`**

```
1. //Listing 1-1
2. package com.java9.defaultmethod;
3.
4. import java.io.FileInputStream;
5. import java.io.IOException;
6.
7. public interface PrivateMethodInterface {
8. default void method١() {
9.   System.out.println("In method 1");
10.   try {
11.    utilResource();
12.   } catch (IOException e) {
13.    // TODO Auto-generated catch block
14.    e.printStackTrace();
15.   }
16. }
17.
18. default void method٢() {
19.   System.out.println("In method 2");
20.   try {
21.    utilResource();
```

```
  } catch (IOException e) {
   // TODO Auto-generated catch block
   e.printStackTrace();
  }

}

// private method for sharing common resource across
// default methods
private void utilResource() throws IOException {
  System.out.println("==Reading common resources== ");
  FileInputStream fis = new FileInputStream("data.dat");
  while (fis.read() != -1) {
   // Some business logic
  }
}
}
```

Listing 1-2 is **PrivateMethodInterfaceDemo.java** which is the application class to invoke the methods from the interface:

```
//Listing 1-2
package com.java9.defaultmethod;

public class PrivateMethodInterfaceDemo implements PrivateMethodInterface{

public static void main(String[] args) {
  // TODO Auto-generated method stub
            PrivateMethodInterfaceDemo demo= new PrivateMethodInterfaceDemo();
  demo.method1();
  demo.method2();
```

11. }
12. }

Output

If we execute **`PrivatemethodInterfaceDemo.java`**, we will get the output as shown in *Figure 1.1*:

```
In method 1
==Reading common resources==
In method 2
==Reading common resources==
```

Figure 1.1: *Private method interface*

Explanation

Java 9 introduced private methods which can be added in the interface. Earlier we had default methods in the interface. But the limitation of using default methods for common code is that such methods can be overridden by a subclass, which will change the common behavior that is applied in the interface. So, private methods serve both purposes. It reduces redundancy as well as makes your common code more secure.

In our **`Listing 1-1`**, at line number 29, we have implemented a private method **`utilResource()`**. This private method is invoked internally by two default methods: **`method1()`** and **`method2()`**. So, this reduces the data redundancy in the code and at the same time, it is more maintainable.

We can also declare the private methods as static as follows:

```
private static void utilResource(){
  //implementation code
}
```

The advantage of this method is that it can be invoked by both static default methods and non-static default methods.



- Private methods in interfaces are used to share the common code across default or static methods to avoid redundancy.
- Private methods can be static or non-static.

Stream API update

Stream is a powerful feature of Java 8. It provides an efficient way to the iterator and operates the collections through the concept of functional programming. Java 9 provided a few more updates in streams by which you can conditionally choose the starting and ending of the stream elements iteration.

The following new methods are added to the Stream API:

- takeWhile()
- dropWhile()
- iterate()

Problem

We have a list of products, the category of which is essentially "*Garments*". We would like to stream through it and increase the value of all the garment products by some percent. But we are not sure that the list we are getting consists of only the garment-type products. Is there any other efficient way to check this?

Solution

This can be achieved by using the **`takeWhile()`**. Follow the steps as listed:

1. Create a POJO class **`Product.java`** as shown in **`Listing 1-3`** in our repo. This class consists of **`productName`**, **`productCategory`**, and **`productPrice`**. The code snippet is as follows:

```
1. //Listing 1-3
2. package com.java9.stream;
3.
4. public class Product{
5.   private String productName;
6.   private String productCategory;
7.   private double productPrice;
8.   …
9. }
```

2. Create a class **`TakeWhileDemo.java`** as shown in **`Listing 1-4`**. This class instantiates the list of products. The **`takeWhile()`** method is invoked on the

stream generated by this list to make sure that we get only the "Garment" product:

```
1. //Listing 1-4
2. package com.java9.stream;
3. 
4. import java.util.Arrays;
5. import java.util.List;
6. public class TakeWhiledemo {
7.   public static void main(String[] args) {
8.    List<Product> productList=Arrays.asList(
9.      new Product("Denim Jeans","Garment",1500.00),
10.     new Product("T shirt","Garment",500.00),
11.     new Product("Nike","Sports",0···.··),
12.     new Product("Kurtis","Garment",0··.··));
13. 
14.   productList.stream()
15.                                          .takeWhile(e-
   >e.getProductCategory().equals("Garment"))
16.    .forEach(System.out::println);
17. 
18.  }
19. }
```

Output

If we execute **`TakeWhileDemo.java`**, we will get the output as shown in *Figure 1.2*:

```
Denim Jeans , Garment , 1500.0
T shirt , Garment , 500.0
```

Figure 1.2: *Implementing takeWhile()*

The **`takeWhile()`** functions take a Predicate. If that fails, it is a "*stop-the-chain*" process for streams. In this program, the moment the first product does not

match the category, the stream operation is terminated. It does not check the next elements at all. So, you will get the products Denim Jeans and T shirt but you will not get Kurtis. As the category of Nike is not garment, according to the predicate applied to **`takeWhile()`**, the iteration is terminated.

Explanation

The **`takeWhile()`** method was introduced in Java 9 which is used to break the iteration of stream elements. This method is, in fact, used to apply logical or conditional breaks on the elements of the streams.

Earlier versions, that is, Java 8, provided methods like **`limit()`** and **`filter()`**. But **`limit()`** takes the argument of the integer which limits the number of elements given as an output. On the other hand, **`filter()`** takes a predicate similar to **`takeWhile()`**. But the behavior of **`filter()`** is different than **`takeWhile()`**. Unlike **`takeWhile()`**, the method **`filter()`** does not terminate the iteration based on the condition. It iterates through the entire stream.

The syntax of the method is as follows:

```
default Stream<T> takeWhile(Predicate<? super T> predicate)
```

It takes the argument of type **`Predicate`** as a condition to check whether to break the iteration or continue further.

The method returns:

- The elements matching the Predicate are provided if the stream is ordered.
- The elements, up to the element which does not match according to the Predicate provided if the stream is not ordered.

We have invoked this method in our **`Listing 1-4`**, at line number 15. Though we have 3 products with "*Garment*" as type, we get only the first 2 as the third item in the stream is not of type "*Garment*". So, the stream applies the break for further processing.

Let us look at the following snippet:

```
productList.stream()
  .filter(e->e.getProductCategory().equals("Garment"))
  .forEach(System.out::println);
```

This would return you the products Denim Jeans, T shirt and Kurtis, as when you apply the filter, it is applied to all the elements.



The output generated from the **`filter()`** or **`takeWhile()`** method is nondeterministic if the streams are generated from unordered collections like **`Set`**.

Consider the snippet as shown:

```
Set<Integer> numbers=Set.of(10,20,30,100,50,55, 60,40,15);
numbers.stream().takeWhile(e->e<55).forEach(System.out::println);
```

In this case, the output is not guaranteed at all, as the sequence of streams itself is not predictable. This can be controlled by invoking the **`sorted()`** method on the stream.



- If you want to break the iterations of stream elements based on the first occurrence where the conditin failed, you can use the takeWhile() method.
- The method takeWhile() takes Predicate as an argument, and the iteration is terminated when the test condition fails for a particular element.

Problem

In the banking application, we have a list of customers of the bank. While they do the transactions, their balance gets updated. We have a list of all such customers and we want to group out only those customers whose balance is more than 500 $. How can we achieve the same?

Solution

This problem can be solved by using the **`dropWhile()`** method introduced in Java 9. Perform the following steps:

1. Create a POJO class **`Customer.java`** as shown in **`Listing 1-5`**:

```
1. //Listing 1-5
2. public class Customer implements Comparable<Customer>{
3.    int customerId;
4.    String customerName;
5.    double customerBalance;
6.    …
7.    }
```

2. Create **DropWhileDemo.java** as shown in **Listing 1-6**. This class creates a list of customers and drops the customers in the stream till the **customerBalance< 500**:

```
1. //Listing 1-6
2. package com.java9.stream;
3.
4. import java.util.Arrays;
5. import java.util.Iterator;
6. import java.util.List;
7. import java.util.function.Predicate;
8. import java.util.stream.Collectors;
9. public class DropWhileDemo {
10.   public static void main(String[] args) {
11.    List<Customer> customerList=Arrays.asList(
12.      new Customer(101,"Alex Kerry",440),
13.      new Customer(102,"John Smith",55),
14.      new Customer(103,"Alan Willis",770),
15.      new Customer(104,"Mark Flex",667));
16.
17.    List<Customer> eligibleCustomers=
18.     customerList.stream()
19.      .sorted()
20.      .dropWhile(e->e.getCustomerBalance()<500)
21.      .collect(Collectors.toList());
22.
23.    Iterator<Customer> itr=eligibleCustomers.iterator();
24.    while(itr.hasNext()) {
25.     System.out.println(itr.next());
26.    }
27.   }
28. }
```

Output

If we execute **`DropWhileDemo.java`**, we will get the output as shown in *Figure 1.3*:

```
104 , Mark Flex , 667.0
103 , Alan Willis , 770.0
```

Figure 1.3: *Implementing dropWhile()*

Explanation

In this scenario, you have given a list that is not sorted. The expectation is you sort the stream and then apply the **`dropWhile()`**. Otherwise, the results from **`dropWhile()`** are unpredictable.

The method **`dropWhile()`** is introduced to skip the elements in the stream based on some condition. Java 8 provided a **`skip()`** method in the stream API. But this method is not conditional. You can skip the first 'n' number of elements in the stream which is provided as an argument in the **`skip()`** method.

The syntax of the method is as follows:

```
default Stream<T> dropWhile(Predicate<? super T> predicate)
```

The method takes the **`Predicate`** as an argument, a condition based on which the elements are skipped.

The method returns stream of elements, after skipping the first element, matched with the predicate provided in an argument.

In our solution, we have invoked the **`dropWhile()`** method on the stream of the customer list on line number 19 as shown in **`Listing 1-6`**. This will generate the list of only those customers whose current balance is equal to or more than 500 $. The elements which match the **`Predicate`** implemented in **`dropWhile()`** are dropped in the stream operations.



- If you want to start your iteration of stream, by skipping some of the elements from starting position, you can invoke the dropWhile(Predicate p) function.
- This method drops all the elements till the Predicate provided in the argument returns true. The iteration will start from the first element for whom the predicate returns false.

Problem

How to use the **`iterate()`** method which is introduced in the Stream API?

Solution

To implement the **`iterate()`** method, create the **`IterateDemo.java`** as shown in the following **`Listing 1-7`**:

```
1. //Listing 1-7
2. package com.java9.stream;
3.
4. import java.util.stream.Stream;
5. public class IterateDemo {
6.
7.   public static void main(String[] args) {
8.    // iterate() without Predicate : Java 8
9.           System.out.println("Java   8   style   :   iterate()
   without  Predicate: ==>");
10.    Stream.iterate(101, i -> i + \).limit(10)
11.      .forEach(number->System.out.print(number+" "));
12.
13.    // iterate() with Predicate : Java 9
14.          System.out.println("\nJava   ९   style   :   iterate()
   with  Predicate: ==>");
15.
16.    Stream.iterate(101, i -> i <= \\·, i -> i + \)
17.      .forEach(number->System.out.print(number+" "));
18.   }
19. }
```

Output

If we execute **`IterateDemo.java`**, we will get the output as shown in *Figure 1.4*:

```
Java 8 style : iterate() without Predicate: ==>
101 102 103 104 105 106 107 108 109 110
Java 9 style : iterate() with Predicate: ==>
101 102 103 104 105 106 107 108 109 110
```

Figure 1.4: *Implementing iterate()*

Explanation

Creating a stream of sequence is sometimes the need of your application. In earlier versions (prior to Java 8), we used the traditional for loop to generate the sequence, which needed to be passed to the collection. But from Java 8, you can create the stream of sequence using the **`iterate()`** method. This method in Java 8 was infinite and we had to apply the limit for the generated sequence. Java 9 provided an additional method, where you can add a Predicate in an argument to control the sequence generation.

The syntax of the **`iterate()`** method is as follows:

```
static <T> Stream<T> iterate(T start,
  Predicate<? super T> hasNext, UnaryOperator<T> step)
```

The method returns:

- The stream with the starting element as start, till the Predicate provided in the second argument returns false.
- The iteration from the start to end element is controlled by the step provided as a third argument.
- This method also contains the overloaded form, where you can skip the Predicate in the second argument. But this will return the infinite stream in sequential order. This form of method is available in Java 8.

In our solution, we have used the **`iterate()`** method at line number 10:

```
Stream.iterate(101, i -> i <= \\·, i -> i + \)
  .forEach(number->System.out.print(number+" "));
```

This creates a Stream which starts from 101. It increases its value by 1, till the value reaches 110. Once the value reaches 111, the Predicate implemented in the second argument, that is, `i ->i<=110`, returns false and the sequence is terminated.



- iterate() is a static method of Stream.
- This method is used to generate the sequential or ordered Streams starting from the value given as first argument, till the Predicate fails which is written in the second argument.
- It changes the value of the element as per the expression written in the third argument.

Collection API update

Collections were always an integral part of any Java version right from the beginning of Java. Over the years, different classes were introduced in the Collection API which were used in different scenarios and they were very famous. In Java 9, the Collection API is modified slightly to add some methods which makes the process of initialization of collection simpler and readable.

Problem

We are using the collection API for long period of time. But when it comes to initializing the collection, the process really looks very verbose. How to make use of Java 9 features to simplify this process to create unmodifiable collections?

Solution

To solve this, we can make the use of static factory methods of collections which are introduced in Java 9.

We will first observe the factory methods of List. To achieve this, follow the given steps:

1. Create a simple POJO class **`Employee.java`** as shown in **`Listing 1-8`**:

```
1. //Listing 1-8
2. package com.java9.collection;
3.
4. class Employee{
5.   int empId;
6.   String name;
7.   public Employee(int empId, String name) {
```

```
    super();
    this.empId = empId;
    this.name = name;
   }

   @Override
   public String toString() {
    return "Employee [empId=" + empId + ", name=" + name + "]";
   }

}
```

2. Use the factory methods for creating the unmodifiable list, as shown in **Listing 1-9**:

```
//Listing 1-9
package com.java9.collection;

import java.util.List;

public class UnmodifiableList {
   public static void main(String[] args)
   {
    //creating empty List
    List<Employee> empList١=List.of();

    //creating List with one object
                    List<Employee>      empList٢=List.of(new Employee(101,"William Smith"));

    //creating list with multiple objects, can accept upto 10 elements
```

```
16.                     List<Employee>      empList۳=List.of(new
    Employee(101,"William Smith"),
17.             new Employee(102,"Rakesh Ahuja"),
18.             new Employee(103,"David Monte"));
19.     System.out.println("Printing empty list:==>");
20.     empList۱.forEach((e)->System.out.println(e));
21.
22.     System.out.println(("Printing list of 1 element ==>"));
23.     empList۲.forEach((e)->System.out.println(e));
24.
25.          System.out.println("Printing   list   of   multiple
    elements  (upto 10)==>");
26.     empList۳.forEach((e)->System.out.println(e));
27.
28.   }
29. }
```

The output of the preceding code is shown in *Figure 1.5*:

```
Printing empty list:==>
Printing list of 1 element ==>
Employee [empId=101, name=William Smith]
Printing list of multiple elements (upto 10)==>
Employee [empId=101, name=William Smith]
Employee [empId=102, name=Rakesh Ahuja]
Employee [empId=103, name=David Monte]
```

Figure 1.5: *Creating un-modifiable List*

Similar updates are introduced in **`Set`**, which return the unmodifiable **`Set`**. Look at the code snippet shown here from **`Listing 1-10`**:

```
1. //Listing 1-10
2. package com.java9.collection;
3.
4. import java.util.Set;
5.
```

```
6. public class UnmodifiableSet {
7.   public static void main(String[] args)
8.   {
9.    //creating empty Set
10.    Set<Employee> empSet١=Set.of();
11.
12.    //creating Set with one object
13.        Set<Employee>  empSet٢=Set.of(new  Employee(101,"William
   Smith"));
14.
15.     //creating Set with multiple objects, can accept upto 10
   elements
16.        Set<Employee>  empSet٣=Set.of(new  Employee(101,"William
   Smith"),
17.             new Employee(102,"Rakesh Ahuja"),
18.             new Employee(103,"David Monte"));
19.
20.    System.out.println("Printing empty Set ==>");
21.    empSet١.forEach(e->System.out.println(e));
22.
23.    System.out.println("Printing set with 1 element ==>");
24.    empSet٢.forEach(e->System.out.println(e));
25.
26.         System.out.println("Printing  set  with  multiple(upt0
   10)  elements ==>");
27.    empSet٣.forEach(e->System.out.println(e));
28.   }
29. }
```

The output of this program is shown in *Figure 1.6*:

```
Printing empty Set ==>
Printing set with 1 element ==>
Employee [empId=101, name=William Smith]
Printing set with mutiple(upt0 10) elements ==>
Employee [empId=102, name=Rakesh Ahuja]
Employee [empId=101, name=William Smith]
Employee [empId=103, name=David Monte]
```

Figure 1.6: *Creating un-modifiable Set*

Where in, a **`Map`** works little differently. As the **`Map`** contains the key and value pair, it will form one element from two arguments. So, you can create a **`Map`** of one element; it will need two arguments. The first argument is the key and the second argument is the element. This code is illustrated in **`Listing 1-11`** as follows:

```
1. //Listing 1-11
2. package com.java9.collection;
3. 
4. import java.util.Map;
5. import static java.util.Map.entry;
6. 
7. public class UnmodifiableMap {
8.   public static void main(String[] args) {
9.    //creating empty Set
10.    Map<Integer, Employee> empMap١=Map.of();
11. 
12.    //creating Set with one object
13.             Map<Integer,Employee>    empMap2=Map.of(1,    new
   Employee(101,"William Smith"));
14. 
15.     //creating Set with multiple objects, can accept upto 10
   elements
16.    Map<Integer, Employee> empMap٣=Map.of(
            1,new Employee(١٠١,"William Smith"),
17.            2,new Employee(١٠٢,"Rakesh Ahuja"),
```

```
18.             3,new Employee(١٠٣,"David Monte"));
19.
20.    //creating arbitrary number of elements in Map
21.    Map<Integer,Employee> empMap4=Map.ofEntries(
22.         entry(1, new Employee(101,"William Smith")),
23.         entry(2, new Employee(102, "Rakesh Ahuja")),
24.         entry(3, new Employee(103, "David Monte"))
25.         );
26.
27.    System.out.println("Printing empty Map ==>");
28.    empMap١.forEach((k,v)->System.out.
                    println("key="+k+", value=" +v));
29.
30.    System.out.println("Printing map of one element ==>");
31.           empMap٢.forEach((k,v)->System.out.println("key="+k+",
    value=" +v));
32.
33.    System.out.println("Printing map of multiple(upto 10)
                            elements ==>");
34.           empMap٣.forEach((k,v)->System.out.println("key="+k+",
    value=" +v));
35.
36.    System.out.println("Using Map.ofEntries ==>");
37.           empMap٤.forEach((k,v)->System.out.println("key="+k+",
    value=" +v));
38.   }
39. }
```

The output of this program is shown in *Figure 1.7*:

```
Printing empty Map ==>
Printing map of one element ==>
key=1, value=Employee [empId=101, name=William Smith]
Printing map of multiple(upto 10) elements ==>
key=3, value=Employee [empId=103, name=David Monte]
key=2, value=Employee [empId=102, name=Rakesh Ahuja]
key=1, value=Employee [empId=101, name=William Smith]
Using Map.ofEntries ==>
key=3, value=Employee [empId=103, name=David Monte]
key=2, value=Employee [empId=102, name=Rakesh Ahuja]
key=1, value=Employee [empId=101, name=William Smith]
```

Figure 1.7: *Creating un-modifiable Map*

Explanation

The Collection API in Java is one of the most fascinating APIs provided by Java. From early versions of Java, Collections are an integral part of the Java API. Over the years, many new classes and features are added to this API to improve the application development in Java.

But still initializing a collection was always a tedious job till version 8 of Java. So, if we want to initiate a List of String in Java, we need to write as follows:

```
List<String> strList=new ArrayList<String>();
strList.add("John");
strList.add("Dave");
strList.add("Vivian");
```

This creation of List is very verbose and noisy. If we want to create a List of only a few elements, this code really looks lengthier than needed. So, Java alternatively had provided another approach of creating a List by using the **`Arrays.asList()`** method as follows:

```
List<String> strList=Arrays.asList("John", "Dave", "Vivian");
```

Though this looks less verbose, it is obscure. It is complex and confusing to read. It gives a feeling that we are creating an array rather than a list.

Further, if we need to convert it to an unmodifiable List, we need to invoke the utility method of Collections as follows:

```
stringList = Collections.unmodifiableList(stringList);
```

This situation is identical, even if we talk about **`Set`** and **`Map`**. So, to overcome this, Java 9 introduced a feature where they provided the static factory methods for **`Collections`**, which returns you an unmodifiable collection.

The syntax of these methods for each type of collections is as follows:

1. List

   Following methods are introduced to create the collection of type List:

   - `List.of()`
   - `List.of(s1)`
   - `List.of(s1, s2)  // fixed-argument form overloads up to 10 elements`
   - `List.of(elements…)  // varargs form supports an arbitrary number of elements or an array`

2. Set

   Following methods are introduced to create collection of type Set:

   - `Set.of()`
   - `Set.of(s1)`
   - `Set.of(s1, s2)  // fixed-argument form overloads up to 10 elements`
   - `Set.of(elements…)  // varargs form supports an arbitrary number of elements or an array`

3. Map

   Following methods are introduced to create collection of type Map:

   - `Map.of()`
   - `Map.of(k1, v1)`
   - `Map.of(k1, v1, k2, v2) // fixed-argument form overloads up to 10 key-value pairs`
   - `Map.ofEntries(entry(k1, v1), entry(k2, v2),…) // varargs form supports an arbitrary number of Entry objects or an array`

So, the same list that we created earlier can be now created as:

```
List<String> strList=List.of("John", "Dave", "Vivian");
```

This approach looks more maintainable, flexible, and meaningful.

All these methods are self-explanatory and are used in our **Listings: 1-9**,**1-10** and **1-11**.

All the collections created by such static factory methods are by default unmodifiable. So, if you can try to add new elements as follows

```
strList.add("Emma");
```

you will get the runtime exception of "**UnSupportedOperationException**" as follows:

```
Exception in thread "main" java.lang.UnsupportedOperationException
  at
  java.base/java.util.ImmutableCollections.uoe(ImmutableCollections.
  at
  java.base/java.util.ImmutableCollections$AbstractImmutableList.adc
  at com.collection.UnmodifiableList.main(UnmodifiableList.java:35)
```

Along with making the process of initializing collection smoother, the use of such factory methods is also observed to be more lightweight in terms of memory occupation. As we do not use the new keyword while instantiating the collection type, it uses the heap memory only on an element basis. The overhead for the creation of different buckets for different objects and non-modifiable wrappers are reduced as the process is based on the algorithm of the field.



Sometimes, you would want to create an unmodifiable collection. Let us say in some situations where you need the List of resources or a Map of properties and values, you need an unmodifiable collection. Such collections are not lengthier, they will contain a few numbers of elements. To work with such collections, Java 9 has added static factory methods to collection interfaces:

- Use the static factory methods like of(), of(e), of(e,e,..), ofEntries(entry, entry) to initiate the default unmodifiable collections.
- These methods are used if the collection size is at the max of 10 elements. Although there is an overloaded method given in List, Set, and Map, you can add an arbitrary number of elements in the collections.

Updates in the Arrays class

Arrays is a utility class provided in Java. It provides many methods which perform some operations on arrays like converting them to lists, sorting, and so on. Java 9 provides some additional methods like the overloaded **`equals()`** method and the **`compareTo()`** method, and so on.

Problem

We want to use more flexible Arrays APIs which are introduced in Java 9 to

check the equality of arrays.

Solution

We can use different static methods of Arrays like **`equals()`**, **`compare(),`** and **`mismatch()`**. Create a new class **`ArraysDemo.java`** shown in **`Listing 1-12`**:

```
1. //Listing 1-12
2. public class ArraysDemo {
3.   public static void main(String[] args) {
4.    int[] data1 = {10,20,30,40,50};
5.          int[] data2 = {10,20,30,40,50};
6.          int[] data3 = {8,20,30,40};
7.
8.          checkArray(data1,data2,data3);
9.   }
10.
11.  public static void checkArray(int[] data1, int[] data2, int[]
data3) {
12.
13.   //Demonstration of equals()
14.    System.out.println("\nArrays.equals(data٣ ,٠ ,١, data,٠ ,٣
٣): " +
                Arrays.equals(data1, 0, 3, data3, 0, 3));
15.    System.out.println("Arrays.equals(data٣ ,٠ ,١, data,٠ ,٢
٣): " +
                Arrays.equals(data1, 0, 3, data2, 0, 3));
16.   //Demonstration of compare()
17.   System.out.println("data1 and data2 comparison := "+
18.                Arrays.compare(data1, data2));
19.   System.out.println("data1 and data3 comparison := "+
20.                Arrays.compare(data1, data3));
21.   //Demonstration of mismatch()
22.   System.out.println("data١ and data٢ comparison by mismatch:=
"+
```

```
                Arrays.mismatch(data1, data2));
    System.out.println("data1 and data2 comparison by mismatch:=
"+
                Arrays.mismatch(data1, data3));
  }
}
```

Output

We will get the output as shown in *Figure 1.8* if we execute **`ArrayDemo.java`**:

```
Arrays.equals(data1, 0, 3, data3, 0, 3): false
Arrays.equals(data1, 0, 3, data2, 0, 3): true
data1 and data2 comparison := 0
data1 and data3 comparison := 1
data1 and data2 comparison by mismatch:= -1
data1 and data3 comparison by mismatch:= 0
```

Figure 1.8: *Updates in array*

Explanation

Checking the equality of the arrays elements is one of the primary needs of an application. You can check the equality of the array by checking individual array elements or using the predefined `equals(array1, array2)` method provided by the Arrays class. The limitation of this method is you cannot check the partial array elements. In other words, if you want to slice the arrays and check those slices, it is not possible with this method.

Java 9 provides a special overloaded form of this **`equals()`** method, which is a static method of the **`Arrays`** class. This method facilitates the checking of array slices.

The syntax of this method is as follows:

```
public static boolean equals(int[] a, int index1,int index2,
            int[] b, int index3, int index4)
```

The method returns:

- True, if the slice of the array defined by `index1` to `index2` from array a is equal to the slice of the array defined by index 3 to index4 from array b.
- Both the arrays are considered to be equal if and only if:

- The slice is of an equal number of elements for both arrays.
- Every element and its order are the same in both the array slices.

The method also contains other overloaded forms to take an array of different types like **`double`**, **`boolean`**, **`float`**, and so on.

In our solution given by **`Listing 1-12`**, the equals method is invoked on different arrays. For data1 and data2, the equals method returns true, as both the array slices are the same. But for data1 and data3 arrays, the slices of arrays are different. So, the **`equals()`** method returns false.

Another method for the comparison of elements of an array is the **`compare()`** method. This method provides the lexicographic comparison of each element.

The syntax of the method is as follows:

```
public static int compare(int[] a, int[] b)
```

If both arrays are of the same length, then the lexicographic comparison is carried out by checking individual elements. If the length of both arrays is different, then the result is generated by comparing the length of the two arrays.

The method returns:

- 0, if both arrays are equal in length and contain identical elements in the same order.
- <0, if the first array is lexicographically lesser than the second array.
- >0, if the first array is lexicographically greater than the second array.



If an array element contains null, then lexicographically it is considered as less than a non-null element. However, if both the elements from array1 and array2 are null, then they are considered equal.

In our solution, the **`compare(data1, data2)`** returns 0. But the **`compare(data1, data3)`** returns 1, as the data1 arrays is greater than the data3 array.

The third method that we have used in our solution is **`mismatch()`**.

The syntax of the method is as follows:

```
public static int mismatch(int[] a, int[] b)
```

This method works in the reverse approach of **`equals()`** and **`compare()`**. As the name implies, it finds out the first mismatched element from the arrays specified.

The method returns:

- -1, if there is no mismatch found.
- Index of the first mismatch between two arrays, which also signifies the length of the array up to which the elements are identical.
- Length of the smaller array, if both the arrays are of not identical length.
- **`NullPointerException`**, if either of the arrays is null.

The API also provides another form as an overloaded method of **`mismatch()`**, which checks the mismatch for the slices of arrays.

The syntax is as follows:

```
public static int mismatch(int[] a, int index1, int index2,int[] b,
int index3, int index4)
```

It works similarly to the earlier method, but it provides additional arguments in the form of index numbers. The arguments **`index1`** and **`index2`** declare the starting and ending element index number (the ending index number is excluded) of the first array while **`index3`** and **`index4`** represent the same for the second array. Instead of checking the entire array element, it checks the slice of the array which is formed by these index numbers. The return types work exactly in the same manner as the earlier method.

In our solution in **`Listing 1-12`**, **`mismatch(data1, data2)`** returns -1, as there is no mismatch between both arrays. But **`mismatch(data1, data3)`** returns 0, which is the index number of the first mismatch element from both the arrays.



The method mismatch() is used in a situation where you want to achieve one of the following:

- Checking the length of the array, up to which both the arrays are equal.
- Checking whether there is any mismatch in the slice provided by array index numbers.

Problem

We want to use the modified methods of Arrays for our own objects. How to achieve that?

For example, we have some transactions stored in different arrays, and we want to check the equality of those array elements by any of the methods like **`equals()`**, **`compare()`**, or **`mismatch()`**.

Solution

We will use another overloaded **`mismatch()`** method to solve this problem statement.

To proceed, we need the following resources:

- The **`Transaction`** class with attributes of **`transactionId`** and **`transactionAmount`**.
- The **`Comparator`** implementor with the business logic of comparing the objects of transactions.
- Arrays of transaction to pass as an argument for the **`mismatch()`** function.

Follow the steps given below:

1. Create **`Transaction.java`** as shown in **`Listing 1-13`**:

```
1. //Listing 1-13
2. public class Transaction {
3.   private String transactionId;
4.   private double transactionAmount;
5.   public Transaction(String transactionId,
                        double transactionAmount) {
6.    super();
7.    this.transactionId = transactionId;
8.    this.transactionAmount = transactionAmount;
9.   }
10.  public String getTransactionId() {
11.   return transactionId;
12.  }
13.  public void setTransactionId(String transactionId) {
14.   this.transactionId = transactionId;
15.  }
16.  public double getTransactionAmount() {
17.   return transactionAmount;
18.  }
19.           public     void     setTransactionAmount(double
```

```
transactionAmount) {
20.    this.transactionAmount = transactionAmount;
21.   }
22. }
```

2. We need also the business logic to compare the transactions. This comparison is shown in **`TransactionComparator.java`** as shown in **`Listing 1-14`**:

```
1. //Listing 1-14
2. public class TransactionComparator implements
        Comparator<Transaction>{
3.   @Override
4.   public int compare(Transaction o1, Transaction o2) {
5.    // TODO Auto-generated method stub
6.                                                    return
o1.getTransactionId().compareTo(o2.getTransactionId());
7.   }
8.
9. }
```

3. Finally, we need the application class to create arrays and check their equality using mismatch. This is achieved by **`TransactionProcess.java`** as shown in **`Listing 1-15`**:

```
1. //Listing 1-15
2. package com.java9.arrays;
3.
4. public class TransactionProcess {
5.   public static void main(String[] args) {
6.    Transaction transaction١=new Transaction("txn١٠٠٠,"١);
7.    Transaction transaction٢=new Transaction("txn٢٠٠٠,"٢);
8.    Transaction transaction٣=new Transaction("txn٣٠٠٠,"٣);
9.    Transaction transaction٤=new Transaction("txn٤٠٠٠,"٤);
10.
11.    Transaction transaction٥=new Transaction("txn١٠٠٠,"١);
```

```
    Transaction transaction٦=new Transaction("txn٢····,"٢);
    Transaction transaction٧=new Transaction("txn٣····,"٥);
    Transaction transaction٨=new Transaction("txn٤····,"٤);

                    checkTransaction(new      Transaction[]
{transaction1,transaction2,
              transaction3,transaction4},new  Transaction[]
{transaction5,
       transaction6,transaction7,transaction8});
     }

     private  static  void  checkTransaction(Transaction[]
transactions1,
                                             Transaction[]
transactions2) {
    // TODO Auto-generated method stub
    int x=Arrays.mismatch(transactions1, transactions2,
                          new TransactionComparator());
    if(x==-١)
     System.out.println("Arrays are matched");
    System.out.println("First "+x+" elements are matched");
   }
}
```

Output

The output shown in *Figure 1.9* will be displayed if we execute **`TransactionProcess.java`**.

```
First 2 elements are matched
```

Figure 1.9: *Updates in array – using mismatch()*

Explanation

To solve this problem statement, we have used the overloaded form of the **`mismatch()`** method. As we know, the object comparison is achieved by the **`Comparable`** or **`Comparator`** interface. The default **`mismatch()`** method takes another argument of the type **`Comparator`**, which provides the business logic for comparison of the objects in an array.

The syntax of the method is as follows:

```
public static <T> int mismatch(T[] a, T[] b, Comparator<? Super T>
cmp)
```

The method returns:

- -1, if there is no mismatch found.
- Index of the first mismatch between two arrays, which also signifies the length of the array up to which the elements are identical. The comparator specified in the third argument is used to check the equality of array elements.
- Length of the smaller array, if both the arrays are of not identical length.
- **`NullPointerException`**, if either of the arrays is null.



- If you would like to compare two arrays, which are essentially of type objects, then you should use mismatch().
- This mismatch method takes three arguments. The first two arguments take arrays that you want to compare and the third argument is an object of the type Comparator which has the business logic for the comparison.

Improved optional interface

`The Optional<T>` class is being used to bypass the **`NullPointerException`** for a few years now. But still, it lacked some of the functionalities expected by developers. Java 9 provided three additional methods to improve the working of this class. The following are the different methods introduced for this interface:

- **`ifPresentOrElse()`**
- **`or()`**
- **`stream()`**

Problem

Though we have worked often on the **`Optional<T>`** class from Java 8, it does not provide the functionality about how to proceed further if the object is not found. Though there is no **`NullPointerException`**, there is no other way than to use the control structures like **`if… else`** to perform some action when the object is not retrieved. Does Java 9 provide any API for this?

Solution

Definitely! Java 9 provides methods like **`ifPresentOrElse()`**, **`or(),`** and **`stream()`** which provide extra features to the **`optional`** class.

Follow the steps given below:

1. Create **`Employee.java`** to create POJO as shown in **`Listing 1-16`**:

```
1. //Listing 1-16
2. package com.java9.optional;
3.
4. public class Employee {
5.   private String name;
6.   private Integer age;
7.   private Double salary;
8.   ..
9. }
```

2. Utilize this **`Employee`** in the **`Optional`** interface demonstration as shown in **`Listing 1-17`**:

```
1. //Listing 1-17
2. import java.util.Arrays;
3. import java.util.List;
4. import java.util.Optional;
5. import java.util.function.Supplier;
6. import java.util.stream.Collectors;
7.
8. public class Java9OptionalFeatures {
9.    static List<Employee> employeeList
```

```
            = Arrays.asList(new Employee("Bob Wiliams", 45,
15000.00),
           new Employee("Johny Hiscus", 45, 7000.00),
           new Employee("Alan Marsh", 65, 8000.00),
           new Employee("Jack Tavolta", 22, 10000.00),
           new Employee("Brayan Waugh", 29, 9000.00));

   public static void main(String[] args) {

    Optional<Employee> maxSalaryEmp =
             employeeList.stream()
               .filter(e->e.getSalary()>20000)
               .findFirst();

     /* Java 8 way to find the object conditionally
     if(maxSalaryEmp.isPresent())
      System.out.println(maxSalaryEmp.toString());
     System.out.println("Condition is not matched");
    */

     //working with ifPresentOrElse() in Optional
         System.out.print("Checking  availablity  of  object
using
                       ifPresentOrElse() ---> ");
     maxSalaryEmp.ifPresentOrElse((e)->
                          System.out.println(e.toString()),
()->
                          System.out.println(" Employee not
found"));

     //Creation of Supplier to supply the default object
     Supplier<Optional<Employee>> supplierEmployee = () ->
```

```
                Optional.of(new Employee("Tom Jones", 45, 7000.00));

     //working with or() in Optional
      System.out.println("Supplying the default object if object not
                          found --->");
     maxSalaryEmp=maxSalaryEmp.or(supplierEmployee);
     Employee e=maxSalaryEmp.get();
     System.out.println(e);

        //Creation of list consisting non-null and null objects
     List<Optional<Employee>> list = Arrays.asList(
            Optional.empty(),
                Optional.of(new Employee("Mark Root", 25, 7000.00)),
             Optional.of(new Employee("Daniel Martin", 35, 17000.00)));

    /*Java 8 way to create list of non-null objects
      using isPresent()
     List<Employee> filteredListJava٨ = list.stream()
        .flatMap(o -> o.isPresent() ?
                Stream.of(o.get()) : Stream.empty())
        .collect(Collectors.toList());
     */

     // Java 9 way to create list of nonnull objects using
    // Optional::stream.
      List<Employee> filteredListJava٩ = list.stream()
             .flatMap(Optional::stream)
             .collect(Collectors.toList());
```

```
64.
65.         System.out.print("List of non-null Employees --->");
66.         System.out.println(filteredListJava9);
67.     }
68. }
```

Output

If we execute **`Java9OptionalFeatures.java`**, we will get the output as shown in *Figure 1.10*:

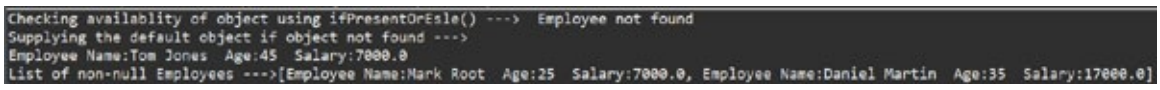


Figure 1.10: *Optional Interface*

Explanation

The solution provided above consists of the usage of **`ifPresentOrElse()`**, **`or()`,** and **`stream()`** methods that are introduced in Java 9. The **`Optional`** class controls the **`NullPointerException`** which might be thrown if no object is returned. So, to retrieve the object from **`Optional`,** we need to invoke the **`ifPresent()`** method as shown in line number 24 of **`Listing 1-17`**.

This conditional **`if … else`** block is always an extra burden, which we can avoid by using the **`ifPresentOrElse()`** method.

The syntax of this method is as follows:

```
public void ifPresentOrElse(Consumer<? super T> action1,
    Runnable action۲)
```

It works similarly to the **`if … else`** block internally. If the object or value is present, it performs the action1, which is connected to **`Consumer`**. If the object or value is not present, it performs action2 which is of the type **`Runnable`**.

If a value is present, it performs the given action with the value, otherwise it performs the given empty-based action.

On line number 32, we have used:

```
maxSalaryEmp.ifPresentOrElse((e)->System.out.println(e.toString()),
()->System.out.println(" Employee not found"));
```

In this case, we are trying to find an Employee whose salary is more than 20000. As this condition is not matched, the Optional returns null. But as we have used

ifPresentOrElse(), the program will generate the output as "**Employee not found**".

Another method introduced in **Optional** is **or()** which is used to supply the default object if the Optional does not contain any object.

The syntax of this method is as follows:

```
public Optional<T> or(Supplier<? extends Optional<? extends T>>
supplier)
```

This accepts a **Supplier**, which supplies an object of the type **Optional**.

The method returns:

- An **Optional,** which describes the value, if it is present.
- Or, **Optional** created by supplying a function.

If a value is present, it returns an Optional describing the value, otherwise it returns an **Optional** produced by the supplying function.

In our solution at line number 37, we have first created the **Supplier** as follows:

```
Supplier<Optional<Employee>> supplierEmployee =  () ->
Optional.of(new Employee("Tom Jones", 45, 7000.00));
```

We should keep in mind that the **Supplier** should of the type **Optional<T>**. Here, T is the type which is going to be produced by **Optional**.

Further at line number 43, we have used this supplier as an argument passed to the **or()** method as,

```
maxSalaryEmp=maxSalaryEmp.or(supplierEmployee);
```

Here, as the condition for finding an **Employee** does not match, **Optional<Employee>** returns null. So, we can control this by invoking **or()** on the generated **Optional**. This acts as a default object in the scenario when the **Optional** does not have any object associated with it.

The third method in the Optional is the **stream()** method. As the name indicates, this generates the stream from **Optional**, which we can iterate on.

The syntax for the method is as follows:

```
public Stream<T> stream()
```

The method returns:

- Sequential Stream, containing values, if present.
- Or, empty Streams.

This method has a special behavior because of which it ignores the empty

`Optional` elements and generates the list only for the values that are available. In our solution, we have created the list of **`Optional<Employee>`** as follows:

```
List<Optional<Employee>> list = Arrays.asList(
            Optional.empty(),
            Optional.of(new Employee("Mark Root", 25, 7000.00)),
            Optional.of(new Employee("Daniel Martin", 35,
17000.00)));
```

As we can observe, this list contains one of the elements as **`Optional.empty()`**. We do not want this element to be part of our iteration, so we have applied the **`stream()`** function on line number 61 as follows:

```
List<Employee> filteredListJava9 = list.stream(
                    .flatMap(Optional::stream)
                .collect(Collectors.toList());
```



Java 9 provides different methods to enhance the working of Optional as:

- ifPresentOrElse()

  This method is used to provide the alternate business logic if the Optional does not contain any value or object.

- or()

  The or() method takes the Supplier object, which is provided to Optional as a default object, if the Optional does not contain any value or object.

- stream()

  This method is used to create a list of elements from List of Optional<T> ignoring the empty elements of Optional.

Problem

Most of the programming languages provide **Read-Eval-Print-Loop** (**REPL**) to explore the programming skills or sometimes to learn the new feature. Writing the classes, creating **`main(),`** and then invoking objects and its methods is really time consuming. Is there any such feature available in Java?

Solution

To achieve this, Java 9 has launched a new feature "`JShell`". Using this, you can just type your code snippet and check the outputs. There is no need for any ceremonial code like the **`main()`** function or curly braces to define the start and end of a program. You can, in fact, type your code in this **`jshell`**, test it, and if it works, then you can copy-paste it in your actual class.

To use `jshell`, you can just go to the bin folder of your Java 9, and type **`jshell`**. This will instantiate and display the **`jshell`** editor for you as shown in *Figure 1.11*:

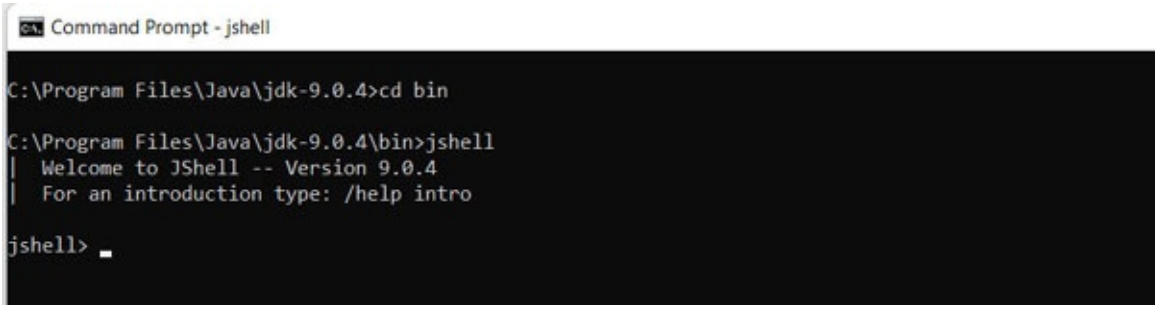


Figure 1.11: *JShell - I*

To exit from the **`jshell`** and return to the normal console, we can give the **`/exit`** command as shown in *Figure 1.12*:

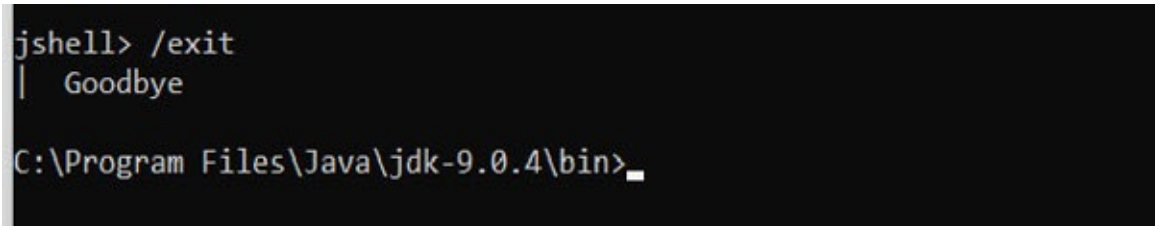


Figure 1.12: *JShell - II*

Explanation

As we now know how to open **`jshell`**, we can now see how to write different code snippets to test the outputs in **`jshell`**.



The code snippet in jshell is denoted in bold letters and the output is displayed in the normal letters.

Case 1: Printing "Hello world"

```
jshell>System.out.println("Hello world!!");
Hello world!!
```



A semicolon (;) for termination of line is optional. It is added by jshell automatically.

Case 2: Creating a variable

```
jshell>String s="Java"
s ==> Java
```

In this case, **jshell** creates and initializes the variable **s** and displays the current value of it. This variable can be also used in the latter operations performed in the same **jshell** instance.

Case 3: Creating and invoking functions

```
jshell>int findMax(int x,int y){
   …> return(x>y?x:y);
   …> }
created method findMax(int,int)
```

In this case, the function **findMax(int x, int y)** is created. Usually, **jshell** completes the command after you hit the "*Enter*" key. But while writing a function, when you type the opening curly braces, **jshell** waits till you close the same to know that the function body or command is completed.

This function can be invoked as follows:

```
jshell>findMax(10,20)
$1 ==> 20
```

$1 is a temporary variable created to store the value returned from a function which can be used later. This temporary number is a sequence number of command, which stores the value generated from that command.

Case 4: Creating a forward reference

JShell works differently than your normal Java code. In Java, you cannot refer to anything which is futuristic. The variable or method that you are using must be declared earlier. That is not the case with **jshell**. You can create functions in **jshell**, and refer to "*something*" which can be created at a later stage. Of course, you will not be able to invoke that function, unless that "*something*" is created.

```
jshell>double calculatePrice(int rate, int qty){
   …> return (rate*qty)+ (rate*TAX)/100;
   …> }
```

The created method **calculatePrice(int,int)**, however, cannot be invoked until the variable TAX is declared:

```
jshell>int TAX=15;
TAX ==> 15
jshell>calculatePrice(1200,3);
$16 ==> 3780.0
```



Jshell is an interactive environment which can be used to test the code snippet. You can test the logical expressions, or even function definitions, and so on, using this. You need not have to write the complete class or create an object for generating the output.

ObjectInputFilter interface

Serialization is one of the major tasks which developers might need to do when they need to share the object in the network. But while doing so, there is always a threat that the object may get attacked by some external intruders. Java 9 introduced **`ObjectInputFilter`** interface to filter the object before it is deserialized and used in an application.

Problem

De-serialization facilitates retrieving the serialized object from the local system/network. But sometimes, the object retrieved may have some vulnerabilities or it may be insecure. How do I make sure that the de-serialized object that is being retrieved is secure, non-vulnerable?

Solution

This can be achieved by filtering the deserialized object by using the **`ObjectInputFilter`** interface which is introduced in Java 9. The following are the steps to filter the serialized object before de-serialization:

1. Create **`Employee.java`**, a POJO class as shown in **`Listing 1-18`**.
2. Serialize the object of Employee by using the **`Serializable`** interface. The process is shown is **`Listing 1-19`**, **`SerializeEmployee.java`**.
3. Create **`FilterEmployee.java`** as shown in **`Listing 1-20`**. This class implements **`ObjectInputFilter`** and overrides the **`checkInput()`** method as shown:

```
1. //Listing 1-20
2. package com.java9.serialization;
3.
4. import java.io.ObjectInputFilter;
5.
```

```
class FilterEmployee implements ObjectInputFilter{
     public Status checkInput(FilterInfo filterInfo) {
        //Generating the class type which object is being serialized
         Class<?> serialClass = filterInfo.serialClass();
         if (serialClass != null) {
                                                    return (Employee.class.isAssignableFrom(serialClass)) ?
                        Status.ALLOWED : Status.REJECTED;
         }
         else {
          System.out.println("NULL");
          return Status.UNDECIDED;
         }
     }
}
```

4. Create **`FilterEmployeeByPackage.java`** shown in **`Listing 1-21`** as follows. This provides a method, which returns the **`ObjectInputFilter.Status`** based on the business logic. In the following code, we are checking whether the serialized object is from a specific package or not:

```
//Listing 1-21
package com.java9.serialization;

import java.io.ObjectInputFilter;

public class FilterEmployeeByPackage {
   static ObjectInputFilter.Status
      empFilter(ObjectInputFilter.FilterInfo filterInfo) {
        //Generating the class type which object is being serialized
         Class<?> serialClass = filterInfo.serialClass();
         if (serialClass != null) {
```

```
            //checking if the deserialized object is part of specific     //package
                                                    return serialClass.getPackageName().equals("java.util")
                        ? ObjectInputFilter.Status.ALLOWED
                        : ObjectInputFilter.Status.REJECTED;
            }
            return ObjectInputFilter.Status.UNDECIDED;
        }
}
```

5. Finally create the application class **`FilteringDeserialization.java`** as shown in **`Listing 1-22`**:

```
//Listing 1-22
package com.java9.serialization;

import java.io.FileInputStream;
import java.io.IOException;
import java.io.ObjectInputStream;

public class FilteringDeserialization {
        public static void main(String[] args) throws IOException {
     filteringByClass();
     filteringByMethod();
    }

   // filtering the de-serialization by invoking the filter class
        private static void filteringByClass() throws IOException {
     // TODO Auto-generated method stub
     FileInputStream is = new FileInputStream("emp.dat");
```

```
18.    try (ObjectInputStream ois = new ObjectInputStream(is))
    {
19.     ois.setObjectInputFilter(new FilterEmployee());
20.     Employee employee = (Employee) ois.readObject();
21.        System.out.println("Employee object after applying
    filter:"+›\n›+
           employee);
22.    } catch (ClassNotFoundException ex) {
23.     System.out.println("Cannot deserialize");
24.    }
25.   }
26.
27.   // filtering the de-serialization by invoking the method
28.       private static void filteringByMethod() throws
    IOException {
29.    // TODO Auto-generated method stub
30.    FileInputStream is = new FileInputStream("emp.dat");
31.    try (ObjectInputStream ois = new ObjectInputStream(is))
    {
32.     ois.setObjectInputFilter(FilterEmployeeByPackage::empFi
33.     Employee employee = (Employee) ois.readObject();
34.        System.out.println("Employee object after applying
    filter:"+›\n›+
                                              employee);
35.     System.out.println(employee);
36.    } catch (ClassNotFoundException ex) {
37.     System.out.println("Cannot deserialize");
38.    }
39.   }
40. }
```

Output

If we execute **`FilteringDeserialization.java`**, we will get the output as

shown in *Figure 1.13*:

```
Employee object after applying filter:
Employee [empId=101, empName=William Smith]
Exception in thread "main" java.io.InvalidClassException: filter status: REJECTED
```

Figure 1.13: *Using ObjectInputFilter*

Explanation

`ObjectInputFilter` is a functional interface introduced in Java 9. It is used as a filter which can be applied to classes, array lengths, and graph metrics that are obtained from the deserialization process. To achieve this, the interface provides a method **`checkInput()`** method.

The syntax of the method is as follows:

```
ObjectInputFilter.Status checkInput(ObjectInputFilter.FilterInfo
filterInfo)
```

The method takes a parameter of **`FilterInfo,`** which provides the information about the current object which is being serialized.

The method returns:

- **`Status.ALLOWED`**, if the object is accepted.
- **`Status.REJECTED`**, if the object is rejected.
- **`Status.UNDECIDED`**, if no decision of done to accept or reject.

Where `Status` is the static enum present in **`ObjectInputFilter`** with constants as, **`ALLOWED`**, **`REJECTED,`** and **`UNDECIDED`**.

The implementation of this method is passed as a lambda expression to the **`setObjectInputFilter()`** method of **`ObjectInputStream`**.

In our solution at line number 19 of **`Listing 1-22`**, we have used **`checkInput()`** as follows:

```
ois.setObjectInputFilter(new FilterEmployee());
```

Where, **`ois`** is the **`ObjectInputStream`** reference. The **`FilterEmployee.java`** is essentially of the type **`ObjectInputFilter`** which overrides the **`checkInput()`** method as follows:

```
class FilterEmployee implements ObjectInputFilter{
  public Status checkInput(FilterInfo filterInfo) {
    //business logic for filtering
  }
}
```

Another approach to implementing the **`ObjectInputFilter`** for deserialization is passing the static method reference in the **`setObjectInputFilter()`** method, which should return the **`Status`**.

In our solution, we have used **`FilterEmployeeByPackage.java`** in **`Listing 1-21`**. It provides a method **`empFilter()`** on line number 6, which takes **`ObjectInputFilter.FilterInfo`** as an argument, which provides information about the object which is supposed to filter.

We have passed this reference of **`empFilter()`** to our **`ObjectInputStream`** on line number 32 on **`Listing 1-22`** as follows:

```
ois.setObjectInputFilter(FilterEmployee::empFilter);
```

This lambda invokes the business logic of the filter defined in the method **`empFilter()`**.



ObjectInputFilter facilitates the filtering of a deserialized object. This makes the application secure as it controls the deserialization of unwanted and vulnerable objects. This can be achieved with the following resources:

- checkInput(FilterInfo filterInfo)

  Method from ObjectInputFilter which takes the FilterInfo reference that holds the information about the object to be serialized. This returns the status based on the filter condition.

- setObjectInputFilter(ObjectInputFilter filter)

  Method from ObjectInputStream, which passes the reference of ObjectInputFilter or the lambda of the method which returns Status.

- Status

  This is enum from ObjectInputFilter which returns the status out of the filter as ALLOWED, REJECTED, and UNDECIDED.

Flow API

The Flow API is a completely new API launched by Java 9. This is Java's way of reactive programming. Java 9 introduced different classes like **`Subscriber`**, **`Publisher`**, **`Submission`**, **`SubmissionPublisher`**, and so on, to achieve this publish-subscribe model.

Problem

How to use the Flow API which is introduced in Java9?

Solution

To understand this, we will solve a real-time problem. There is a book publishing organization which publishes books. The publishing company have their own members which we can call as subscribers. But subscribers are not aware that when the new books are published. This limitation can be resolved with the help of Flow API with the following steps:

1. Create a Java bean to store book details like **`bookId`** and **`bookName`** as shown in **`Listing 1-23 Book.java:`**

```
1. //Listing 1-23
2. public class Book {
3.   private int bookId;
4.   private String bookName;
5.
6.   public int getBookId() {
7.    return bookId;
8.   }
9.   public void setBookId(int bookId) {
10.   this.bookId = bookId;
11.  }
12.  public String getBookName() {
13.   return bookName;
14.  }
15.  public void setBookName(String bookName) {
16.   this.bookName = bookName;
17.  }
18.  public Book(int bookId, String bookName) {
19.   super();
20.   this.bookId = bookId;
21.   this.bookName = bookName;
22.  }
```

```
  @Override
  public String toString() {
     return "Book [bookId=" + bookId + ", bookName=" + bookName + "]";
  }

}
```

2. Create a Subscriber which will fetch the books when it is published by a publisher. This is implemented in **`Listing 1-24`**, **`BookSubsriber.java`**:

```
//Listing 1-24
import java.util.concurrent.Flow;
import java.util.concurrent.Flow.Subscription;

public class BookSubscriber implements Flow.Subscriber<Book>{
  private String subscriberName;
  private Flow.Subscription subscription;

  public BookSubscriber(String subscriberName) {
   this.subscriberName=subscriberName;
  }

  @Override
  public void onSubscribe(Subscription subscription) {
   // TODO Auto-generated method stub
   this.subscription=subscription;
   subscription.request(1);
  }

  @Override
  public void onNext(Book book) {
```

```
    // TODO Auto-generated method stub
    subscription.request(1);
                        System.out.println(book+",Received by:"+this.subscriberName);
  }

  @Override
  public void onError(Throwable throwable) {
    // TODO Auto-generated method stub
    System.out.println(throwable.getMessage());
  }

  @Override
  public void onComplete() {
    // TODO Auto-generated method stub
       System.out.println(subscriberName +" got all the books");
  }

}
```

3. Create a **`BookPublisher.java`** to subscribe the book subscribers, as shown in **`Listing 1-25`**:

```
//Listing 1-25
import java.util.Arrays;
import java.util.List;
import java.util.concurrent.SubmissionPublisher;
import java.util.concurrent.TimeUnit;

public class BookPublisher {
  public static void main(String[] args) {
    List<Book> bookList = Arrays.asList(
```

```
      new Book[] {
        new Book(101, "Java 9 to 13"),
        new Book(102, "Java 11")
        });

    SubmissionPublisher<Book> bookPublisher =
      new SubmissionPublisher<Book>();

                BookSubscriber subscriber1 = new BookSubscriber("Dave");
                BookSubscriber subscriber2 = new BookSubscriber("Richardson");
    bookPublisher.subscribe(subscriber1);
    bookPublisher.subscribe(subscriber2);

    // submit() method:
    System.out.println("Using submit() method :==>");
    bookList.stream().forEach(i -> {
     bookPublisher.submit(i);
     try {
      Thread.sleep(3000);
     } catch (InterruptedException e) {
      // TODO Auto-generated catch block
      e.printStackTrace();
     }
    });

    // offer() method : simple form
    System.out.println("Using offer() :==>");
    bookList.stream().forEach(book -> {
     bookPublisher.offer(book, null);
     try {
```

```
            Thread.sleep(3000);
          } catch (InterruptedException e) {
            // TODO Auto-generated catch block
            e.printStackTrace();
          }
        });

        // offer() method : timeout implementation
        System.out.println("Using offer() with timeout :==>");
        bookList.stream().forEach((book) -> {
          System.out.println(book + " being offered");
        final int result١ =
         bookPublisher.offer(book, ٢, TimeUnit.SECONDS,
                (subscriber, value) -> {
                  try {
                    Thread.sleep(3000);
                  } catch (InterruptedException e) {
                  // TODO Auto-generated catch block
                  e.printStackTrace();
                  }
                return true;
              });
          if (result٢ > ١)
            System.out.println("Dropping " + result1 + " book.");
        });

      }
}
```

Output

If we execute **`BookPublisher.java`**, we will get the output as shown in *Figure 1.14*:

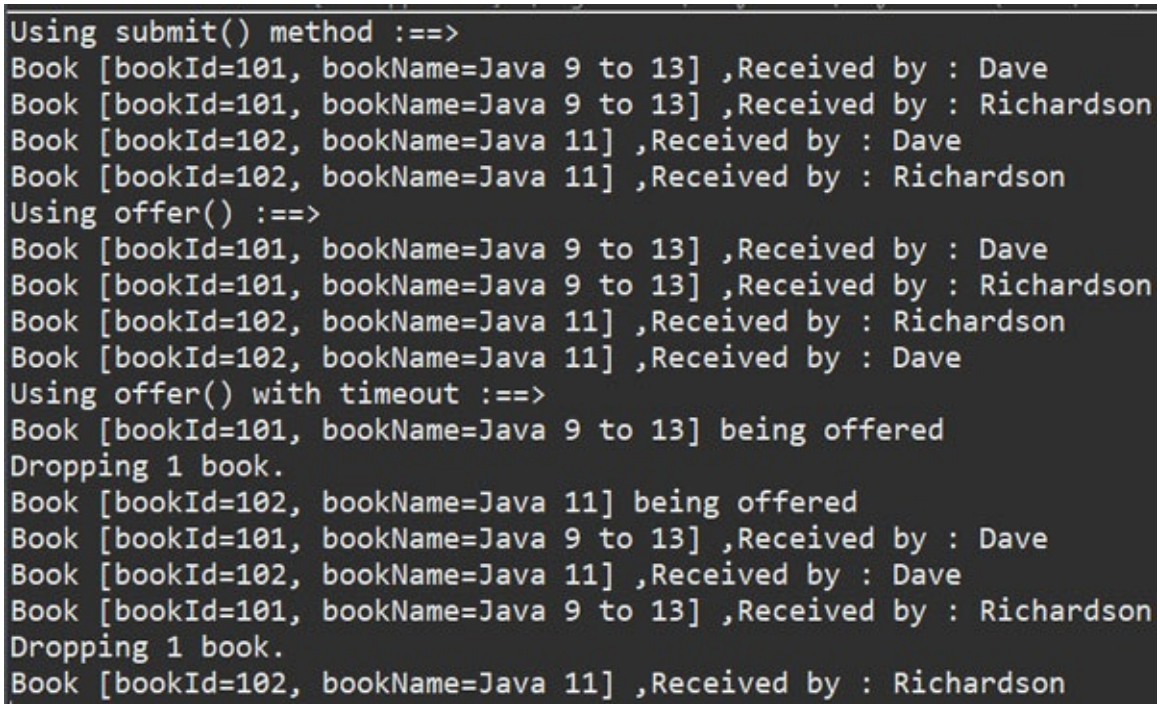


Figure 1.14: *Implementing the Flow API*

Explanation

Java 9 provides a new API termed as the Flow API, which is Java's implementation of Reactive Stream Specification. The API is internally dependent on the Iterator and Observable pattern. The iterator normally pulls the item from the source and pushes it to the subscriber. This pushing of items is observed by the observer.

This implementation is achieved by the Flow API. The topmost class for this API is the **`java.util.concurrent.Flow`** class. This class provides a combination of static interfaces which provides the basic functionalities for Publisher-Subscriber behaviors.

The following are the different interfaces declared within this class:

public static interface Flow.Subscriber<T>

This interface facilitates subscribing to the publisher for callbacks. This pushing of data items is only possible upon request. Subscribers need to make a request, to get the data items, that are pushed from the Publisher. Such invocations are always ordered.

The interface declares the following methods:

- onSubscribe(Subscription subscription)

Before invoking any other method for a given subscription, this method is invoked. There is a possibility that this method throws exceptions. If that happens, the result is indeterministic. But the subscription may get cancelled.

In our solution, we have implemented this method in the **BookSubcriber** class at line number 14. **BookSubscriber** implements the **Subscriber**, as a result of which it implements methods of the Subscriber. Observe the following snippet where **BookSubscriber.java** has implemented **onSubscribe()**:

```
@Override
public void onSubscribe(Subscription subscription) {
  // TODO Auto-generated method stub
  this.subscription=subscription;
  subscription.request(1);
}
```

In this method, the **BookSubscriber** initiates the **Subscription** object when it is subscribed by **Publisher**. Also, the **request()** method is invoked to fetch the items which are published by **Publisher**.

- onNext(T item)

  This method is invoked with the next item of **Subscription**. You need to request the item which is pushed by the Publisher by using the **request()** method.

  In our solution, we have implemented this method in **BookSubscriber.java** as shown:

```
@Override
public void onNext(Book book) {
  // TODO Auto-generated method stub
  subscription.request(1);
  System.out.println(book +", Received by :
"+this.subscriberName);
}
```

  This method fetches the Book item from the Publisher. Again, the **request()** method here is used to request the next item. The integer number provided in the **request()** must be greater than zero.

- onError(Throwable throwable)

  This method is implicitly invoked if any error occurs in **Publisher** or **Subscriber**. After the invocation of this method, the entire subscription is cancelled and no further process is carried out.

  In our code, we do not have this feature at the moment. But if we invoke:

```
subscription.request(0);
```

In the **`onNext()`** method, it will raise **`IllegalArgumentException`**. This will internally invoke the **`onError()`** method.

- onComplete()

  When the entire data is pushed by the Publisher and processed by the **`Subscriber`** without any error, then this method is invoked. This is the natural termination of the Publisher-**`Subscriber`** architecture.

public static interface Publisher<T>

This is a functional interface, which is used for generating the items, which can be utilized or subscribed by the **`Subscriber`**. The items produced by the **`Publisher`** are utilized by the **`Subscriber`** by invoking the **`onNext()`** method. If while producing the items or message, the Publisher raises any exception, then the Subscriber receives **`onError(),`** and then the process is terminated.

The only method this interface provides is as follows:

- subscribe(Subscriber<? super T> subscriber)

  This method adds the Subscriber for the Publisher. You cannot subscribe to the same Subscriber again due to policy violation.

public static interface Subscription

This interface provides the services by which the link is established between the **`Publisher`** and **`Subscriber`**. It provides only the methods by which the **`Subscriber`** can request the item and disconnect the **`Publisher`**.

It declares following methods:

- request(long n)

  This method adds the given number of items from the **`Publisher`** to the **`Subscriber`** to consume. This number must be greater than 0, else this will generate **`onError()`** with **`IllegalArgumentException`**.

- cancel()

  This method cancels the subscription established with the **`Publisher`** by the **`Subscriber`**. This invocation does not follow either the **`onError()`** or **`onComplete()`** method.

  For example, if we want to cancel the subscription, we can add the following code to our existing **`onNext()`** method:

```
@Override
public void onNext(Book book) {
```

```
        // TODO Auto-generated method stub
        …..
        …..
        if(book.getBookName().equals("Java 17"))  {
       System.out.println(subscriberName+" cancelling
       subscription");
       subscription.cancel();
        }
     }
```

Along with all these interfaces, we need to also talk about the class, **`SubmissionPublisher,`** a new concrete implementation of the **`Publisher<T>`** interface which is declared as follows:

```
public class SubmissionPublisher<T> extends Object
             implements Flow.Publisher<T>, AutoCloseable
```

This object uses the Executor which is provided in one of the constructors for delivery to subscribers. The type of Executor (**`newFixedThreadPool`**, **`CachedThreadPool`**, **`ScheduledExecutor`**) to be provided depends on the usage for which the **`Publisher`** is going to be utilized.

It has the following overloaded constructors:

- SubmissionPublisher()

  This creates a default **`SubmissionPublisher`**. As the **`Executor`** is not provided in its argument, it uses the default **`Executor ForkJoinPool.commonPool()`** which manages the async delivery to subscribers. The exceptions from **`onNext()`** are not handled. It uses the default buffer capacity to perform the publishing.

- SubmissionPublisher(Executor executor, int maxBufferCapacity)

  Creates a **`SubmissionPublisher`** using the given **`Executor`** in the first argument. The second argument denotes the maximum buffer capacity for each subscriber. The exceptions from **`onNext()`** are not handled.

- **`SubmissionPublisher(Executor executor, int maxBufferCapacity, BiConsumer<?super Flow.Subscriber<?super T>,? super Throwable> handler)`**

  Creates a **`SubmissionPublisher`** using the given Executor in the first argument. The second argument denotes a maximum buffer capacity for each subscriber. The third argument provides the handler for the **`onNext()`** method.

In our solution, we have initiated the object of **`SubmissionPublisher`** in the

class **BookPublisher.java**. Also, we have subscribed to two objects of **BookSubscriber** to this publisher as follows:

```
bookPublisher.subscribe(subscriber1);
bookPublisher.subscribe(subscriber2);
```

The SubmissionPublisher class provides the following important methods:

- submit(T item)

  This method is used to publish the given item to the current subscribe. It invokes the **onNext()** method asynchronously. This blocks uninterruptedly while resources for any subscriber are not available.

  In **BookPublisher.java,** we have invoked this method as follows:

```
bookList.stream().forEach(i -> {
bookPublisher.submit(i);
   try {
    Thread.sleep(3000);
   } catch (InterruptedException e) {
    // TODO Auto-generated catch block
    e.printStackTrace();
   }
  });
```

  It is observed that the **submit()** method submits each item to the subscriber with the help of **forEach()**.

- offer(T item, BiPredicate<Flow.Subscriber<?super T>, ?super T)onDrop)

  This method is used to publish the given item to the current subscribe. It invokes the **onNext()** method asynchronously.

  The **offer()** method takes two forms. The simple form **offer()** method only offers the item and does not have any **BiPredicate** which is implemented in our solution as follows:

```
// offer() method : simple form
System.out.println("Using offer() :==>");
bookList.stream().forEach(book -> {
   bookPublisher.offer(book, null);
   try {
    Thread.sleep(3000);
   } catch (InterruptedException e) {
    // TODO Auto-generated catch block
    e.printStackTrace();
   }
});
```

  Here, the **offer()** method only offers the item. There is no condition or logic based on which the items will be dropped for the subscriber.

The other form of **`offer()`** takes the additional argument of **`BiPredicate`** which is essentially a combination of the subscriber and the item type. In our solution, we have achieved this in the following code snippet:

```
// offer() method : timeout implementation
   System.out.println("Using offer() with timeout :==>");
   bookList.stream().forEach((book) -> {
     System.out.println(book + " being offered");
     final int result1 = bookPublisher.offer(book, 2,
           TimeUnit.SECONDS, (subscriber, value) -> {
     try {
         Thread.sleep(3000);
      } catch (InterruptedException e) {
         // TODO Auto-generated catch block
         e.printStackTrace();
      }
       return true;
    });
```

In the preceding code snippet, we are instructing the **`SubmissionPublisher`** to drop the items if the items are not offered in 2 seconds. This is achieved by the second and third argument of the offer method. This returns a true that interprets that we do not want to raise any exception after the items are dropped, but we will have some business logic to implement if the items are dropped.

In our solution, we have set up the sleep time as 3 seconds. As this is more than 2 seconds, the items will be dropped and the result1 will contain a number of items delivered back to the subscriber.



As this whole process is internally managed by Executors (and ultimately by threads), the output is not guaranteed.

- **consume(Consumer <? super T> consumer)**

  This method processes all published items. This processing is achieved by the Consumer provided in argument.

Updates in CompletableFuture

`CompletetableFuture` was introduced in JDK 8.0. This API is updated in Java 9 with some new features.

Problem

How can we leverage the new features introduced in the **`CompletableFuture`** API ?

Solution

There are many updates in the **`CompletableFuture`** API incorporated in version 9. In the following solution, we will implement a few of them like:

- completedOnTime()
- orTimeOut()
- delayedExecutor()

Create the class **`CompletableFutureDemos.java`** as shown in **`Listing 1-26`**:

```
1. //Listing 1-26
2. import java.util.concurrent.CompletableFuture;
3. import java.util.concurrent.TimeUnit;
4.
5. public class CompletableFutureDemos {
6.
7.   public static void main(String[] args) {
8.    // TODO Auto-generated method stub
9.    CompleteOnTime();
10.   orTimeOut();
11.   delayedExecutor();
12.
13.  }
14.
15.  //implementation of completeOnTime()
16.  private static void completeOnTime() {
17.   // TODO Auto-generated method stub
18.   int value١٠٠ = ١;
19.   int value٢٠٠ = ٢;
```

```
20.
21.    CompletableFuture.supplyAsync(() -> {
22.     try {
23.      TimeUnit.SECONDS.sleep(5);
24.     } catch (InterruptedException e) {
25.      e.printStackTrace();
26.     }
27.     return value١ + value٢;
28. }).completeOnTimeout(10,2,TimeUnit.SECONDS).thenAccept(result->
          System.out.println("Result from completeOnTime()==>
    "+result));
29.     //waits for 2 seconds to complete, else returns 10 as a
    value
30.    try {
31.     TimeUnit.SECONDS.sleep(10);//waiting for 10 seconds
32.    } catch (InterruptedException e) {
33.     // TODO Auto-generated catch block
34.     e.printStackTrace();
35.    }
36.   }
37.
38. //implementaion of orTimeOut()
39.   //sets a time limit of 1 seconds else returns exception
40.   private static void orTimeOut() {
41.    // TODO Auto-generated method stub
42.    int value١٠٠ = ١;
43.    int value٢٠٠ = ٢;
44.    CompletableFuture.supplyAsync(() -> {
45.     try {
46.      TimeUnit.SECONDS.sleep(3);
47.     } catch (InterruptedException e) {
```

```
      e.printStackTrace();
     }
     return value١ + value٢;})
     .orTimeout(1, TimeUnit.SECONDS).whenComplete(
                                        (result, exception)  -> {
     System.out.println("Result from orTimeOut()==> "+result);
     if (exception != null) {
      exception.printStackTrace();
      System.out.println("Job not completed on Time");
     }
    });
    try {
     TimeUnit.SECONDS.sleep(10); //waiting for 10 seconds
    } catch (InterruptedException e) {
     // TODO Auto-generated catch block
     e.printStackTrace();
    }
   }
   //implementation of delayedExecutor()
   private static void delayedExecutor() {
    // TODO Auto-generated method stub
    int value١٠٠ = ١;
    int value٢٠٠ = ٢;
    CompletableFuture.supplyAsync(() -> value1 + value2,
      CompletableFuture.delayedExecutor(2, TimeUnit.SECONDS))
      .thenAccept(result -> System.out.println("Result from
                             delayedExecutor()==> "+result));
    try {
     TimeUnit.SECONDS.sleep(10);
    } catch (InterruptedException e) {
     // TODO Auto-generated catch block
```

```
77.      e.printStackTrace();
78.     }
79.    }
80. }
```

Output

If we execute **`CompletableFutureDemos.java`**, we will get the output as shown in *Figure 1.15*:

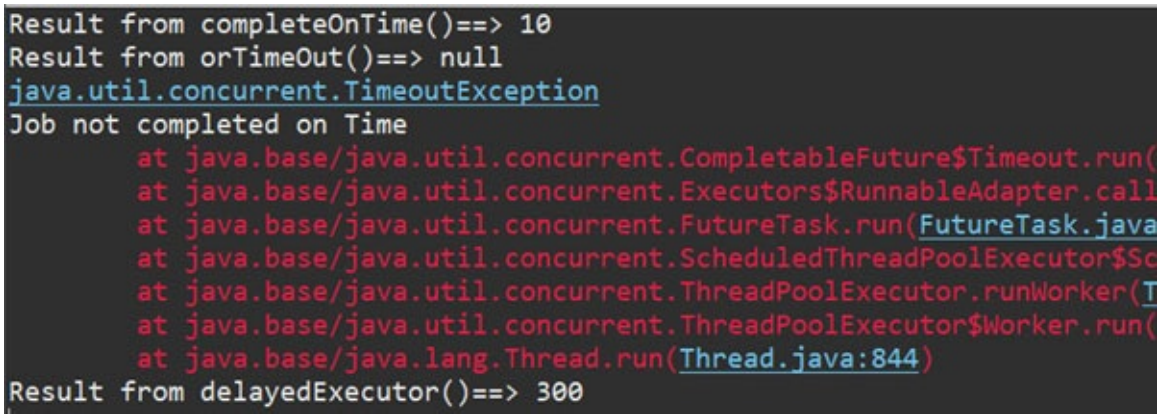


Figure 1.15: *Implementation of the CompletableFuture API*

Explanation

`CompletableFuture<T>` was introduced in Java 8. This API became popular immediately after its release. But developers found that it lacked in some of the functionalities like how to handle the situation if the task is not completed in the given time. To achieve this and some other functionalities, Java 9 added a few more methods to **`CompletableFuture`**.

The following methods are added:

- public <U> **`CompletableFuture`**<U> newIncompleteFuture()

  This method returns a new incomplete **`CompletableFuture`**. This is also known as "*virtual constructor*" of **`CompletableFuture`**. This method is normally overridden in the classes which are subtypes of **`CompletableFuture`**.

  The method returns new **`CompletableFuture`**.

- public **`Executor`** defaultExecutor()

  This method is used to return the default **`Executor`** which is used for async

methods. When developers do not want to specify an **Executor,** this method is used to get the default **Executor**. Internally, it uses **ForkJoinPool.commonPool()** or a simple **Executor** which uses one thread per async task.

The method returns an **Executor.**

- public **CompletableFuture**<**T**> copy()

  As the name indicates, this method is used to create a copy of **CompletableFuture**. If the current **CompletableFuture** completes without any exception, then it returns the copy of **CompletableFuture** with the same value. If the current **CompletableFuture** generates an exception while completion, then the new **CompletableFuture** also completes with **CompletionException**.

  The method returns copy of **CompletableFuture**.

- public **CompletionStage**<**T**> minimalCompletionStage()

  This method is used to return the **CompletionStage** which is completed normally. It has the same value of the **CompletableFuture** when it completes normally. If **CompletableFuture** completes with an exception, then this method returns the completed **CompletableStage** with **CompletionException**.

  The method returns new **CompletionStage**

- **public CompletableFuture<T> completeAsync(Supplier<? extends T> supplier)**

  This method is used to complete the **CompletableFuture** asynchronously with the value or result supplied by the Supplier, which is provided as an argument. It uses the default executor to execute this task.

  The method returns **CompletableFuture**.

- **public CompletableFuture<T> completeAsync(Supplier<? extends T> supplier,Executor** executor)

  This method is used to complete the **CompletableFuture** asynchronously with the value or result supplied by the **Supplier**, which is provided as an argument. It uses the executor provided as a second argument to execute the task.

  The method returns **CompletableFuture**.

- **public CompletableFuture<T> orTimeout(long timeout, TimeUnit unit)**

This method completes **CompletableFuture** exceptionally. If, it is not completed in the time provided in the first argument, it is completed with **TimeoutException**.

The method returns **CompletableFuture.**

Observe the following snippet, which we have used in our solution at line number 53 of **Listing 1-26**:

```
orTimeout(1, TimeUnit.SECONDS).whenComplete((result,
                                  exception) -> {
  //business logic to process the result or handle the
exception
}
```

Here, we are setting up the time out period as 1 second. The second argument in or **Time()** defines which unit you are using for the first argument. We are invoking **whenComplete()** which takes **BiConsumer** to consume the values. If the task is completed in 1 second, then the result will be generated, else an exception will be raised. In line number 62, as we are applying **TimeUnit.SECONDS.sleep(10)** the task doesn't complete in 1 second. So, we get the result as null and an exception will be raised which we can handle:

- **public CompletableFuture<T> completeOnTimeout(T value,long timeout, TimeUnit unit)**

  This method takes three arguments. If the **CompletableFuture** does not complete in the given timeout period, provided in the second argument, then it completes with the value provided in the first argument.

  The method returns **CompletableFuture**.

  Observe the following snippet from our solution, we have used **completeOnTimeout()** at line number 28 in **Listing 1-26** as follows:

```
completeOnTimeout(10, 2, TimeUnit.SECONDS).thenAccept(result -
>
            System.out.println("Result from
completeOnTime()==> "+result));
```

  Here, we have invoked **completeOnTimeOut()** with three arguments. The first argument provides the default result which will be generated, if the task is not completed as per the timeline in seconds which is provided in the second argument. At line number 31, we have invoked **TimeUnit.SECONDS.sleep(10)** because of which the task will be delayed by 10 seconds. This is not according to the time out condition mentioned in the **completeOnTimeout()** method, so this produces the default result as

10.

- **`public static Executor delayedExecutor(long delay, TimeUnit unit)`**

  This method is used to return the Executor. The responsibility of this Executor is to submit the task to the default Executor after the given delay provided in the argument.

  The method returns new delayed **`Executor`**.

- **`public static Executor delayedExecutor(long delay, TimeUnit unit, Executor executor)`**

  This method is used to return the Executor which is used to submit the task to the base **`Executor`** and not the default Executor. This base **`Executor`** is provided as the third argument in the method declaration.

  The method returns new delayed **`Executor`**.

  Observe the snippet from our solution. At line number 76, we have invoked **`delayedExecutor()`**. This method has the first argument 2 which is in seconds denoted as type of Unit in the second argument. It is followed by the **`thenAccept()`** lambda expression:

```
CompletableFuture.supplyAsync(() -> value1 + value2,
CompletableFuture.delayedExecutor(2, TimeUnit.SECONDS))
.thenAccept(result ->
 System.out.println("Result from delayedExecutor()==>
 "+result));
```

  So, this method will produce the result via the default **`Executor`** after 2 seconds and generates the output as 300 as shown.

- **`public static <U> CompletionStage<U> completedStage(U value)`**

  This method is used to return the new **`CompletionStage`** with the given value provided in the argument.

  The method returns completed **`CompletionStage`**

- **`public static <U> CompletableFuture<U> failedFuture(Throwable ex)`**

  This method facilitates to return the already **`CompletableFuture`** which is completed with an exception provided as an argument.

  The method returns exceptionally completed **`CompletableFuture`**

- **`public static <U> CompletionStage<U> failedStage(Throwable ex)`**

  This method facilitates to return **`CompletionStage`** which is completed with an exception provided as an argument.

The method returns exceptionally completed **`CompletionStage`**



If you want to generate the **`CompletableFuture`** based on some timeline, you can use the updated **`CompletableFuture`** API introduced in Java 9. It provides the methods like:

- **`orTimeOut()`**
- **`completeOnTimeOut()`**
- **`delayedExecutor()`**

Java Platform Module System

This is probably one of the most discussed updates in Java 9. The JPMS recommends using the module instead of using the packages directly. Packages will now become components of the module rather than an individual component.

Problem

How do we implement the new module system that is launched by Java 9? In what way it will be helpful to me in the entire development or deployment process?

Solution

To solve this, we will use 2 cases:

1. Working with a single module to understand what is a module and how it works internally.
2. Working with multiple modules to learn how to handle interdependencies between them.

Case 1: Working with a simple module

To understand this, please perform the following steps in your IDE. (Some of the steps/options may vary based on the IDE being used.):

1. Create a new project with the name **`MathModule`**. (Make sure that your compiler version is set to Java 9.x).
2. Observe that the create **`module-info.java`** is enabled. (You can even disable this at this moment and add this file later on.)

3. Once you create the new project, you will observe that the default **module-info.java** file is created as shown in **Listing 1-25**:

```
module MathModule {
}
```

   The module name by default matches with the project name.

4. Create a package **com.mathmodule**
5. Create a class **MathService.java** and add the method **chooseNumbers()** as shown in **Listing 1-26**:

```
1.  //Listing 1-26
2.  package com.mathmodule;
3.
4.  import java.util.ArrayList;
5.  import java.util.List;
6.
7.  public class MathService {
8.
9.    public static List<Integer>
10.    checkNumbers(List<Integer> numbers, int value){
11.
12.      List<Integer> numberList=new ArrayList<Integer>();
13.      numbers.forEach((x)->{
14.      if(x % value==0)
15.         numberList.add(x);
16.      });
17.      return numberList;
18.   }
19.
20. }
```

6. Create application class **MathMain.java** as shown in **Listing 1-27**.

```
1.  //Listing 1-27
2.  package com.mathmodule;
```

```
3.
4. import java.util.Arrays;
5. import java.util.List;
6.
7. public class MathMain {
8.
9.   public static void main(String[] args) {
10.    // TODO Auto-generated method stub
11.    List<Integer> generatedList=
12.
   MathService.checkNumbers(Arrays.asList(10,20,35,40), 2);
13.    generatedList.forEach(System.out::println);
14.    }
15. }
```

7. Execute **`MathMain.java.`**

   This will print all the elements of the arraylist which are divisible by 2.

Case 2: Using multiple modules

In this case, we will modify the same problem but this time, we will create 2 different modules and will communicate with them.

Follow the given steps:

1. Create a new project with the name **`MathServiceModule`**.
2. Create the package **`com.mathmoduleservice`**.
3. Copy the file **`MathService.java`** from the earlier project **`MathModule`** in this package. The class is listed in the project as **`Listing 1-30`**.
4. Modify the file **`module-info.java`** as shown in **`Listing 1-31`**:
   ```
   module MathServiceModule {
     exports com.mathmoduleservice;
   }
   ```
5. Create another project with the name **`MathProject`**.
6. Create a package **`com.mathproject`**.
7. Copy the file **`MathMain.java`** from the earlier project **`MathModule`** in this package. The class is listed in the project as **`Listing 1-32`**.

8. Observe that **`MathMain.java`** fails in compilation as it cannot refer to the **`MathService`** class which is present in another module.
9. To add the module reference in your project, follow the steps as follows:
    a. Right click on the project **`MathProject`** and select "**`Java Build Path`**".
    b. Go to the **`Project`** tab and select **`Modulepath`**.
    c. Click on the button "**`Add`**" and select **`MathServiceModule`** to add in your own project.
10. Modify the file **`module-info.java`** from **`MathProject`**:

```
module MathProject {
 requires MathServiceModule;
}
```

11. Import **`com.mathmoduleservice.MathService`** in your **`MathMain.java`** to resolve the dependency error.
12. Execute **`MathMain.java`**, which will print all the numbers divisible by 2.



We have only changed the structure of our program. There is no change in the code, only the modules are introduced. So, the output will be similar to what we observed in case1.

Explanation

Now, we will discuss both these cases one by one.

Case 1: Working with a simple module:

Right from its inception, Java is known for its package-centric architecture. Different classes are bundled in the single unit according to their functionalities, which is termed as a package. When we build and deploy our application, we always need to take the references of predefined classes which are part of the Java API.

All these classes are bundled into a single file termed as '**`rt.jar`**' which is provided to your application because of which you can use the import statement and refer to the classes as per your need. But such monolithic application utilizes a lot of memory as all the classes of Java are bundled into this '**`rt.jar`**' file. As of Java 8, the size of the rt.jar file is 64 MB. This is really a burden on the applications which are built by lightweight languages like Java.

The module system of Java 9 is a solution towards this monolithic approach. In this system, the java packages are grouped into different sets which are termed as modules. So, instead of using the entire API through '**rt.jar**', now you can add only those modules, the packages of which you would like to use in your application.

To list out different modules in Java, you can give the '**java --list-modules**' command as shown in *Figure 1.16*:

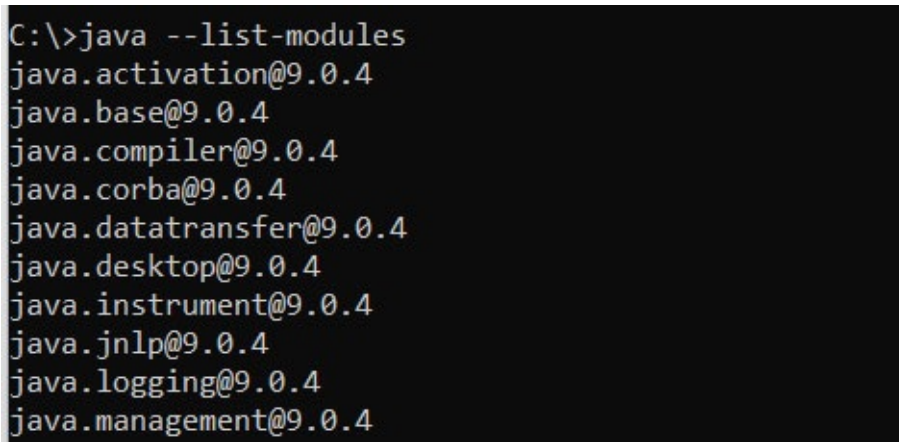


Figure 1.16: *Modules in Java 9*

This is just a cut short image, considering the size of the output. In all, there are currently 98 modules available in Java 9 and this number is still increasing. The default module which is provided to all the applications is the **java.base** module, which contains many packages which you can use to develop Java applications.

Creating applications based on modules is not much different than creating it using a normal package-oriented application. The only change here is you need to add a module inside which the packages will be residing. The module information is written in the '**module-info.java**' file which is kept in the **src** folder.

In our solution, we have created the **MathModule** project. This project also contains one package inside which we have kept the **MathService.java** and **MathMain**. **java** files. **MathService.java** file utilizes the **ArrayList** which is part of the **java.util** package which is imported in that class. As **MathMain** and **MathServices** are in the same package, they can communicate with each other without any extra import statements. So far there are no changes in the approach of building a project with the module concept. But if you observe, there is file **module-info.java** which is added in **src** which is presently empty.

This module descriptor file defines the following attributes of a module:

- Name of the module.
- Packages it can export which can be utilized by other modules.
- Modules it requires to use the dependencies.

In case 1, **`module-info.java`** contains only the module name, though the module is using **`ArrayList`**. This happens because every module by default uses **`java.base`** and this **`java.base`** also contains the package **`java.util`**.

So, the following snippet is as follows:

```
module MathModule {
}
```

Is equivalent to:

```
module MathModule {
  requires java.base;
}
```

Case 2: Working with multiple modules

When we work with multiple modules, we need to establish a relationship between different modules. This can be achieved by using different attributes or directives in **`module-info.java`** which are as follows:

- **`requires`**

  This directive specifies that the current module is dependent on another module. If the module is using the services from another module, then you must supply this directive.

  The syntax for using **`requires`** is as follows:

  ```
  requires modulename;
  ```

  In our solution, we have created two different modules: **`MathServiceModule`** and **`MathModule`**. **`MathModule`** is dependent on **`MathServiceModule`**, as it is using the service **`checkNumbers()`** defined within it.

  So, the **`module-info.java`** for **`MathModule`** is specifying that it requires the services from another module, that is, **`MathServiceModule`** as follows:

  ```
  module MathProject {
    requires MathServiceModule;
  }
  ```

- **`requires transitive`**

  This is also known as implied readability. Normally, this approach is used

if you are using a hierarchical approach in the module. Let us say if module B is specifying its dependency on module A. But module C is specifying its dependency on module B. In that case, if you want to say that whatever dependency module B specifies, should be given to module c as well, then you can use **`requires transitive.`**

The syntax for using requires transitive is as follows:

```
requires transitive modulename;
```

- **`exports`**

  This module directive specifies which packages from the current module are readable or available to other modules. This is a very important and interesting update from Java9, as the modules will not able to import any other class's service, unless the module inside which that class is present permits it.

  The syntax for export is as follows:

```
exports packagename;
```

  In our solution, as we know, **`MathModule`** specifies that it requires the services from **`MathServiceModule`**. But this won't be possible unless **`MathServiceModule`** exports the package so that the resources within it will be used by other modules. This is achieved by modifying the **`module-info.java`** of the **`MathServiceModule`** module as follows:

```
module MathServiceModule {
  exports com.mathmoduleservice;
}
```

- **`exports to`**

  This directive is modified or an advanced version of exports directive. Exports exposes its packages to all the modules. But **`exports .. to`** specifies which specific modules can use the packages and its resources.

  The syntax for export to is as follows:

```
exports packagename to module1,module2…;
```

- **`uses`**

  This module specifier is used to denote that the module requires some services. Unlike requires, this is always referred to the interface or abstract class the implementation of which is required.

  The syntax for uses is as follows:

```
uses servicerequired
```

- **`provides with`**

This module specifier is used to denote that the current module is providing some services which can be used by other modules. Essentially, it is the implementation of the abstract class or interface which is provided by the current module.

The syntax for provides with is as follows:

```
provides servicename with serviceimplementation
```



Using modules from Java 9, we can design more flexible and lightweight applications. It avoids the inclusion of unnecessary packages in an application. With different specifiers mentioned in the module-info.java file, it controls which packages can be exposed by a module or which modules are required by the particular module.

Conclusion

In this chapter, we discussed many new features which were launched in Java 9. We solved many problems statements to understand how these new features could be leveraged in a real-time scenario. Java 9 not only provided API changes but along with that it also introduced the concepts like **`Jshell`** and modules. Journeying through this chapter, we also discussed how the concept of module has changed the entire mindset of Java application designers while they design their projects.

In the next chapter, we will talk about creating some more recipes to understand the features introduced in Java 10.

Key terms

- takeWhile()
- dropWhile()
- iterate()
- List.of()
- Set.of()
- Map.of()
- Map.ofEntries()
- Arrays.compare()
- Arrays.mismatch()

- ifPresentOrElse(),or()
- stream()
- Colletors.toList()
- ObjectInputFilter
- checkInput()
- setObjectInputFilter()
- Subscriber
- Publisher
- Submission
- SubmissionPublisher
- completedOnTime()
- orTimeOut()
- delayedExecutor()

Questions

1. Why is the private method introduced in interfaces?
2. Explain the difference between takeWhile() and limit() of stream API.
3. In which situations, will you use dropWhile()?
4. How to create the unmodifiable List?
5. What is the use of Map.ofEntries()? How it is different from Map.of()?
6. Explain the working of Arrays.mismatch().
7. How to use ifPresentOrElse() from the Optional interface?
8. What is ObjectInputFilter.Status and what are its the possible values?
9. Explain the working of the submit() method of SubmissionPublisher.
10. What is the difference between requires and requires transitive directives in a Java module?

CHAPTER 2

Java 10 – Crack of a Dawn

Introduction

Releasing the updated version with the new API is always challenging for any the programming language. The process behind proposing and creating such a new API is time-consuming. Also, one needs to test it, so that the industry can accept it. Java was no exception to this process. Probably that is the reason when Oracle launched Java 9.0 in Sept 2017, it took them almost three years from the launch of Java 8.0. On one hand, we can say that such delays are good because you can rely more on such soundly testing API. But on the other hand, there is always pressure from developers and competitors. And of course, no one wants to lose customers. Also, the developers always need to be alert on the events for release. After giving it a lot of thought, Oracle came up with the new policy for releasing the new versions. The concept of "*time-based version release*" was introduced. Under this concept, it was decided that the new version will be released after every six months. Now, there is no any requirement that every version comes up with the major updates in the API. Sometimes, the version may contain only some minor updates or removals and incomplete JEPs may be shifted to next cycle. As per this policy, Java 10.0 was released after exactly 6 months, that is, in March 2010. Though Java 10.0 is not providing any major changes in Java API unlike Java 8.0 and Java 9.0, it introduced a few methods which we will study in this chapter.

Structure

In this chapter, we will cover the following concepts:

- Local Variable Type Inference
- Collection API enhancements
- Stream API enhancements
- Optional Interface enhancement
- Application Class data sharing

Objectives

After studying this unit, you should be able to use the new features introduced in Java 10. You should be able to use the features like **`copyOf()`** and **`orElseThrow()`** in your application so as to work more effectively with collections and Optional interface Installation

Installation

Visit the following URL to download Java 10 to follow all the code snippets in this chapter: **https://www.oracle.com/java/technologies/java-archive-javase10-downloads.html**

Local Variable Type Inference

From Java 10 onwards while declaring variables, developers need not have to mention the data type. The data type will be declared as var, and the data type of the identifier will be generated dynamically based on the value passed to that variable. This concept is treated as the local variable type inference.

Problem

Discuss how can we use the local variable type inference from Java 10.

Solution

We will discuss this with different scenarios:

Scenario 1: Basics of using the var keyword

Let us observe the basic situations, where we can use the **`var`** keyword. **`Listing 2-1, TypeInferenceDemo_Basic.java`**, shown as follows, describes the usage of the **`var`** keyword in the normal scenarios:

```
1. //Listing 2-1
2. package com.java10.typeinference;
3.
4. import java.util.ArrayList;
5. import java.util.Arrays;
6. import java.util.List;
```

```
import java.util.function.BiFunction;
import java.util.function.Function;

public class TypeInferenceDemo_Basic {

  public static void main(String[] args) {
   // TODO Auto-generated method stub
   var x=\·;//type of x is inferred as int
   var message="Java is simple to learn";// type of message is
                                          inferred as String
    var person\=new Person();//type of person1 is inferred as
Person
   var myList=getList();//type of myList is inferred according
to
                          return type of getList()
   //working with lambda expressing
   Function<Integer, Integer> process = value -> value + 10;
   var data=process.apply(10);

   //declaring the collection types
   var list = new ArrayList<>();
  }
  private static List<Integer> getList() {
   // TODO Auto-generated method stub
   return Arrays.asList(43,54,12,53);
  }
}
```

Scenario 2: Using the var keyword in polymorphism

The **var** keyword can be used in the polymorphism scenario as well. Suppose we have a **SalesPerson**, which is a type of **Person**. Then, we can use the **var** keyword to refer to the subtype using the reference of supertype as shown in **Listing 2-4, TypeInferencePolymorphism.java**. (**Listing 2-2** and **2-3** in the repo contains the code for **Person.java** and **SalesPerson.java**,

respectively):

```
//Listing 2-4
package com.java10.typeinference;

public class TypeInferencePolymorphism {
  public static void main(String[] args) {
   var p١=new Person();
   var p٢=new SalesPerson();

   p١=p٢;
   p٢=p١;//fails as subclass ref cannot point to superclass
  }
}
```

Scenario 3: Using the var keyword in loops

We can also use the **`var`** keyword in the looping statements as shown in **`Listing 2-5, TypeInferenceLoops.java.`**

```
//Listing 2-5:
package com.java10.typeinference;

public class TypeInferenceLoops {
  public static void main(String[] args) {
   traditionalLoop();
   enhancedLoop();
  }

  private static void enhancedLoop() {
   // TODO Auto-generated method stub
   for(var x=1;x<=10;x++) {
    System.out.println(x);
   }
```

```
  }

  private static void traditionalLoop() {
   // TODO Auto-generated method stub
   var intArray=new int[] {34,54,12,54};
   for(var x:intArray)
    System.out.println(x);
  }
}
```

Scenario 4: Using the var keyword in the anonymous class

To understand this, let us create an interface **`SampleInterface`** as shown:

```
//Listing 2-6
package com.java10.typeinference;

public interface SampleInterface {
public void processData();
}
```

Now, using the **`var`** keyword we can generate an anonymous implementation of this interface as shown in **`Listing 2-7, TypeInferenceAnonymous.java:`**

```
//Listing 2-7
package com.java10.typeinference;

public class TypeInferenceAnonymous {

  public static void main(String[] args) {
   // TODO Auto-generated method stub
   var sampleref=new SampleInterface() {

    @Override
    public void processData() {
```

```
12.       // TODO Auto-generated method stub
13.       System.out.println("Method Implemented");
14.      }
15.     };
16.
17.     sampleref.processData();
18.    }
19. }
```

Explanation

Declaring and initializing the variables is not new to developers. Whether you are Java developers or non-Java developers, you are omniscient. The practice of declaring variables like **`int x=10,`** is not incorrect. No one would have thought that there is any need for improvement in this pattern. But when you consider this pattern for declaring variables of different types, it makes the code clumsier. This also looks more ceremonial. So, in order to make the declarations more readable and to reduce the ceremony of writing the code, Java 10 introduced the **`var`** keyword. The **`var`** keyword also makes the variable dynamically typed.

Using the **`var`** keyword, we can infer that the data type must always be connected while declaring the variable.

The following are some examples of declaring the variables using the **`var`** keyword:

Statement 1:

```
var x=10;
```

Statement 2:

```
var mylist=new ArrayList<Integer>();
```

Statement 1, infers the declaration of int data type.

Statement 2, infers the declaration of **`ArrayList<Integer`**>

We can use the **`var`** keyword for local variables, loops, anonymous inner class implementation, and so on. However, there are some rules that you need to follow when you use the **`var`** keyword, which are as follows:

- Variables or members declared with **`var`** must be initialized while we declare it.

  For example:

```
var data; //incorrect
var data=10;//correct
```

- The initialization cannot be null, as the type inference cannot be identified with null.

  For example:
  ```
  var data=null;//incorrect
  var data=10;//correct
  ```

- The **var** keyword cannot be followed by multiple variables.

  For example:
  ```
  var data=10, count=1; //incorrect
  ```

- The initializer used for initializing cannot have another variable with initialization.

  For example:
  ```
  var data=(value=10); //incorrect
  ```

- If var is used for referring an array, we cannot initiate it without the data type.

  For example:
  ```
  var data={10,20,30}; //incorrect
  ```

- The **var** keyword cannot be followed by [], which defines the array dimension.

  For example:
  ```
  var[] data=new int[3]; //incorrect
  ```

- The **var** keyword cannot be directly used to point to the lambda expression because the lambda expression needs a target type to infer its types.

  For example:
  ```
  var data=value->value+20; //incorrect
  Function<Integer, Integer> data=value->value+20; //correct
  ```

- The **var** keyword cannot be used with static keywords and instance variables.
- We cannot use the **var** keyword in the formal argument of the lambda expression.

  For example:
  ```
  Function<Integer,Integer> process = (var value) -> value +
  10;  //incorrect
  ```



The var keyword is used to increase the readability of statements in Java 10. We do not have to declare the data type which will look similar to JavaScript. The type of identifier will be decided by the compiler based on the value passed to it.

Collection API enhancements

Java 10 provided an additional method **`copyOf()`** in the collection which is used to create collections like **`List`**, **`Set`**, and **`Map`** from the existing collections.

We will now walk-through different recipes in order to understand how to use this method in different collections.

Problem

How do we use different methods introduced in the Collection API of Java 10?

Solution

`Listing 2-8, CopyOfDemoList.java` shows the usage of the **`List.copyOf()`** method:

```
1. //Listing 2-8
2. package com.java10.collection;
3.
4. import java.util.Arrays;
5. import java.util.Collections;
6. import java.util.List;
7. import java.util.stream.Collectors;
8.
9. public class CopyOfDemoList {
10. public static void main(String[] args) {
11.   useCopyOfNotNull();
12.   useCopyOfWithNull();
13. }
14. private static void useCopyOfNotNull() {
```

```
15.   // TODO Auto-generated method stub
16.                         List<Integer>        intListOriginal=
    Arrays.asList(34,54,12,54,25);
17.   System.out.println("Original List ==>");
18.   intListOriginal.forEach((e)->System.out.print(e +" "));
19.   List<Integer> intListCopy=List.copyOf(intListOriginal);
20.    System.out.println("\nCopied List ==>");
21.    intListCopy.forEach((e)->System.out.print(e+" "));
22.   }
23.
24.   private static void useCopyOfWithNull() {
25.    // TODO Auto-generated method stub
26.                         List<Integer>        intListOriginal=
    Arrays.asList(٣٤,٥٤,١٢,54,null, 25);
27.    System.out.println("Original List ==>");
28.    intListOriginal.forEach((e)->System.out.print(e +" "));
29.    List<Integer> intListCopy=List.copyOf(intListOriginal);
30.    System.out.println("\nCopied List ==>");
31.    intListCopy.forEach((e)->System.out.print(e+" "));
32.   }
33. }
```

Output

If we execute **`CopyOfDemoList.java`**, we will get the output as shown in *Figure 2.1*:

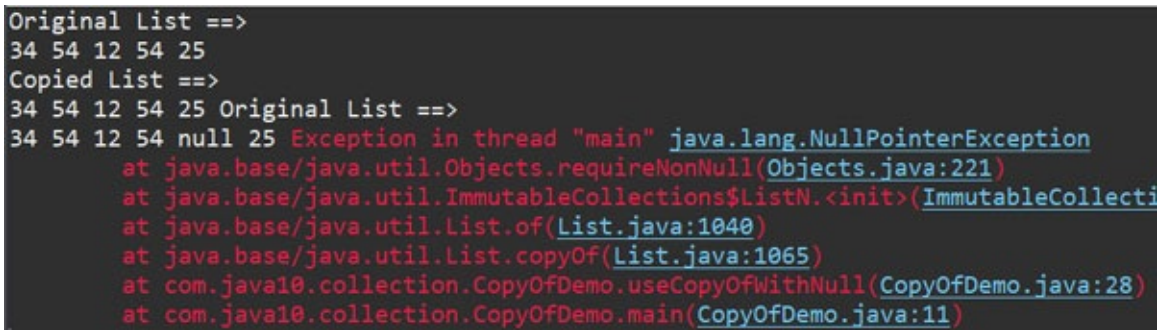


Figure 2.1: *Use of the List.copyOf() method*

Listing 2-10, CopyOfDemoSet.java shows the usage of the **Set.copyOf()** method:

```
1. //Listing 2-9
2. package com.java10.collection;
3. import java.util.Set;
4.
5. public class CopyOfDemoSet {
6.   public static void main(String[] args) {
7.    useCopyOfNotNull();
8.    useCopyOfWithNull();
9.   }
10.  private static void useCopyOfNotNull() {
11.   // TODO Auto-generated method stub
12.   Set<Integer>intSetOriginal =Set.of(54,23,58,15);
13.  //Set<Integer>intSetOriginal =Set.of(54,23,54,15);
14.   System.out.println("Original Set ==>");
15.   for(int number:intSetOriginal)
16.    System.out.println(number);
17.   Set<Integer> intSetCopy=Set.copyOf(intSetOriginal);
18.   System.out.println("Copied Set ==>");
19.   for(int number:intSetCopy)
20.    System.out.println(number);
21.  }
22.
23.  private static void useCopyOfWithNull() {
24.   // TODO Auto-generated method stub
25.   Set<Integer>intSetOriginal =Set.of(54,23,null,15);
26.   //Set<Integer>intSetOriginal =Set.of(54,23,54,15);
27.   System.out.println("Original Set ==>");
28.   for(int number:intSetOriginal)
```

```
29.      System.out.println(number);
30.     Set<Integer> intSetCopy=Set.copyOf(intSetOriginal);
31.     System.out.println("Copied Set ==>");
32.     for(int number:intSetCopy)
33.      System.out.println(number);
34.    }
35. }
```

Output

If we execute **`CopyofDemoSet.java`**, we will get the output as shown in *Figure 2.2*:

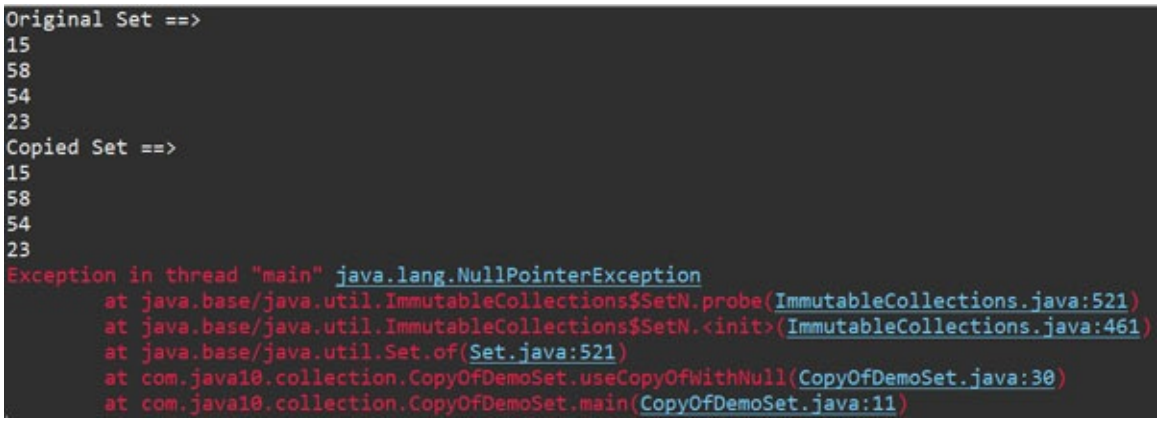


Figure 2.2: *Use of the Set.copyOf() method*

`Listing 2-10, CopyOfDemoMap.java` shows the usage of the **`Map.copyOf()`** method:

```
1. //Listing 2-10
2. package com.java10.collection;
3. import java.util.HashMap;
4. import java.util.Map;
5.
6. public class CopyOfDemoMap {
7.   public static void main(String[] args) {
8.    useCopyOfNotNull();
9.    //useCopyOfWithNull();
```

```
  }
  public static Map<Integer, String> createMap(){
   Map<Integer, String> map=new HashMap<Integer, String>();
   map.put(101, "Kornet");
   map.put(102, "John");
   map.put(103, "Bob");
   //map.put(null, "Bob");
   map.put(104, "Willis");
   return map;
  }
  private static void useCopyOfNotNull() {
   // TODO Auto-generated method stub

   Map<Integer, String> originalMap=createMap();

   System.out.println("Original Map ==>");
   System.out.println(originalMap);

    Map<Integer, String> copiedMap=new HashMap<Integer, String>
();
   copiedMap=Map.copyOf(originalMap);

   System.out.println("Copied Map ==>");
   System.out.println(copiedMap);

  }
  private static void useCopyOfWithNull() {
   // TODO Auto-generated method stub
   Map<Integer, String> originalMap=createMap();

   System.out.println("Original Map ==>");
```

```
40.     System.out.println(originalMap);
41.
42.     Map<Integer, String> copiedMap=new HashMap<Integer, String>
    ();
43.     copiedMap=Map.copyOf(originalMap);
44.
45.     System.out.println("Copied Map ==>");
46.     System.out.println(copiedMap);
47.
48.   }
49. }
```

Output

If we execute **`CopyOfDemoMap.java`**, we will get the output as shown in *Figure 2.3*:

```
Original Map ==>
{101=Kornet, 102=John, 103=Bob, 104=Willis}
Copied Map ==>
{104=Willis, 103=Bob, 102=John, 101=Kornet}
```

Figure 2.3: *Use of the Map.copyOf() method*

Explanation

Java 10 provided additional methods in the Collection API to create unmodifiable collections. The following are the different methods and their explanations:

- **`List.copyOf()`**
- **`Set.copyOf()`**
- **`Map.copyOf()`**

Let us discuss each of these methods in detail.

List.copyOf()

The **copyOf()** method is a static method of the List interface, which returns the unmodifiable List from the existing Collection.

The syntax of the method is as follows:

```
static <E> List<E> copyOf(Collection<? Extends E> mycoll)
```

The method parameters are:

- **E** is the type of element
- **mycoll** is the original collection, the copy of which needs to be created.

The method returns unmodifiable **List**, containing all the elements of the given collection, that is, **mycoll**. The iteration order of the collection is maintained.



- The collection should not be null or should not contain any null elements; otherwise, the process throws NullPointerException.
- If the original collection is modified after the invocation of the copyOf() method, the newly created list is not updated.
- If the original collection itself is unmodifiable, invoking copyOf() generally does not creates a copy.

In our solution, that is, **Listing 2-8**, we have invoked the **copyOf()** method on line number 19, which does not contain any null element as:

```
List<Integer> intListCopy=List.copyOf(intListOriginal);
```

This will create a copy of **intListOriginal** and store the entire list in the new list, that is, **intListCopy**.

But if you take a look at line number 26, we have declared a new List, which contains a list shown as follows:

```
List<Integer> intListOriginal= Arrays.asList(34,54,12,54,null,25);
```

So, when you try to invoke the **copyOf()** on the **intListOriginal** List, this generates **NullPointerException** as shown in the output snap.

Set.copyOf()

Similar to **List**, the Set interface also provides the static method **copyOf()** which creates an unmodifiable copy of Collection.

The syntax of the method is as follows:

```
static <E> Set<E> copyOf(Collection<? Extends E> mycoll)
```

The method parameters:

- **`E`** is the type of element.
- **`mycoll`** is the original collection, the copy of which needs to be created.

The method returns unmodifiable **`Set`**, containing all the elements of the given collection, that is, **`mycoll`**. The iteration order of the collection is maintained.



- The collection should not be null or should not contain any null elements; otherwise, the process throws NullPointerException.
- If the original collection is modified after the invocation of the copyOf() method, the newly created map is not updated.
- If the original collection itself is unmodifiable, invoking copyOf() generally does not create a copy.
- If the given collection contains any duplicate elements, then the arbitrary element of the duplicate element is persevered.

In our solution provided in **`Listing 2-9`**, we have used the **`copyOf()`** method of Set on line number 17 as shown in the following:

```
Set<Integer> intSetCopy=Set.copyOf(intSetOriginal);
```

So, the **`intSetCopy`** collection will be created as an unmodifiable Set generated from **`intSetOriginal`**.

But on the line number 25, we have created the Set which contains a null element shown as follows:

```
Set<Integer>intSetOriginal = Set.of(54,23,null,15);
```

As mentioned earlier, this generated **`NullPointerException`** when we tried to invoke the **`copyOf()`** on **`intSetOriginal`**. The snapshot of the same is shown in the output.

Map.copyOf()

The Map interface also provides a similar method to generate a copy of the existing Map.

The syntax of method is as follows:

```
static <K,V> Map<K,V> copyOf(Map<? Extends K,? extends V> map)
```

The method parameters are:

- **`K`**: the type of Key
- **`V`**: the type of Value

The method returns unmodifiable **`Map`**, which contains entries of the given **`Map`**.



- The map should not be null or should not contain any null elements; otherwise, the process throws NullPointerException.
- If the original collection is modified after the invocation of the copyOf() method, the newly created map is not updated.
- If the original collection itself is unmodifiable, invoking copyOf() generally does not creates a copy.

We have implemented this in our solution, **`Listing 2-10`**. On line number 29, we have used the following:

```
copiedMap=Map.copyOf(originalMap);
```

This statement will copy the contents of **`originalMap`** to the **`copiedMap`**. If we change any of the keys from **`originalMap`** to null, the program throws **`NullPointerException`**. So, try to uncomment **`map.put(null, "Bob")`**, and invoke the method **`useCopyOfWithNull()`**. You will observe that the program will throw **`NullPointerException`**.



To create an unmodifiable copy of the existing collection, you can invoke the copyOf() method. This method is implemented in List, Set, and Map interfaces in Java.

Stream API enhancement

Java 10 now contains an additional method introduced in the **`Collectors`** class. The static method **`UnmodifiableXXX()`** converts the existing collection into the unmodifiable collection.

Problem

Create an unmodifiable **`List`**, **`Set`**, and **`Map`** from the existing collection.

Solution

Listing 2-11, UnmodifiableDemo.java shows how to create the unmodifiable **List**, **Set,** and **Map**:

```
1. //Listing 2-11
2. package com.java10.streams;
3.
4. import java.util.List;
5. import java.util.Map;
6. import java.util.Set;
7. import java.util.stream.Collectors;
8. import java.util.stream.Stream;
9. public class UnModifiableDemo {
10.   public static void main(String[] args) {
11.    createUnModifiableList();
12.    createUnModifiableSet();
13.    createUnModifiableMap();
14.   }
15.   public static Stream<Employee> getStream(){
16.              return    Stream.of(new    Employee(101,"Joy"),new
    Employee(102,"Bob"));
17.   }
18.
19.   private static void createUnModifiableMap() {
20.    // TODO Auto-generated method stub
21.    Employee e١=new Employee(١٠١,"Joy");
22.    Employee e٢=new Employee(١٠٢,"Bob");
23.
24.                                    Map<Object,           Object>
    empMap=Stream.of(e1,e2).collect(Collectors.
25.         toUnmodifiableMap(e->e.getEmpId(),e->e));
26.
27.  //unModifiableMap() with binary function or Merge function
```

```
28.                                   Map<Object,           Object>
   empMap=Stream.of(e1,e2).collect(Collectors.
29.       toUnmodifiableMap((i)->(int)Math.random(),
                                              i->i ,(x,y)-
     >x+","+y));
30.    System.out.print"n("UnModifiable Map "=>");
31.    System.out.println(empMap);
32.
33.   }
34.   private static void createUnModifiableSet() {
35.    // TODO Auto-generated method stub
36.    Set<Employee> empSet = getStream()
37.      .collect(Collectors.toUnmodifiableSet());
38.    System.out.println("UnModifiable Set "=>");
39.    System.out.println(empSet);
40.
41.    //empSet.add(new Employee(13,"Smith"));
42.   }
43.   private static void createUnModifiableList() {
44.    // TODO Auto-generated method stub
45.    List<Employee> empList = getStream()
46.      .collect(Collectors.toUnmodifiableList());
47.    System.out.print"n("UnModifiable List "=>");
48.    System.out.println(empList);
49.   }
50. }
```

Output

If we execute **`UnModifiableDemo.java`**, we will get the output as shown in *Figure 2.4*:

```
UnModifiable List ==>
[Employee [empId=101, empName=Joy], Employee [empId=102, empName=Bob]]
UnModifiable Set ==>
[Employee [empId=101, empName=Joy], Employee [empId=102, empName=Bob]]
UnModifiable Map ==>
{101=Employee [empId=101, empName=Joy], 102=Employee [empId=102, empName=Bob]}
```

Figure 2.4: *Use of the toUnModifiableXXX() method*

Explanation

The **`toUnModifiableXXX()`** method is introduced in the stream API of Java 10. This is a static method that is added in the **`Collectors`** class. Using this method, we can convert the elements in the stream to **`List`**, **`Set`**, and **`Map`**.

The following are the different forms of this method:

Collectors.toUnModifiableList()

This method is cused to convert the stream element into an unmodifiable list.

The syntax of the method is as follows:

```
public static <T> Collector<T,?,List<T>> toUnmodifiableList()
```

The method parameters:

`T`: the type of elements in the Stream.

The method returns **`Collector`**, which accumulates the elements from the stream to an unmodifiable **`List`**.



The stream elements should not contain null element, else the method throws NullPointerException.

In our solution provided in **`Listing 2-11`**, on line number 47, we have invoked this method as follows:

```
List<Employee> empList = getStream()
        .collect(Collectors.toUnmodifiableList());
```

This creates an unmodifiable list **`empList`**, which is essentially a collection of all the elements which are collected by the collector. As this is an unmodifiable list, if you try to add an element to this list, it will throw an exception.

Collectors.toUnModifiableSet()

This method is used to convert the stream element into an unmodifiable set.

The syntax of the method is as follows:

```
public static <T> Collector<T,?,Set<T>> toUnmodifiableSet()
```

The method parameters:

T: the type of elements in the Stream.

The method returns **Collector,** which accumulates the elements from the stream to an unmodifiable **Set**. This set is unordered.



- The stream elements should not contain the null element, else the method throws NullPointerException.
- If the stream contains the duplicate element, then the arbitrary element from the duplicates is preserved as a part of the collector.

Try to uncomment line number 42, in **Listing 2-11, empSet.add(new Employee(13,"Smith"))** and execute the program again. You will observe the exception as **java.lang.UnSupportedOperationException**.

Collectors.toUnmodifiableMap()

This method is used to collect the stream elements in the **Map**. There are 2 forms of this method.

Form 1:

This is simplest form of the method, which provides 2 mapping functions to define the key and values of the corresponding generated map.

The syntax of the method is a s follows:

```
public static <T,K,U> Collector<T,?,Map<K,U>> toUnmodifiableMap
  (Function<? super T,? extends keyFunction> key,
    Function<? super T,? extends U> valueFnction)
```

The method returns **Collector,** which accumulates the input elements into the unmodifiable map. The keys and values are applied through the mapping function which is applied to input elements.

The method parameters are:

- **keyFunction**: Mapping function to generate the keys.
- **valueFunction**: Mapping function to generate the values.

The type parameters are:

- **T**: Type of input element
- **K**: Type of key generated from **keyFunction**

- **`U`**: Type of value generated from **`valueFunction`**



If the mapped keys contain duplicates, IllegalStateException is thrown.

In our solution provided in **`Listing 2-11`**, on line number 24, we have invoked this method as follows:

```
Map<Object, Object> empMap=Stream.of(e1,e2)
       .collect(Collectors.toUnmodifiableMap(e)->e.getEmpId(),e-
>e));
```

This creates the unmodifiable map from stream elements which are collected by the collector. But the map contains the key, value pair, and the stream contains only elements without keys. So, to generate the keys and value combination, we have to provide to lambdas representing the Function interface. The first lambda takes the element e, as a parameter and the key is generated by invoking **`e.getEmpId()`**, so each employee will have the key as **`empId`**. The value is generated by the second lambda expression which takes the employee object e as a parameter, that is, the actual **`Employee`** object.

Form 2:

In this form of method, another function is provided as an argument which is called as a merge function. This function is used to merge the results for the duplicate keys if any, present in the given collector.

The syntax of the method is as follows:

```
public static <T,K,U> Collector<T,?,Map<K,U>> toUnmodifiableMap
   (Function<? super T,? extends K> keyFunction,
     Function<? super T,? extends U> valueFunction,
     BinaryOperator<U> mergeFunction)
```

The method returns **`Collector`**, which accumulates the input elements into an unmodifiable map. The keys and values are applied through the mapping function which is applied to input elements.

The method parameters are:

- **`keyFunction`**: Mapping function to generate the keys.
- **`valueFunction`**: Mapping function to generate the values.
- **`mergeFunction`**: Merger function for values associated with duplicate keys.

The type parameters are:

- **`T`**: Type of input element
- **`K`**: Type of key generated from **`keyFunction`**
- **`U`**: Type of value generated from **`valueFunction`** and **`mergeFunction`**



If the mapped keys contain duplicates, IllegalStateException is thrown.

In our solution provided in **`Listing 2-11`**, on line number 28, we have invoked this method as follows:

```
Map<Object, Object> empMap=Stream.of(e1,e2)
  .collect(Collectors.toUnmodifiableMap((i)->(int)Math.random(),
     i->i,(x,")->"+", "+y));
```

Similar to the earlier form, this creates the map with the key value. But along with this, it also takes care of duplicate keys. The third lambda expression sets the merger value for the duplicate keys. So, if you uncomment the code in the actual listing and execute it again, you will get the output as shown in *Figure 2.5*:

```
UnModifiable List ==>
[Employee [empId=101, empName=Joy], Employee [empId=102, empName=Bob]]
UnModifiable Set ==>
[Employee [empId=102, empName=Bob], Employee [empId=101, empName=Joy]]
UnModifiable Map ==>
{0=Employee [empId=101, empName=Joy] , Employee [empId=102, empName=Bob]}
```

Figure 2.5: *Use of the toUnModifiableXXX() method with the merger function*

The **`toUnmodifiableMap()`** creates the duplicate key because of the **`(int)Math.random()`** function supplied as a key generation function. So, in order to avoid the arbitrary value, the merger function is provided as **`(x,y)->x+", "+y`**. This merges the duplicate employee objects with a comma (,) as a separator.



The elements from the streams can be converted to the unmodifiable List, Set, or even Map by using toUnmodifiableXXX(). To convert the stream to the map, special functions should be provided to generate the key and value.

Optional Interface Enhancement

The Optional interface is one of the favorite features of the developers. Java

constantly monitors what change they can do to make this interface more reliable. As a result of such dedicated efforts, Java added one more method **`orElseThrow()`** to the Optional interface. This method throws **`NoSuchElementException`**, if the object is not present.

Problem

We have a list of Employees. We would like to find the first occurrence of Employee based on some condition. Also, if the Employee is not found, the program should throw some exception.

Solution

This problem can be solved by the solution provided in **`Listing 2-12, OrElseThrowDemo.java`**:

```
1. //Listing 2-12
2. package com.java10.streams;
3.
4. import java.util.Arrays;
5. import java.util.List;
6. import java.util.NoSuchElementException;
7. import java.util.Optional;
8.
9. public class OrElseThrowDemo {
10. static List<Employee> employeeList
11.       = Arrays.asList(new Employee(104,"Bob Williams"),
12.         new Employee(101,"Johny Hiscus"),
13.         new Employee(103,"Alan Marsh"));
14.   public static void main(String[] args) {
15.    // TODO Auto-generated method stub
16.
17.     for(int id=101;id<=105;id++) {
18.      int localID=id;
19.      Optional<Employee> findEmpById =
```

```
                employeeList.stream()
                .filter(e->e.getEmpId()==localID)
                .findFirst();
    try {
     System.out.println(findEmpById.orElseThrow());
    }
    catch(NoSuchElementException elementException) {
           System.out.pr"ntln("Employee  wit"  Id:""+id+"  not
f"und!!");
    }
   }
  }
}
```

Output

If we execute **`OrElseThrowDemo.java`**, we will get the output as shown in *Figure 2.6*:

```
Employee [empId=101, empName=Johny Hiscus]
Employee with Id: 102 not found!!
Employee [empId=103, empName=Alan Marsh]
Employee [empId=104, empName=Bob Wiliams]
Employee with Id: 105 not found!!
```

Figure 2.6: *Use of orElseThrow() method*

Explanation

Java 10 provided another method **`orElseThrow()`** that can be used with the Optional interface for exception handling. Instead of using the **`if.. else`** block with the **`ifPresent()`** method, we can use this method. This method returns the object or value if present, else it throws **`NoSuchElementException`**.

The syntax of method is as follows:

`public T orElseThrow()`

The method returns a value, if present, else it throws **`NoSuchElementException`**.

The type parameter:

T: Type of value or object.

In our solution **Listing 2-12**, on line number 23, we have used the following:

```
try {
  System.out.println(findEmpById.orElseThrow());
}catch(NoSuchElementException elementException) {
 System.out.pr"ntln("Employee wit" Id:""+id+" not f"und!!");
}
```

The **findEmpById** is of type Optional, which is generated by the terminal function **findFirst()** on the stream of Employees. So, the **orElseThrow()** method will print the value or object if found by **findFirst()**, else throws **NoSuchElementException** which is handled in the **try.. catch** block.



The Optional's orElseThrow() method is used to return the object or value if present. If the object or value is not present, it throws NoSuchElementException which can be handled. This avoids unnecessary code block of ifPresent() which needs to be inserted into the if condition.

Application Class Data Sharing

Application Class Data Sharing is the new concept introduced in Java 10. By this concept, the required system classes and application classes are shared through the user-defined dump file created by the developer.

Problem

Can you elaborate with different steps, how we can use the feature of Application Class Data Sharing introduced in Java 10?

Solution

We will use two classes to implement this. **Employee.java** which is shown in **Listing 2-13**.

```
1. //listing 2-13
2. public class Employee{
3.     private int empId;
4.     private String empName;
5.     public Employee(int empId,String empName){
6.         this.empId=empId;
```

```
7.        this.empName=empName;
8.     }
9.     public String toString(){
10.       return this.empId+", "+this.empName;
11.    }
12. }
```

Another class that we will use is **EmployeeTest.java** as shown in **Listing 2-14**:

```
1. //Listing 2-14
2. import java.util.Arrays;
3. import java.util.List;
4.
5. public class EmployeeTest {
6.
7.   public static void main(String[] args) {
8.    // TODO Auto-generated method stub
9.                    List<Employee>      empList=Arrays.asList(new
   Employee(101,"John"),
10.                          new Employee(102,"Dave"));
11.    for(Employee e: empList) {
12.     System.out.println(e);
13.    }
14.   }
15. }
```

These two are very simple classes and self-explanatory. Observe that EmployeeTest.java uses system classes **Arrays** and **ArrayList**. At the same time, it uses the application class **Employee**.

The following steps are followed to achieve the Application Class Data Sharing:

Step 1: Creating a dump of classes

Execute the command as shown in *Figure 2.7* on console:

```
D:\>java -Xshare:dump
narrow_klass_base = 0x000001dd60460000, narrow_klass_shift = 3
Allocated temporary class space: 1073741824 bytes at 0x000001de20460000
Allocated shared space: 3221225472 bytes at 0x000001dd60460000
Loading classes to share ...
Loading classes to share: done.
Rewriting and linking classes ...
Rewriting and linking classes: done
Number of classes 1297
    instance classes   =  1235
    obj array classes  =    54
    type array classes =     8
Updating ConstMethods ... done.
Removing unshareable information ... done.
Scanning all metaspace objects ...
Allocating RW objects ...
Allocating RO objects ...
Relocating embedded pointers ...
Relocating external roots ...
Dumping symbol table ...
Relocating SystemDictionary::_well_known_klasses[] ...
Removing java_mirror ... done.
```

Figure 2.7: *Re-generating dump*

Step 2: Creating a list of shared classes

In this step, we need to create the list of the shared classes and store it in the local file. Execute the command as shown in *Figure 2.8* on console*:*

```
D:\>java -XX:+UnlockCommercialFeatures -XX:+UseAppCDS -XX:DumpLoadedClassList=employee.lst EmployeeTest
101 , John
102 , Dave
```

Figure 2.8: *Creating a list of shared classes*

This creates the **`employee.lst`** file, which contains the list of the classes referred. The snippet of the same is shown as follows:

```
emp.lst
-------------
java/lang/Object
java/lang/String
java/io/Serializable
java/lang/Comparable
java/lang/CharSequence
java/lang/Class
java/lang/reflect/GenericDeclaration
java/lang/reflect/AnnotatedElement
java/lang/reflect/Type
java/lang/Cloneable
java/lang/ClassLoader
java/lang/System
java/lang/Throwable
java/lang/Error
```

```
java/lang/ThreadDeath
java/lang/Exception
java/lang/RuntimeException
java/lang/SecurityManager
java/security/ProtectionDomain
java/security/AccessControlContext
java/security/SecureClassLoader
java/lang/ClassNotFoundException
java/lang/ReflectiveOperationException
java/lang/NoClassDefFoundError
java/lang/LinkageError
java/lang/ClassCastException
java/lang/ArrayStoreException
…..
…..
```

Step 3: Creating a dump of shared archive file

Refer to *Figure 2.9*, here we will execute the following command:

```
D:\>java -XX:+UnlockCommercialFeatures -XX:+UseAppCDS -Xshare:dump -XX:SharedClassListFile=employee.lst
-XX:SharedArchiveFile=emp.jsa EmployeeTest
narrow_klass_base = 0x0000000800000000, narrow_klass_shift = 3
Allocated temporary class space: 1073741824 bytes at 0x00000008c0000000
Allocated shared space: 3221225472 bytes at 0x0000000800000000
Loading classes to share ...
Loading classes to share: done.
Rewriting and linking classes ...
Rewriting and linking classes: done
Number of classes 645
    instance classes   =   582
    obj array classes  =    55
    type array classes =     8
Updating ConstMethods ... done.
Removing unshareable information ... done.
Scanning all metaspace objects ...
Allocating RW objects ...
Allocating RO objects ...
Relocating embedded pointers ...
Relocating external roots ...
Dumping symbol table ...
Relocating SystemDictionary::_well_known_klasses[] ...
Removing java_mirror ... done.
mc  space:      5440 [  0.1% of total] out of     65536 bytes [  8.3% used] at 0x0000000800000000
rw  space:   2060536 [ 21.5% of total] out of   2097152 bytes [ 98.3% used] at 0x0000000800010000
ro  space:   3906632 [ 40.8% of total] out of   3932160 bytes [ 99.4% used] at 0x0000000800210000
md  space:      6160 [  0.1% of total] out of     65536 bytes [  9.4% used] at 0x00000008005d0000
od  space:   3373096 [ 35.3% of total] out of   3407872 bytes [ 99.0% used] at 0x00000008005e0000
total    :   9351864 [100.0% of total] out of   9568256 bytes [ 97.7% used]
```

Figure 2.9: *Creating a dump of shared classes*

Step 4: Using a shared dump to launch the app

In this step, we will launch the application using the shared dump which we have created earlier, as shown in *Figure 2.10*:

```
D:\>java -XX:+UnlockCommercialFeatures -XX:+UseAppCDS -Xshare:on -XX:SharedArchiveFile=emp.jsa EmployeeTest
101 , John
102 , Dave
```

Figure 2.10: *Using AppCDS feature*

Explanation

Class Data sharing is already there till Java 9.0. So, before we discuss about what is Application Class Data Sharing, let us discuss briefly about Class Data Sharing and how it is achieved.

If you go to your **`jre.bin/server`** folder you will observe, there is a file named **`classes.jsa`**, as shown in *Figure 2.11*:

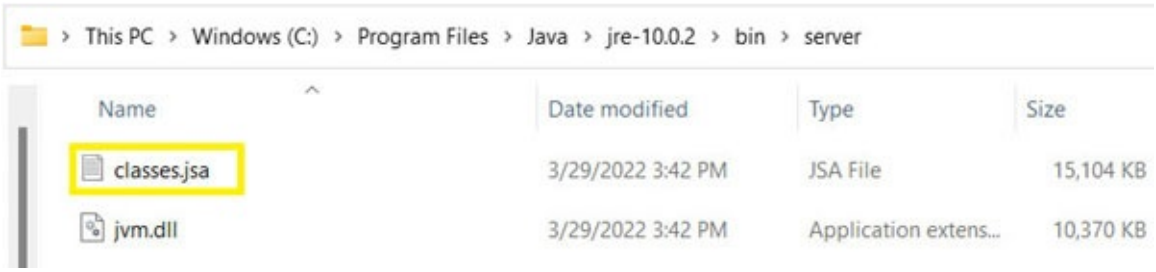


Figure 2.11: *Location of classes.jsa*

This file is the dump of all converted forms of many System classes. So, every time when your application is launched this file is loaded for the reference. This file is dependent on another file, titled **`classlist`** which is present in **`jre/lib`**. Refer to *Figure 2.12*:

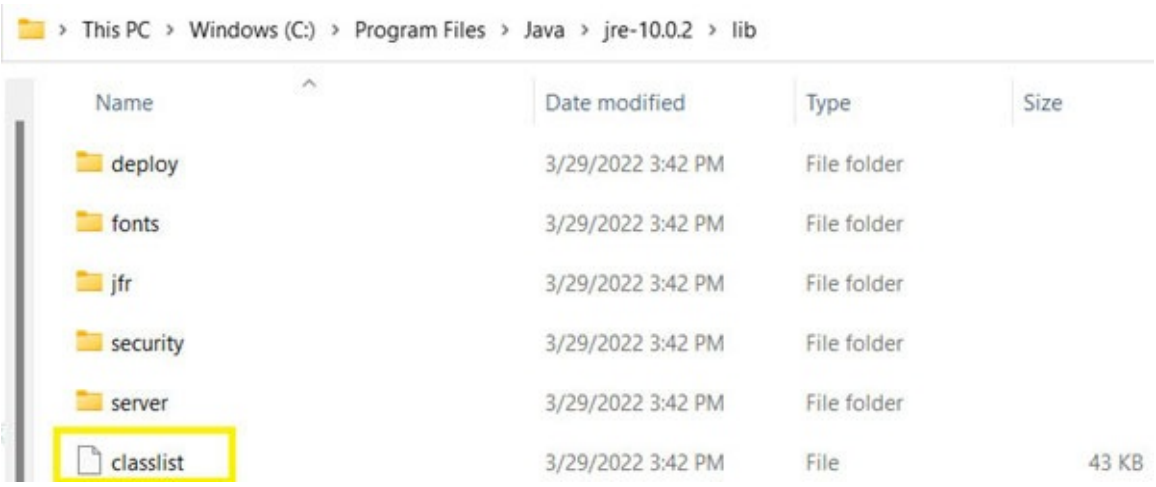


Figure 2.12: *Location of the classlist*

The **`classes.jsa`** file can be created manually using the command **`java -Xshare:dump`**. This command reads the classes from **`classlist`** and creates a

dump of the same. We have shown this step in our solution.

From Java 10 onwards, instead of loading the classes from the source, you can create your own dump. To generate such dump file you need to use the option **`-XX:+UseAppCDS`** which is available from Java 10. When you launch your application, you can provide the reference of this dump file so that the classes will be loaded from your own dump file and not from the source file. This will improve the performance of your application.

To achieve this, you need to perform the following steps:

1. Record all the classes that you required in the file.

   This can be achieved by the following command:
   ```
   java -XX:+UnlockCommercialFeatures -XX:+UseAppCDS -
   XX:DumpLoadedClassList=<classlist_filename>
   ```
2. Create a dump file for this list.

   This can be achieved by the following command:
   ```
   java -XX:+UnlockCommercialFeatures -Xshare:dump -XX:+UseAppCDS
   -XX:SharedClassListFile=<classlist_filename> -
   XX:SharedArchiveFile=<name of jsa file>
   ```
3. Using the shared dump while launching the app
   ```
   java -XX:+UnlockCommercialFeatures -Xshare:on -XX:+UseAppCDS -
   XX:SharedArchiveFile=<name of jsa file> <app_class_name>
   ```



Application Class Data Sharing is a process of sharing the required classes from the separate dump file instead of loading the same from the source file. You need to use -XX:+UseAppCDS option to achieve this. This improves performance of your application.

Conclusion

In this chapter, we discussed the important updates that were introduced as a part of Java 10.0. We designed recipes to use the **`var`** keyword along with the methods like **`copyOf()`** and **`toUnModifiableXXX()`**. We also learned how to implement the concept of Application Class Data Sharing to improve the performance of an application. In the next chapter, we will walk through the updates provided in Java 11.

Questions

1. How to create the immutable copy of **`List`** from the existing list?

2. If the original set contains the null element, can the copy of that set created by **`Set.copyOf()`** permit the null value?
3. How to create the unmodifiable map from the stream created from Map?
4. What are the different forms of the **`toUnmodifiableMap()`** method?
5. What is the use of **`orElseThrow()`** that is used in the Optional interface?

Key Terms

- List.copyOf()
- Set.copyOf()
- Map.copyOf()
- Collectors.toUnModifiableList()
- Collectors.toUnModifiableSet()
- Collectors.toUnModifiableMap()

CHAPTER 3

Java 11 – Crack of a Dawn

Introduction

In the previous chapter, we extensively discussed Java 10.0. We discussed many features programmatically as well. But while we did that, you might have observed that there were not many features launched by version 10.0. When a new version is launched, we always have some expectations from that version. We expect that something dramatic if not lengthy will be introduced in the version, which will change the mindset of the developer all together. Unfortunately, Java 10.0 did not live up to that expectation. But it was expected from the software developers, not from API developers. Oracle never promised that each version launched will consist of a lot of changes. What was promised was to release the version every six months with some changes. The changes would be in the form of addition of features, updates in the existing features, or even may be removal of some features.

Based on the similar pipeline, the new version, that is, Java 11.0 was launched on 25th September 2018. Similar to the earlier version, in this version also there are no major changes in API. There are a lot of features which work internally which helps to improve the performance of the application. Many of the features which were deprecated in earlier versions are removed.

Structure

In this chapter, we will cover the following concepts:

- Nest-based access control
- Updates in the Reflection API
- String API updates
- Updates in the reflective access of the nested class
- Local variable syntax for lambda parameters
- HttpClient (Standard)

- Launch Single-File Source-Code Program

Objectives

After studying this unit, you should be able to work with the new features introduced in Java 11.0. We will create some recipes to utilize each of the flavors of these features. A few of the features are not implementation-oriented, so we will just discuss such features.

Installation

Visit the URL as follows to download Java 11, and to follow all the code snippets in this chapter:

https://www.oracle.com/in/java/technologies/javase/jdk11-archive-downloads.html

Nest-based access control

The concept of Inner classes is used in Java for years. The way the communication happens between the inner class and private members of the outer class is changed in this version. Let us discuss this with some recipes.

Problem

Explain the concept of nest-based access control in Java 11.

Solution

Consider the following code:

```
1. //Listing 3-1
2. package com.nestbasedaccess;
3.
4. public class Car {
5.   private String carType;
6.   public Car(String type) {
7.    this.carType = type;
8.   }
```

```
  class Engine {
   String engineType;
   void setEngine() {
    if(carType.equals("disel"))
     this.engineType="customized";
    else
     this.engineType="standard";
   }

   String getEngineType() {
    return this.engineType;
   }
  }
}
```

Output

To understand how the nest-based access control works, we need to use the **`javap`** command. This command disassembles the classes. This command prints the public members and methods of the class.

We will use the **`javap`** command in the pre-Java 11 compiler version and Java 11 compiler and observe the output.

Scenario 1: Using Java 9 compiler

If we use Java 9 compiler to execute the **`javap`** command, we will get the output as shown in *Figure 3.1*:

```
D:\>javap Car.class
Compiled from "Car.java"
public class Car {
  public Car(java.lang.String);
  static java.lang.String access$000();
}
```

Figure 3.1: *Nest-based access using the bridge*

Scenario 2: Using Java 11 compiler

If we use Java 11 compiler to execute the **`javap`** command, we will get the following output shown in F*igure 3.2*:

```
D:\>javap Car.class
Compiled from "Car.java"
public class Car {
  public Car(java.lang.String);
}
```

Figure 3.2: *Nest-based access without the bridge*

Explanation

The nest-based access control is observed in the inner class implementation. As we all know, an inner class is an approach where we write the class in another class. This concept is also termed as a nested class. As both the classes are bundled into a single file, the inner class have the access to all the members of the outer class. This includes the private members of the outer class as well. Though this looks not complicated, it violates the rule of privacy. We declare the members as private because we do not want outsiders to access the same.

So, to bypass this restriction for private keywords, the compiler internally creates a bridge for such private keywords. This bridge converts the invocation of private data members package-level private member invocation method.

Such a bridge is used to get created prior to Java 11.0

In our **`Listing 1-1`**, we have an outer class **`Car`**, which contains the inner class **`Engine`**. The **`Car`** class declares one private data member **`carType`**. So, when this class is compiled, the private member is converted to the bridge method as **`access$000()`**. Such a method internally returns the private member for which the bridge is created.

For instance, for a private member **`carType`**, the bridge method is created as follows:

```
String access$000(){
  return carType;
}
```

To observe this, you have to execute the **`javap`** command as shown in Figure 3.1. This command is executed using Java 9 compiler, that is, pre-Java 11 compiler.

But from Java 11, there is no need to create such a bridge to access the private data member from the inner class. So, the same **`javap`** command if we give from

Java 11 compiler, you will observe that there is no presence of **`access$000`** as shown in *Figure 3.2*. Similarly, if you have the private method in the outer class, the bridge to access the same will be created by the compiler.



The nest-based access control is the way by which we can facilitate the inner class to access the private members of the outer class without creating a bridge. This reduces the burden on the deployer as the additional code is not added by the compiler. Also, this would increase the speed of invocation of private members from the inner class.

Updates in the reflection API

The reflection API in Java revolves around **`java.lang.Class`**. This class now provides some additional methods like:

- **`getNestHost()`**
- **`getNestMembers()`**
- **`isNestmateOf()`**

We will discuss all these methods in detail with some recipes now.

Problem

How do we identify the nest hosts or nest members of the classes if any? How does the new API of Java 11 facilitate the same?

Solution

`Listing 2-3, ReflectionAPIDemo.java`, shows the in-detail implementation of the different methods introduced in this API:

```
1. //Listing 3-2
2. package com.nestbasedaccess;
3.
4. import java.util.Arrays;
5. import java.util.Set;
6. import java.util.stream.Collectors;
7.
```

```
import com.nestbasedaccess.Car.Engine;

public class ReflectionAPIDemo {

  public static void main(String[] args) {
 // TODO Auto-generated method stub

   Car car=new Car("Disel");
   Engine engine=car.new Engine();
              System.out.println("Host    name    for    Engine class:"+hostName(engine));
   if(checkNestMate(engine)==true)
    System.out.println("Engine is nestmate of Car");
   else
    System.out.println("Engine is not nestmate of Car");
   Set<String> nestMembers=findNestMembers(engine);
   System.out.println("Nestmembers of Engine are: ==>");
   int counter=·;
   for(String member:nestMembers)
    System.out.println(++counter+ ". "+member+" ");
  }

  public static String hostName(Object o){
   return o.getClass().getNestHost().getName();
  }
  public static boolean checkNestMate(Object o) {
   return o.getClass().isNestmateOf(Car.class);
  }

  public static Set<String> findNestMembers(Object o){
   return Arrays.stream(o.getClass().getNestMembers())
```

```
                                        .map(Class::getName)
                                        .collect(Collectors.toSet());
    }
}
```

Output

The output shown in *Figure 3.3* is displayed if we execute **`ReflectionAPIDemo.java`**:

```
Host name for Engine class:-com.nestbasedaccess.Car
Engine is nestmate of Car
Nestmembers of Engine are: ==>
1. com.nestbasedaccess.Car$Engine
2. com.nestbasedaccess.Car$SteeringWheel
3. com.nestbasedaccess.Car
```

Figure 3.3: *Updated Reflection API*

Explanation

Let us now discuss all the methods that we have implemented or invoked in our solution in detail.

getNestHost()

The **`getNestHost()`** method is used to get the details of the host class inside which the nest class is embedded. Even if your class is contained within an interface, this method will get the interface details.

The syntax of the method is as follows:

```
public Class<?> getNestHost()
```

The method returns:

- The nest host class or interface if it is declared inside another class.
- If this method is invoked on the class which is not nested, then it itself is considered to be the host. In this case, the method returns the same class as the nest host.

In our **`Listing 3-2`**, the class **`Engine`** is nested in Car. We can invoke the **`getNestHost()`** on the Engine object as shown in the method as follows:

```
public static String hostName(Object o){
  return o.getClass().getNestHost().getName();
```

```
}
```

So, in this case, as **`o`** is **`instanceof Engine`**, the **`getNestHost().getName()`** returns **`com.nestbasedaccess.Car`** (fully qualified name of Car) as shown in the output. If we change the object type to **`Car`**, the output would be still the same as **`Car`** is not nested inside any other class.

isNestmateOf()

Another method added in the reflection API is **`isNestmateOf()`**.

The syntax of the method is as follows:

```
public boolean isNestmateOf(Class<?> c)
```

The method parameters:

```
Type Class<?> c, which is to be checked
```

The method returns:

- True, if the current class and the **`Class<?> c`** are members of the same outer class or the members of the same nest host.
- False, if the current class and the **`Class<?> c`** are not the members of the same outer class or the members of the same nest host.

In our solution, **`Listing 3-2`**, we have invoked this method on line number 32, as shown here:

```
public static boolean checkNestMate(Object o) {
  return o.getClass().isNestmateOf(Car.class);
}
```

In this method, we have passed the Engine object as an argument. As we already know, **`Car`** is the outer class and it does not contain another nest host. In this scenario, the nest host of the **`Car`** is **`Car`** itself. So, when we compare the **`Car`** and **`Engine`** object with the **`isNestmateOf()`** method, it returned true.

getNestMembers()

This method is used to find the other nest members of the given nest host.

The syntax of the method is:

```
public Class<?>[] getNestMembers()
```

The method returns:

- An array of the **`Class`** objects, which is a group of all the classes and interfaces which are nest members of the current class.
- Along with the nest members, this array also contains the nest host of the given class. As it is denoted as the nest host of itself. This class is always placed at the first element of the array which is followed by other classes

or interfaces present within it.

- If the class does not contain any nest members, the method returns only **`this`**.

In order to understand this concept clearly, we have added one more nest **`SteeringWheel`** in the **`Car`** class as shown here:

```
public class Car {
…
  class Engine {
  …}
  class SteeringWheel {
}
}
```

In our solution, **`Listing 3-2,`** we have invoked the **`getNestmembers()`** method on the **`Engine`** class on line number 35, as follows:

```
public static Set<String> findNestMembers(Object o){
return Arrays.stream(o.getClass().getNestMembers())
             .map(Class::getName)
             .collect(Collectors.toSet());
}
```

We have passed the Engine on line number 22, as shown here:

```
Set<String> nestMembers=findNestMembers(engine);
```

Now, as the method is invoked on the **`Engine`** class, the output for this invocation starts from **`Car`**. **`Engine`** as the first element in the array followed by **`SteeringWheel`** and also Car as the nest host of the Car is Car itself.



Java 11 provides more functionalities to retrieve information about the nested class. The getNestHost() method is used to get the host class of the current class. You can use the nestOf() method to check whether the given class is a nest of the other class or not. Also, there is another method getNestMembers() which returns a list of the other nest members of the class for the given host class.

String API updates

The String API is already rich in Java. Though there are no major changes in this API, Java 11 introduced some additional methods to enhance the white space handling in Strings. Along with this, it also provides some more methods for processing the Strings.

The following are the different methods:

- **strip()**
- **stripLeading()**
- **stripTrailing()**
- **isBlank()**
- **lines()**
- **repeat()**

Problem

Discuss new methods introduced in Strip. Also, explain how these methods behave differently than the existing methods.

Solution

Listing 3-3, StringAPIDemo.java, shows different methods introduced in the String API:

```
1. //Listing 3-3
2. package com.strings;
3.
4. public class StringAPIDemo {
5.
6.   public static void main(String[] args) {
7.    // TODO Auto-generated method stub
8.    System.out.println("Stripping the String==>");
9.    String originalString="\u٢٠٠١ \t BPB Publications \t \u٢٠٠٢
";
10.                                    System.out.println("Stripped
String="+originalString.strip());
11.    System.out.println("Trimmed String ="+(originalString
                                                    .trim())+"\n");
12.
13.
14.     System.out.println("Stripping leading and training spaces
=>");
```

```
15.    System.out.println("Leading Strip  ="+originalString
       .stripLeading());
16.    System.out.println("Trailing Strip ="+
       originalString.stripTrailing()+"\n");
17.
18.    System.out.println("Checking the blank String=>");
19.    String blankString="\u٢٠٠١";
20.   System.out.println("Is String blank ="+blankString.
                                              isBlank()+"\n");
21.    System.out.println("Generating lines from the String
                                                 using lines()
       ==>");
22.    String text="This book contains recipes for Java.\n
                              It covers the concept from Java 9 to
       18";
23.    text.lines().forEach((line)->System.out.println(line));
24.    System.out.println("\n");
25.
26.           System.out.println("Repeating   the   String   using
    repeat()==>");
27.    String bookName="Java ٩+ Recipes!";
28.    System.out.println(bookName.repeat(0));
29.   }
30. }
```

Output

If we execute **`StringAPIDemo.java`**, you will observe the following output as shown in F*igure 3.4*:

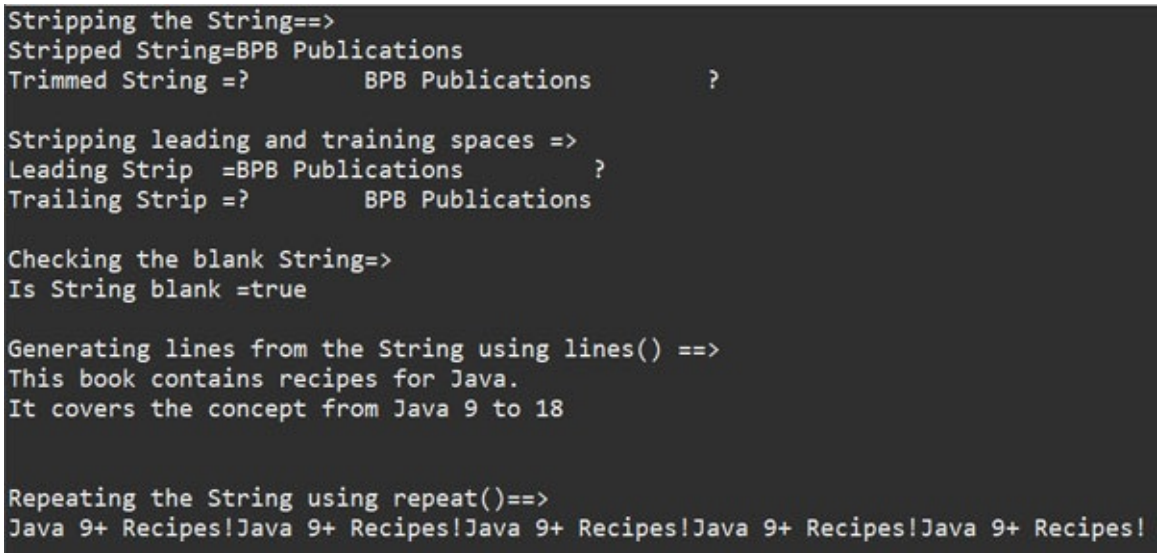


Figure 3.4: *Updated String API*

Explanation

As you have seen in the solution, there are a few methods added in the String API as a part of Java 11. Let us discuss each of these methods separately:

strip()

This method is used to strip off the leading training white spaces of the given string.

The syntax of the method is as follows:

```
public String strip()
```

The method returns:

- **`String`**, which represents the modified current string by removing all the leading and trailing white spaces.
- If the **`String`** is empty, the returning **`String`** is also empty.
- If all the characters of the **`String`** are white spaces, then it returns the empty **`String`**.

The earlier versions of Java had a similar method termed as **`trim()`** which was used to remove the leading and trailing white spaces from the String. But the **`strip()`** method is different from the **`trim()`** method.

Table 3.1 shows the differences between the **`trim()`** and **`strip()`** method:

trim()	strip()

Available from Java 1.0.	Available from Java 11.0.
It uses the codepoint (ASCII) value for identifying the white spaces.	It uses Unicode values for identifying the white spaces.
It removes characters with ASCII values less than or equal to ‘U+0020' or ‘32'.	It removes all the white spaces according to Unicode.

Table 3.1: *Difference between the trim() and strip() method*

The strip()method internally uses **isWhiteSpace(int codePoint)** to check whether the given character is white space or not. This method was launched in Java 5.0

According to the Java documentation, a character is a Java whitespace character if and only if it satisfies one of the following criteria:

- It is a Unicode space character (**SPACE_SEPARATOR**, **LINE_SEPARATOR**, or **PARAGRAPH_SEPARATOR**) but is not also a non-breaking space (‘**\u00A0**', ‘**\u2007**', ‘**\u202F**').
- It is ‘**\t**', **U+0009 HORIZONTAL TABULATION.**
- It is ‘**\n**', **U+000A LINE FEED.**
- It is ‘**\u000B**', **U+000B VERTICAL TABULATION.**
- It is ‘**\f**', **U+000C FORM FEED.**
- It is ‘**\r**', **U+000D CARRIAGE RETURN.**
- It is ‘**\u001C**', **U+001C FILE SEPARATOR.**
- It is ‘**\u001D**', **U+001D GROUP SEPARATOR.**
- It is ‘**\u001E**', **U+001E RECORD SEPARATOR.**
- It is ‘**\u001F**', **U+001F UNIT SEPARATOR.**

In our solution, that is, in **Listing 3-3**, we have used both the **strip()** as well as the **trim()** method on the same String to identify the behavior of both the methods.

Observe the following code:

```
String originalString="\u2001 \t BPB Publications \t \u2002 ";
System.out.println("Stripped String="+originalString.strip());
System.out.println("Trimmed String ="+
(originalString.trim())+"\n");
```

In this, our original String contains special characters like "**\u2001**" and "**\t**" at the beginning and ending. In the output, you will observe that we used the **strip()** method; both these special characters were considered as white space as

per the documentation, so they are removed. But that was not the case with the **`trim()`**. You will observe that the trim() method considers it as special characters and not the white space and keeps it as it is.

stripLeading()

As the method name indicates, this method strips the leading white spaces of the string.

The syntax of the method is as follows:

```
public String stripLeading()
```

The method returns:

- The current **`String`**, by removing all the leading white spaces.
- Empty **`String`**, if the string contains only white spaces.

In our solution, that is, **`Listing 3-3`**, line number 15, shows the result of invocation of this method.

```
System.out.println("Leading Strip  ="+
                originalString.stripLeading());
```

Similar to the **`strip()`** method, it considers "**`\u2001`**" and "**`\t`**" as white spaces and remove it.

stripTrailing()

This method is exactly similar to the earlier method, that is, **`stripLeading()`**. The only difference is it is used to remove the trailing white spaces.

The syntax of the method is as follows:

```
public String stripTrailing()
```

The method returns:

- current **`String`**, by removing all the training white spaces.
- empty **`String`**, if the string contains only white spaces.

In our solution, that is, **`Listing 3-3`**, line number10 shows the result of invocation of this method:

```
System.out.println("Leading
Strip  ="+originalString.stripTrailing());
```

Similar to the **`strip()`** method, it considers "**`\u2001`**" and "**`\t`**" as white spaces and remove it.

isBlank()

This method is used to check whether the **`String`** is empty or not.

The syntax of the method is as follows:

```
public boolean isBlank()
```

The method returns:

- **`True`**, if the string is empty or only contains white spaces.
- **`False`**, if the string contains any other characters than white spaces.

In our solution provided in **`Listing 3-3`**, line number 20, invokes this method as shown here:

```
String blankString="\u2001";
System.out.println("Is String blank ="+blankString.isBlank()+"\n");
```

This will return true as the **`blankString`** contains the special characters which are identified by white space according to the rules specified in the **`strip()`**.

lines()

This method returns the lines from the given String.

The syntax of method is as follows:

```
public Stream<String> lines()
```

The method returns stream of lines, which is generated from the given string.

The lines generated do not contain the line separator. The stream generated from this method is the collection of lines, which is ordered in the sequence of occurrence in the original string.

The **`lines()`** method uses the line terminator present within the given strings. The line terminator can be any of the following:

- "**`\n`**" **`(U+oooA)`**, a line feed character
- "**`\r`**" **`(U+oooD)`**, a carriage return character
- "**`\r\n`**" **`(U+oooD U+oooA)`**

In our solution, **`Listing 3-3`**, shows the invocation of this method on line number 23, as shown here:

```
String text="This book contains recipes for Java.\n
                  It covers the concept from Java 9 to 18";
text.lines().forEach((line)->System.out.println(line));
```

In this code snippet, the string text contains the line terminator **`\n`** because of which the generated stream will consist of two elements, each of that presents one line.



Though this method is introduced in Java 11, such a concept is not new to a Java developer. The split() method is already present in earlier versions of the String API. However, unlike the lines() method, this method returns the string array.

The lines() method is considered to be more faster than the split() method as it supplies the elements lazily by searching the new line terminator more speedily.

repeat()

This method is used to repeat the given string.

The syntax of method is as follows:

```
String repeat(int count)
```

The method returns:

- A **`String`**, repeated by a number of times provided in arguments.
- Empty **`String`**, if the string is empty or the argument, that is, the count is zero.

The method throws:

- **`IllegalArgumentException`**, if the method argument is negative.

In our solution in **`Listing 3-3`**, we have used this method in line number 28, as shown here:

```
String bookName="Java 9+ Recipes!";
System.out.println(bookName.repeat(5));
```

This will repeat the string **`bookName`** 5 times and return it.



The String class from Java is enhanced with some more methods to handle the white spaces in this version. The methods like strip(), stripLeading(), and stripTrailing() are used to remove the white spaces. These methods also consider some special characters as white spaces. Also, to check the empty string, you can now use the isBlank() method. The lines() method generates the lines based on the line separator. And, the repeat(int) method returns you the same line multiple times depending on the argument passed to it.

Updates in the reflective access of the nested class

There is one more addition in the working of inner classes in Java 11. Java 11

had introduced the new approach of reflection to access the private elements of the outer class.

Problem

We have a private element in our outer class. If we want to access the private element using the reflection, how can we achieve that? Does Java 11 provide any additional help for the same?

Solution

We will solve this problem statement by the code provided in **`Listing 3-4`**. We will also check what is the difference between Java 9 and Java 11 for such reflective access:

```
1. //Listing 3-4
2. package com.nestbasedaccess;
3.
4. import java.lang.reflect.Field;
5.
6. public class ReflectiveAccess_Outer {
7.   private static int value=١٠;
8.   public static class ReflectiveAccess_Inner{
9.    public static void changeIt() throws Exception {
10.     Field field=ReflectiveAccess_Outer.class
                                  .getDeclaredField("value");
11.     field.setInt(field, ٢٠);
12.     System.out.println("Modified Value ="+value);
13.    }
14.   }
15.   public static void main(String[] args) throws Exception {
16.    ReflectiveAccess_Inner.changeIt();
17.   }
18. }
```

Output

We need to compile this program by both pre-Java 11 compiler and Java 11 compiler. The following figures demonstrate the different output using different versions of Java.

Figure 3.5 shows the output after executing using the pre-Java 11 compiler:

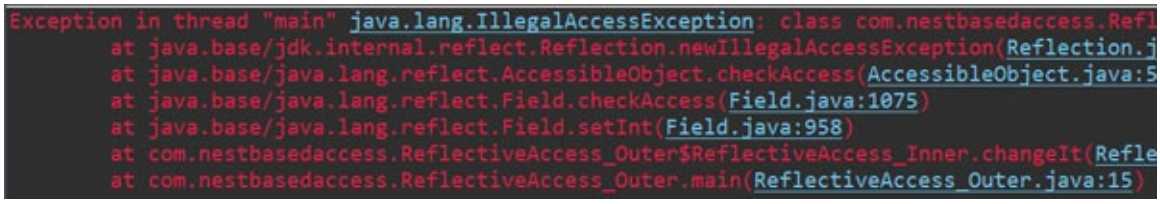


Figure 3.5: *Reflective Access Pre Java 11.0*

If you recompile and execute the same program using the Java 11 compiler, you will get the output as shown in *Figure 3.6*:

```
Modified Value =20
```

Figure 3.6: *Reflective Access Java 11.0*

Explanation

We all know now how the nest-based access is modified in Java 11. This modification also leads to one more feature which is related to reflective access of private members. Before Java 11, if we have any private members in the outer class, we were unable to access it to the reflection process. If you try to do so, the compiler generates **`IllegalAccessException`**.

In our solution **`Listing 3-4`**, we have a private static member value in our outer class. At line number 11, we are trying to access the same as, **`field.setInt(field, 20)`**.

If we compile this program with the pre-Java 11 compiler, it throws the **`IllegalAccessException`**. *Figure 3.5* shows the output of the same when we compile and execute the program by using Java 9 compiler. In case, we want to still access such private variables using reflection, we need to explicitly invoke **`setAccessible(true)`** on the private field as, **`field.setAccessible(true)`**.

Such explicit invocations are generally required because of the access bridge created by compiler for private members.

In Java 11, such an access bridge is not created for private data members. So, using the reflection API to access the private members from the outer class

becomes simpler and more straightforward. *Figure 3.6* shows the output of the same program executed by using Java 11 compiler. Observe the code that, we have not used any **`setAccessible(true)`** method in the code, still we can access the private member value by using the reflection API.



Handling private data members of the host class using reflection was a bit complex prior to Java 11. Earlier we were forced to invoke setAccessible(true) for the field before modifying the private element of the host or outer class. This approach had been changed in Java 11. There are no special methods added for this, but now we can access the private elements of the outer class directly by using the reflection API.

Local-variable syntax for lambda parameters

This update facilitates the use of the **`var`** keyword as a type of formal argument used in lambda expression.

Problem

Demonstrate how the var keyword can be used to declare the variables used in lambda expressions in Java 11.

Solution

This problem can be solved by the solution provided in the **`Listing 3-5`**, as follows:

```
1. //Listing 3-5
2. package com.var;
3.
4. import java.util.Arrays;
5. import java.util.List;
6.
7. @FunctionalInterface
8. interface Calculation{
9.   public abstract double calculateSum(List<Integer> numList);
10. }
```

```
11.
12. public class UseofVar{
13.   public static void main(String[] args) {
14.    // TODO Auto-generated method stub
15.    //using var keyword with consumer interface
16.    List<Integer> intList=Arrays.asList(١٠,٣,٥٤,١٢,٥٤,١٢);
17.    System.out.println("Elements in the list ==>");
18.    intList.forEach((var value)->System.out.print(value+" "));
19.
20.    //using var keyword with user defined functional interface
21.    Calculation calculation=(var list)->
22.       list.stream()
23.       .mapToInt(Integer::intValue)
24.       .sum();
25.    System.out.println("\nAddition of elements in list
                        ==>"+calculation.calculateSum(intList));
26.    }
27.   }
```

Output

The output is shown in *Figure 3.7*:

```
Elements in the list ==>10 3 54 12 54 12
Addition of elements in list ==>145.0
```

Figure 3.7: *The var keyword in lambda parameters*

Explanation

The var keyword was introduced in Java 10, which can be used as a type of local variable rather than an actual type. However, in Java 10, it was not possible to use this keyword in the lambda expression. Though we can completely skip the data type of the lambda, replacing it with the var keyword is not permitted.

For example, the following is the simple lambda expression:

```
(int value1, int value2) -> value1*value2;
```

We know that we can replace this as follows:

```
(value1, value2) -> value1*value2;
```

This is obviously possible right from Java 8. Let us now try to use the **`var`** keyword in the lambda expression declared previously. The statement becomes:

```
(var value1, var value2) -> value1*value2;
```

If we compile this by using Java 10, the compiler complains. But if you compile the same using Java 11, it compiles and executes.

The obvious question is what is the benefit of this approach to a developer? The main reason is uniformity. When you apply the var keyword to lambda, it creates uniformity between the lambda variables and local variables. Also, if we do not use any data type, then we cannot apply any modifiers to the variables. The following declaration shows how to apply other modifiers to the lambda expression parameters:

```
(@Nonnull var value1, @Nonnull var value2) -> value1*value2;
```

However, there are certain rules that must be followed:

- If we are using the var keyword within the lambda expression and the expression contains multiple variable, then you must use the var keyword for all the variables.

  For example, the following declaration is invalid:

  ```
  (var value1, value2) -> value1*value2;
  ```

- Mixing of the var keyword with normal data type is not permitted.

  For example, the following declaration is invalid:

  ```
  (var value1, int value2) -> value1*value2;
  ```

- While we use the var keyword in single parameterized lambda expression, we cannot omit the parenthesis unlike the normal lambda expression.

  For example, the following declaration is invalid:

  ```
  var value1 -> System.out.println(value1);
  ```

In our solution, provided in Listing on line number 18, we have used the **`var`** keyword in declaring the lambda expression for the Consumer interface as follows:

```
intList.forEach((var value)->System.out.print(value+" "));
```

Here, the data type of the value used in lambda is declared as var instead of Integer.

Also, we have declared our own functional interface Calculator which contains the abstract method **`calculateSum()`**. The argument of this method is **`List<Integer>`**. On line number 20, we have created a lambda expression for

this method as follows:

```
Calculation calculation=(var list)->
       list.stream()
          .mapToInt(Integer::intValue)
     .sum();
```

Here, the data type of the list is declared as a **`var`** instead of **`List<Integer>`**.



From Java 11, we can use the var keyword in the lambda expression parameters also. This adds more readability to lambda expressions.

HttpClient (Standard)

Java 11 provides the standardize **`HttpClient`** that implements HTTP/2 and WebSocket. This is a replacement of the existing legacy HttpURLConnection API.

Problem

How can we use the new HTTP client API to communicate with a web resource?

Solution

`Listing 3-6`, **`HttpClientDemo.java`** shows how to make the GET request to **http://www.google.com** and how to print the response information of the type header, status, and body content:

```
1. //Listing 3-6
2. package com.httpclient;
3.
4. import java.io.IOException;
5. import java.net.URI;
6. import java.net.http.HttpClient;
7. import java.net.http.HttpHeaders;
8. import java.net.http.HttpRequest;
9. import java.net.http.HttpResponse;
10.
11. public class HttpClientDemo {
```

```
12.    public static void main(String[] args) throws
                            IOException, InterruptedException {
13.
14.          HttpClient client = HttpClient.newHttpClient();
15.          HttpRequest request = HttpRequest.newBuilder()
16.                  .uri(URI.create("http://www.google.com"))
17.                  .GET()
18.                  .build();
19.         printHeader(client);
20.         printStatus(client, request);
21.         printBodyContent(client, request);
22.
23.      }
24.
25.    public static void printStatus(HttpClient client,
          HttpRequest request) throws IOException,
      InterruptedException {
26.     HttpResponse<Void> response = client.send(request,
27.                   HttpResponse.BodyHandlers.discarding());
28.     System.out.println("\n\nHTTP Status code ==> "+
                                    response.statusCode());
29.
30.      }
31.    public static void printHeader(HttpClient client)
32.                         throws IOException, InterruptedException
    {
33.     var request = HttpRequest.newBuilder()
                     .uri(URI.create("http://www.google.com"))
34.                                                 .method("HEAD",
    HttpRequest.BodyPublishers.noBody())
35.                  .build();
36.
37.     HttpResponse<Void> response = client.send(request,
```

```
                    HttpResponse.BodyHandlers.discarding());

    HttpHeaders headers = response.headers();
    System.out.println("Header Information ==>");
    headers.map().forEach((headerKey, headerValue) -> {
                        System.out.printf("%s: %s%n", headerKey,
headerValue);
      });

    }
    public static void printBodyContent(HttpClient client,
                                            HttpRequest request)
                throws IOException, InterruptedException {

          HttpResponse<String> response = client.send(request,
                  HttpResponse.BodyHandlers.ofString());
          System.out.println("\n\nBody Content ==>");
          System.out.println(response.body());
    }
}
```

Output

The following *Figure 3.8* shows the output generated when we give the GET request to **http://www.google.com**:



Figure 3.8: *Usage of HttpClient*

Explanation

In Java 9, this module was introduced as a part of JEP 11. This module was initially treated as an incubator module. The main objective of this module is to replace the legacy **`HttpURLConnection`** API.

`HttpURLConnection` API had few issues, which are listed below:

- This API is based on **`URLConnection`** which was designed for different protocols such as FTP, Gopher, etc. which are not of much use.
- It is based on HTTP/1.1 which is an older version in the market.
- Difficult to use and not well documented.
- Single threaded model

So, to overcome all these drawbacks or issues with **`HttpURLConnection`**, the line API is developed and released as a full-fledged API in Java 11.

The following are the major advantages of the **`HttpClient`** API:

- Easy to use.
- Exposes the relevant aspects of HTTP protocol by handling the process of request and response.
- Provides the standard authentication approach by implementing the basic

authentication.

- Supports HTTP/2.
- Performance is better than **`HttpURLConnection`**.

Let us now talk more about this class:

The class declaration is as follows:

```
public abstract class HttpClient extends Object
```

This class is used to send the request to the specified URL and retrieve the response. As this is an abstract class, the reference is created through the builder which is provided by the factory method **`newHttpClient()`**. This provides the information about the configuration and the shared resource for all the requests which are sent through it.

To send the request, we need to communicate with **`HttpRequest`**, which is an abstract class. The factory method **`newBuilder()`** `returns` **`HttpRequest.Builder`**, using which we can utilize different HTTP methods of HTTP while we communicate with the URL. This **`HttpRequest.Builder`** `class` provides different methods like:

- **`Uri(URI)`**
- **`GET()`**
- **`POST()`**
- **`DELETE()`**
- **`Version()`**
- **`build()`**

These are just some of the important methods from these classes. But as discussed earlier, it provides the support for all the methods of the HTTP protocol.

The following code snippet used in our solution shows how we can connect to **http://www.google.com** and generate **`HttpRequest`** for the same:

```
HttpClient client = HttpClient.newHttpClient();
HttpRequest request = HttpRequest.newBuilder()
                  .uri(URI.create("http://www.google.com"))
                  .GET()
                  .build();
```

Remember that we need to send the request explicitly to generate any response from this request, which is achieved by the **`send()`** method of **`HttpClient`**. We must provide the **`BodyHandler`** object for each request that we sent. This

`BodyHandler` decides how to handle the response body.

There are different methods used to send the request:

1. **`send(HttpRequest, BodyHandler)`**

   This is a blocking approach. It blocks the next request until the current request is sent and the response is received by the client.

2. **`sendAsync(HttpRequest, BodyHandler)`**

   This sends the request to the resource and receives the response asynchronously. This returns the `CompletetableFuture<HttpResponse>`. This completes when the response is ready and available.

   In our solution, **`Listing 3-6`**, we have used different methods of **`HttpResponse`** to utilize the response.

3. To receive the status code:

```
HttpResponse<Void> response = client.send(request,
      HttpResponse.BodyHandlers.discarding());
System.out.println("\n\nHTTP Status code ==> "+
response.statusCode());
```

4. To receive the header information:

```
HttpResponse<Void> response = client.send(request,
                 HttpResponse.BodyHandlers.discarding());
HttpHeaders headers = response.headers();
```

5. To receive the body content:

```
HttpResponse<String> response = client.send(request,
                HttpResponse.BodyHandlers.ofString());
System.out.println("\n\nBody Content ==>");
System.out.println(response.body());
```



The new HTTPClient class provided by Java 11 facilitates uniform handling of web requests. This HTTPClient can send the request to the resource which is wrapped in HttpRequest object. This HTTPRequest object contains information about whether you want to send the request via the get() or post() method and to which the resource HttpClient wants to communicate. The response generated by the resource can be stored in the HttpResponse object.

Launch single-file source-code program

Many of the programming language provides a facility to compile and execute the programs in a single step. Though as a developer you might have used a similar feature while you use the IDE. In the IDE like eclipse, you execute your

program without compiling it. At least, you get a feel of it. But internally, the compilation is always there. So, this was the feature introduced by IDE and not by the language itself. From Java 11, a similar concept is launched as a part of Java feature.

Problem

How the single-file source-code program concept works in Java 11?

Solution

We will discuss this with very simple program of printing "**`Hello World`**". Instead of your IDE, write the following code in the editor like notepad or VSCode, and so on:

```
1. //Listing 3-7
2. public class HelloWorld{
3.     public static void main(String[] args){
4.         System.out.println("Hello World");
5.     }
6. }
```

Output

We will not follow the traditional practice of compiling and then executing the class file. Instead, we will just execute the file by using the Java command to get the output.

The following output as shown in *Figure 3.9* would be generated as follows:

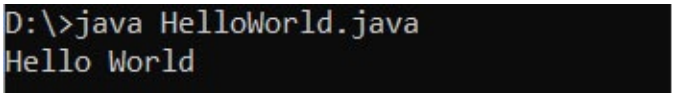


Figure 3.9: *Single-file source code*

Explanation

A single file source code is an interesting feature launched by Java 11. Earlier, prior to execute any file in Java, we needed to compile it. But if want to learn some new features or want to test a small piece of a code, then this process looks bit tedious. Using this new feature now, we can bypass the compilation process

and execute the file directly.

To work with this, we have to just write a Java file with the class having the **`main()`** function in it. Traditionally (till Java 10), there are modes of execution of the Java program:

1. Running a class provided to JRE. (The class must contain the **`main()`** function.)
2. Running the main class of a JAR file, which is declared in the manifest file.
3. Running the main class of the module (if working with Java 9 module system).
4. Now from Java 11, the fourth mode is added.
5. Running the class which is declared in the source file provided to JRE.

In this mode, the source file is compiled to the class file internally and stored in the memory temporarily. At the time of execution, this file is scanned and the first class with the main is executed.

For example, consider the following code snippet for a **`sample.Java`**:

```
class First{
  public static void main(String[] args){
     System.out.println("In first");
  }
}
class Second{
  public static void main(String[] args){
    System.out.println("In Second");
  }
}
```

If you execute the file, you will get the output as "**`In first`**". The file contains two classes but as per the sequence class, the first is executed. Now, change the sequence of the file as shown below:

```
class Second{
  public static void main(String[] args){
    System.out.println("In Second");
  }
}
class First{
  public static void main(String[] args){
     System.out.println("In first");
  }
}
```

Now, if you execute the same file again, you will get the output as "**`In second`**" as the sequence is changed. You can also pass the command line arguments which can be passed to the **`main()`** function.



The single-file source code execution is a new feature introduced in Java 11. This feature enables the developers to execute the Java program without compiling it. If the source file contains more than one class, then the first top-level class in the source file is executed. The class to be executed must contain the public static void main(String[] args) method. The classes are internally compiled and loaded by the custom class loader. Lastly, the compiled classes are executed as a part of unnamed module. Arguments that are added after the file name are passed as an String array arguments to the main() method.

Conclusion

In this chapter, we discussed different solutions to solve the problem statements which are based on the concepts launched in Java 11. We discussed how the updated reflection API can be used to retrieve the information about the host or other nest members. Also, we studied new methods launched in the String API which are used to handle the white spaces more effectively. Further, we talked about how we can access the private member of the nested class by using reflection, use of the var keyword for lambda parameters, and the **`HTTPClient`** class. Lastly, we discussed how we can execute the Java file directly without compiling it.

In the next chapter, we will discuss and implement different features introduced in Java 12. We will create different recipes to understand the complex APIs like **`CompletableFuture`**, **`TeeingCollectors`**, and so on.

Questions

1. Which method is used get the hostname of the current nest class?
2. What is the way to find out the nest members of the current class?
3. How to generate multiple lines from the existing String?
4. List different rules to use the var keyword in lambda parameters.
5. How can you send the asynchronous request by using the new HttpClient?

Keywords

- getNestHost()
- getNestMembers()
- isNestmateOf()
- strip()
- stripLeading()
- stripTrailing()
- isBlank()
- lines()
- repeat()
- HttpClient

CHAPTER 4

Java 12 – Performance is the Key

Introduction

In the last chapter, we discussed the different features introduced in Java 11. As we already know that the Java 11 version was launched in September 2018. As per the policy decided by Oracle, the new version is launched after every six months. Under this time-based version releasing scheme, Java 12 was launched in March 2019, which was exactly after six months after the release of Java 11. This version mainly focuses on some of the features which focus on performance. Of course, apart from this, some other API developments are introduced in this version. The garbage collector can now return the unused memory because the pause time is reduced to improve performance. Along with this, Java 12 also provides the microbenchmark to monitor your application's speed or even the isolated unit of your application.

There are also a few modifications in the String API as the Java API developer knows that the String API is one of the most important APIs for application developers. Java 12 is said to be a rapid version release. This means that there is no **Long-Term Support** (**LTS**) for this version unlike Java 11. This means that Java 12 comes with commercial support until the next version appears in the market.

Structure

In this chapter, we will cover the following concepts:

- Teeing Collectors in the Stream API
- Updates in String API
- Updates in NIO
- Updates in CompletableFuture
- Compact Number Formatting

Objectives

In this chapter, we will walk you through the different features introduced in Java 12.0. We will discuss how these new API changes are useful for developers to enhance coding practices. We will learn these API changes through different examples.

Installation

Visit the following URL to download Java 12 to follow all the code snippets in this chapter:

https://www.oracle.com/java/technologies/javase/jdk12-archive-downloads.html

Collectors.teeing()

Java 12 introduced a static method in `Collectors` as **`Collectors.teeing()`**. Collectors is one of the interfaces provided in the stream API. **`Collectors.teeing()`** is used to collect the generated result from independent collectors.

Problem

Utilize the concept of **`Collectors.teeing()`** to store the maximum and minimum salary of the list of Employees.

Solution

Let us consider **`Employee.java`** as shown in **`Listing 4-1:`**

```
1. //Listing 4-1
2. package com.streams;
3.
4. public class Employee {
5.   private int empId;
6.   private String empName;
7.   private double empSalary;
8.   public Employee(int empId, String empName, double empSalary)
```

```
{
    super();
    this.empId = empId;
    this.empName = empName;
    this.empSalary = empSalary;
  }
  public int getEmpId() {
    return empId;
  }
  public void setEmpId(int empId) {
    this.empId = empId;
  }
  public String getEmpName() {
    return empName;
  }
  public void setEmpName(String empName) {
    this.empName = empName;
  }
  public double getEmpSalary() {
    return empSalary;
  }
  public void setEmpSalary(double empSalary) {
    this.empSalary = empSalary;
  }
}
```

We will create the list of Employees and pass it to the stream. Later, we will invoke the **`Collectors.teeing()`** to generate and group the average salary and count of Employees as shown in **`Listing 4-2, CollectorTeeingDemo.java`**:

```
//Listing 4-2
package com.streams;

```

```
import java.util.Arrays;
import java.util.HashMap;
import java.util.List;
import java.util.Map;
import java.util.stream.Collectors;

public class CollectorTeeingDemo {
  public static void main(String[] args) {
 // TODO Auto-generated method stub
   List<Employee> empList=Arrays.asList(
      new Employee(١٠١,"John",٥٠٤٩),
      new Employee(١٠٢,"Smith",٧٠٠٠),
      new Employee(١٠٣,"William",٩٠٠٠),
      new Employee(١٠٤,"Derek",٢٠٠٠),
      new Employee(١٠٥,"Steve",٥٠٠٠)
     );

             HashMap    <String,    Double>    empSummary    =
empList.stream().collect(
     Collectors.teeing(
       Collectors.averagingDouble(Employee::getEmpSalary),
         Collectors.counting(),(e1, e2) -> {
           HashMap <String,Double> map =
                                new HashMap <String,Double> ();
                        map.put("Total  Number  of  Employees,",
e2.doubleValue());
                       map.put("Average  Salary  of  Employees,",
e1.doubleValue());
                      return map;
          }
     ));
```

```
32.
33.             for    (Map.Entry<String,    Double>    entry    :
    empSummary.entrySet()) {
34.     System.out.println("Key : " + entry.getKey() + " Value : "
    +
                                        entry.getValue().intValue());
35.    }
36.   }
37. }
```

Output

If we execute **`CollectorTeeingDemo.java`**, we will get the output as shown in *Figure 4.1*:

```
Key : Total Number of Employees,  Value : 5
Key : Average Salary of Employees,  Value : 5609
```

Figure 4.1: *Collector Teeing*

Explanation

To generate a resulting value from the stream, normally we use terminal operations of the stream. We can also use the **`collect()`** method to collect all the processed streams into a collection. Taking one step forward, Java 12 facilitates to perform the different operations on the collection and group of the results.

The syntax of the method is as follows:

```
public static <T,R1,R2,R> Collector<T,?,R> teeing(
       Collector<? super T,?,R1> result1,
       Collector<? super T,?,R2> result2,
       BiFunction<? super R1,? super R2,R> resultmerger)
```

Type parameters in the method:

- **`T`**: The type of the input elements.
- **`R1`**: The result type of the first collector.
- **`R2`**: The result type of the second collector.
- **`R`**: The final result type.

Method parameters are:

- **`result1`**: The result generated from the first operation performed on the collection.
- **`result2`**: The result generated from the second operation performed on the collection.
- **`resultmerger`**: The bi-function which merges two results generated from two operations into the single unit.

The method returns **`Collector`**, which combines the results of two supplied collectors.

In our solution **`Listing 4-2`**, we have used this feature as shown as follows:

```
HashMap <String, Double> empSummary = empList.stream().collect(
                  Collectors.teeing(

  Collectors.averagingDouble(Employee::getEmpSalary),
                   Collectors.counting(),
                      (e1, e2) -> {
                          HashMap <String,
                          Double> map = new HashMap <String,
                            Double> ();
```

We have collected the stream of employees into a collection by using the **`collect()`** method. Then, we have applied **`Collectors.teeing()`** which performs two operations. First, it calculates the average of employees by invoking **`Collectors.averagingDouble()`** and then, it also counts the number of elements in the given collector by invoking **`Collectors.counting()`**. These two results are stored in a **`HashMap`** by passing them into the Bi-Function which is provided as the third argument with the help of the lambda expression in the method. As we have stored the results in the **`Map`**, the result is passed to the **`HashMap`**.



The Collectors.teeing() method is used to group the results from different operations that are performed on the collection.

String API updates

The String API is already rich in Java. Though there are no major changes in this API, Java 11 introduced some additional methods to enhance the white space handling in Strings. Along with this, it also provides some more methods for processing the Strings.

Following are the different methods introduced in String:

- `transform()`
- `indent()`
- `describeConstable()`
- `resolveConstantDesc()`

Problem

Which new features or functionalities are introduced in Java 12 to improve the String handling?

Solution

`Listing 4-3, StringDemos.Java` shows different methods of the String API that are part of Java 12:

```
1. //Listing 4-3
2. package com.strings;
3.
4. import java.lang.invoke.MethodHandles;
5. import java.util.Arrays;
6. import java.util.List;
7. import java.util.Optional;
8.
9. public class StringDemos {
10.
11.   public static void main(String[] args) {
12.    // TODO Auto-generated method stub
13.    indentDemo();
14.    transformDemo();
15.    describeConstableDemo();
16.    resolveConstantDescDemo();
17.
18.   }
19.
```

```
20.   private static void describeConstableDemo() {
21.    // TODO Auto-generated method stub
22.    String course="Java١٢";
23.    Optional<String> data=course.describeConstable();
24.    System.out.println();
25.              System.out.println("Using    describeConstable()==>
"+data.get());
26.   }
27.
28.   private static void resolveConstantDescDemo() {
29.    // TODO Auto-generated method stub
30.    String course="java١٢";
31.                                                          String
result=course.resolveConstantDesc(MethodHandles.lookup());
32.           System.out.println("Using   resolveConstantDesc()==>
"+result);
33.   }
34.
35.   private static void transformDemo() {
36.    // TODO Auto-generated method stub
37.    System.out.println("Using transform() ==>");
38.    String str="Java9,Java10";
39.    List<String> strList=str.transform(str1->
40.                                                        {return
Arrays.asList(str1.split(","));});
41.    System.out.println("Original String==> "+str);
42.    System.out.println("Transformed String Content==> ");
43.    strList.forEach(str2->System.out.println(str2));
44.
45.    String str١ = "Life is beautiful,";
46.     var strresult = str١.transform(input -> input.concat(" have
fun"))
```

```
                            .transform(String::toUpperCase);
        System.out.print("Chaining of transform()==> ");
        System.out.print(strresult);
    }

    private static void indentDemo() {
     // TODO Auto-generated method stub
     System.out.println("Using indent() ==>");
     System.out.println("-------------------");
     String str="  Welcome to Java\nIt's simple\nIt's robust";
     System.out.println("using indent(0):");
     String str1=str.indent(0);
     System.out.println(str1);

     System.out.println("using indent(2):");
     str1=str.indent(2);
     System.out.println(str1);

     System.out.println("using indent(-2):");
     str1=str.indent(-2);
     System.out.println(str1);
    }
}
```

Output

The output shown in *Figure 4.2* will be displayed if we execute **`StringDemos.java`**:

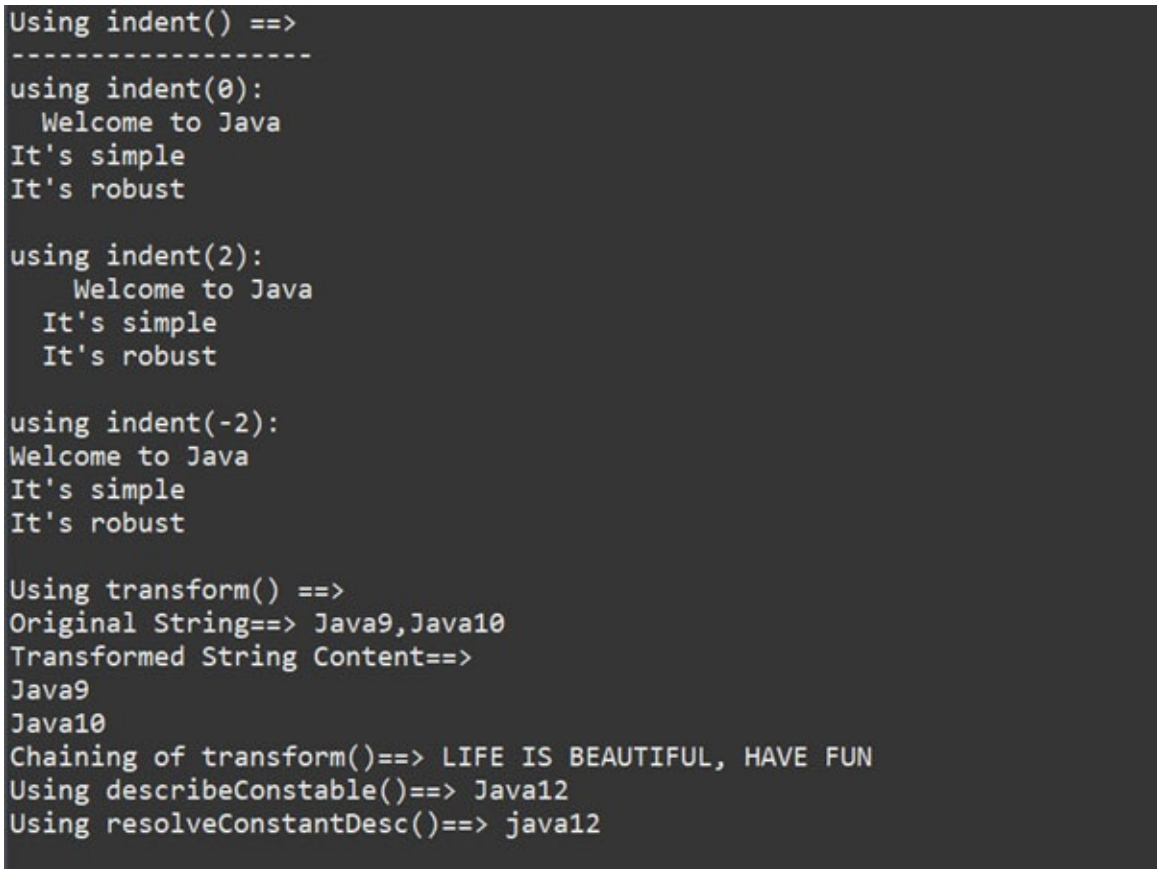


Figure 4.2: *String updates*

Explanation

As seen in the solution, there are a few methods added in the String API as a part of Java 12. Let us discuss each of these methods separately:

indent()

This method of **`String`** is used to adjust the indentation of the String.

The syntax of the method is as follows:

```
public String indent(int i)
```

The method parameters are:

- **i:** number of leading white space characters to add or remove

The method returns **`String`**, with indentation adjusted and line endings normalized

When we are presenting any outputs in the form of a String, the indentation becomes really important. However, handling the indentation is tricky. Because

if you try to handle it manually, it might not work in the same way as it is intended. To achieve this, **`String`** now comes up with a method **`indent()`**. This method adjusts the indentation of the **`String`** as per the value provided to it in the argument.

The indentation is applied according to the following criteria:

- **`i>0`**, then the i spaces are inserted at the beginning of the String.
- **`i=0`**, then the method has no impact on the String.
- **`i<0`**, then the i white spaces are removed from the beginning of the String.

In our solution **`Listing 4-3`**, at line number 56, we have declared a String as follows:

```
String str="  Welcome to Java\nIt's simple\nIt's robust";
```

As you can observe, there are white spaces at the beginning of this entire string. As we have also used the special character \n, the indent method internally uses the **`lines()`** method. So, when we used, **`str.indent(0)`**, the output does not remove any white spaces preceding the string, though it prints the output in three lines because of the line separator character.

But when used **`str.indent(2)`**, the output generated shows that two white spaces from the beginning are removed for the first line, as there are no white spaces for the next two lines where there is no impact. Also, when we used **`str.indent(-2)`**, you will observe that for every line two white spaces are added at the beginning.

transform ()

As the method name indicates, this method is used to transform the given string to some other format.

The syntax of the method is as follows:

```
public <R> R transform(Function<? super String,? extends R> fn)
```

The type parameters of the method:

- **`R`**: class of the result.

The method parameters are:

- **`fn`**: functional interface to a apply.

The method returns result of the applied function to this **`String`**.

This method is used to apply some function to the given string. The provided

function accepts a single string and produces some result based on the logic applied to it. If any exception is generated during this process, then the exception is propagated back to the invoker.

At line number 39, of our solution in **Listing 4-3**, we have used the simplest form of this function as follows:

```
List<String> strList=str.transform(
    str1->{return Arrays.asList(str1.split(","));})
```

The **transform()** method takes the **String str** and converts it to List by applying the **split()** method as shown earlier. This will generate the **List** as **["Java9", "Java10"]**.

We can also chain the **transform()** function, as shown in line number 44, of our solution as follows:

```
String str1 = "Life is beautiful,";
var strresult = str1.transform(input -> input.concat(" have fun"))
                    .transform(String::toUpperCase);
```

Here, we have passed the original **String str** to the first **transform()** function and concatenated " **have fun**" to it. Then, the modified string, "**Life is beautiful, have fun**" is passed to another **transform()** function which is chained to the existing function and converts the string to uppercase giving the result as "**LIFE IS BEAUTIFUL, HAVE FUN**".

describeConstable()

The **String** class is modified in Java 12. Now, it also implements the **Constable** and **ConstantDesc** interfaces as follows:

```
public final class String extends Object implements Serializable,
        Comparable<String>, CharSequence, Constable, ConstantDesc
```

As it implements the **Constable** interface, it overrides the **describeConstable()** method as a part of its API.

The syntax of the method is as follows:

```
public Optional<String> describeConstable()
```

The method returns an **Optional,** describing the **String** instance.

To understand how this method works, we need to know about the **Constable** interface. Any **Constable** type is the one, who values are considered to be final. Such values can be stored in the constant pool. Now as **String** is immutable, String is converted as a type of **Constable** from Java 12. When the **Constables** are stored in the constant pool, the nominal description of **Constable** is attached to it, which can be retrieved by the **describeConstable()** method. **String** serves as its own nominal descriptor.

In our solution **`Listing 4-3`**, we have invoked this method on line number 23 as follows:

```
Optional<String> data=course.describeConstable();
```

This returns the same **`String`** value, that is, **`course`**, as the nominal description of the **`Constable`**.

resolveConstantDesc()

We already know that the **`String`** in Java 12 implements **`ConstantDesc`** along with some other interfaces. The **`resolveConstantDesc()`** method is from this interface, which is implemented by **`String`**.

The syntax of the method is as follows:

```
public String resolveConstantDesc(MethodHandles.Lookup lookup)
```

The method returns the **`String`** `instance`.

This method resolves the instance as **`ConstableDesc`**. The result of this invocation is the same instance on which the method is applied.

In our solution **`Listing 4-3`**, we have invoked this method on line number 31 as follows:

```
String result=course.resolveConstantDesc(MethodHandles.lookup());
```

This returns the instance of String as shown in the preceding solution.



The String class is updated with few more methods in Java 12. The following are the new methods introduced and their usage:

- **indent():** This is used to adjust the indentation of the given String. The method takes the positive or negative value as a parameter. The positive value removes the extra white spaces from the beginning and the negative value adds the extra white spaces at the beginning of the given String.
- **transform():** This is used to transform the given String. It takes the functional interface to process the given String and returns the modified String. We can also chain the transform() to perform multiple operations sequentially on the given String.
- **describeConstable():** This returns the minimal description of the Constable as optional.
- **resolveConstantDesc():** This returns the instance of the Constable type on which the method is invoked.

Updates in NIO

Java 12 has introduced a new feature in the file API to check the equality of the files. Now, you can invoke the **`Files.mismatch()`** method to check whether the files are equal or not.

Problem

We have two files, **`first.txt`** and **`second.txt`**. How do we check whether the files are equal or not using the new API introduced in Java 12?

Solution

`Listing 4-4, FileMismatchDemo.java` provides a solution to the given problem as follows:

```
1. //Listing 4-4
2. package com.files;
3.
4. import java.io.IOException;
5. import java.nio.file.Files;
6. import java.nio.file.Path;
7. import java.nio.file.Paths;
8. import java.util.stream.Collectors;
9.
10. public class FileMismatchDemo {
11.
12.   public static void main(String[] args) throws IOException {
13.    // TODO Auto-generated method stub
14.    System.out.println("Working with Files.mismatch()");
15.    Path path١ = Paths.get("first.txt");
16.    Path path٢ = Paths.get("second.txt");
17.    Path path٣ = Paths.get("third.txt");
18.    System.out.println("Comparing identical files:");
19.    long result١ =Files.mismatch(path1, path2);
```

```
    System.out.println("Result 1==> "+result1);
    if(result١-==١)
            System.out.println("first.txt  and  second.txt  are
matching");
    else
         System.out.println("first.txt  and  third.txt  are  not
matching");
    System.out.println("Comparing non-identical files:");
    long result٢ =Files.mismatch(path1, path3);
    System.out.println("Result 2==> "+result2);
    if(result١-==٢)
            System.out.println("first.txt  and  second.txt  are
matching");
    else
    {
        System.out.println("first.txt  and  second.txt  are  not
matching");
                                  String       content       =
Files.lines(path٣).collect(Collectors.joining());
     System.out.println("The first mismatched character at
                                    index   number   "+result2+"
:==>"+content.charAt((int)result2));
    }
   }
}
```

Output

The output shown in *Figure 4.3* is observed if we execute **`FileMismatchDemo.java`**:

```
Working with Files.mismatch()
Comparing identical files:
Result 1==> -1
first.txt and second.txt are matching
Comparing non-identical files:
Result 2==> 11
first.txt and second.txt are not matching
The first mismatched character at index number 11 :==>j
```

Figure 4.3: *Files.mismatch()*

Explanation

Checking the equality of the files is a little bit complex traditionally in Java. You probably need to write the iteration to check the individual character of the file to identify the equality of the contents.

This comparison of the files had been made relatively simpler by the new method **`mismatch()`** in Java 12.

The syntax of the method is as follows:

```
public static long mismatch(Path path١, Path path٢) throws
IOException
```

The method parameters are:

- **`path1`**: The path to the first file.
- **`path2`**: The path to the second file.

The method returns:

- First mismatch position between the files.
- -1L, it no mismatch found.

The files are considered to be matched if, any of the following condition is matched:

- The path provided for both the files is the same, even if the files do not exist.
- Both the files are of the same size and every byte of each of them is identical with each other.

If any of the conditions do not match, then the files are said to be mismatched and the method will return either of the following values:

- The first position of the mismatched byte, between the files.
- The size of the smaller file in bytes, if the files are of different sizes. Of course, the bytes of the smaller files must be matched with the bytes located at the same position in the other file.

In our solution provided in **`Listing 4-4`**, we have three files **`first.txt`**, **`second.txt`**, and **`third.txt`**. The files **`first.txt`** and **`second.txt`** are identical to each other and **`first.txt`** and **`third.txt`** are non-identical files.

So, if we invoke **`Files.mismatch(first.txt, second.txt)`**, it returns `-1` as shown in the output as both the files are identical.

But if we invoke **`Files.mismatch(first.txt, third.txt)`**, it returns `11`, which is the index number of the first occurrence of the byte which is not matching.



The new java.nio.file.Files class now has a new static method mismatch() in Java 12 to compare to files. This method compares the two files; the path of which is provided as arguments to this method. The method returns -1 if the files match else it returns the index number of the first mismatched byte.

Updates in the CompletableFuture interface

Most widely improved interface from Java 9 onwards is **`CompletableFuture`**. The journey is still on to make the interface more flexible. Even in Java 12 few more methods are added like **`exceptionallyAsync()`** and **`exceptionallyComposeAsync()`**.

Problem

What are the different ways to handle the exceptions generated while dealing with **`CompletableFuture`**?

Solution

We will learn the different approaches of dealing with exception by different solutions.

Using exceptionallyAsyc() without executor

The **`Listing 4-6, ExceptionallyAsync.java`** shows how to work with the new method **`exceptionallyAsyc()`** while dealing with **`CompletableFuture`**:

```
1. //Listing 4-6
2. package com.completablefuture;
3. 
4. import java.util.concurrent.CompletableFuture;
5. 
6. public class ExceptionallyAsync {
7.    public void doSomeTask(int value) {
8.    System.out.println("Working with exceptionallyAsync without
                                          Executor==>");
9.    int data=\··;
10.   CompletableFuture.supplyAsync(() -> {
11.    System.out.println("supplyAsync() executed by:" +
                Thread.currentThread().getName());
12.    int ans = data / value;
13.    return ans;
14.   }).thenAcceptAsync(s -> {
15.    System.out.println("acceptAsync() executed by:" +
              Thread.currentThread().getName());
16.    System.out.println("Answer from AcceptAsync()=>" + s);
17.   }).exceptionallyAsync(e -> {
18.    System.out.println("exceptionallyAsync executed by::" +
              Thread.currentThread().getName());
19.    System.out.println("Exception occured :" + e.getMessage());
20.    return null;
21.   });
22.  }
23. }
```

In our repository, **`Listing 4-5`**, **`CompletableFutureDemo.java`** is an application file with the **`main()`** function. Invoke the **`doSometask()`** method from **`ExceptionallyAsync.java`** by passing such a value, so that no exception occurs as shown:

```
System.out.println("With Exception ==>");
System.out.println("main() executed by:" + Thread.currentThread().
       getName());
```

```
ExceptionallyAsync async=new ExceptionallyAsync();
async.doSomeTask(10);
```

Output

When you execute this, you will get the output as shown in *Figure 4.4*:

```
Without Exception ==>
main() executed by:main
Working with exceptionallyAsync without Executor==>
supplyAsync() executed by:ForkJoinPool.commonPool-worker-3
acceptAsync() executed by:ForkJoinPool.commonPool-worker-3
Answer from AcceptAsync()=>10
```

Figure 4.4: *exceptionallyAsync without Executor no Exception*

Now, pass a value to the same method which generates the exception as follows:

```
ExceptionallyAsync async=new ExceptionallyAsync();
async.doSomeTask(0);
```

When you execute this, you will get the output as shown in *Figure 4.5*:

```
With Exception ==>
main() executed by:main
Working with exceptionallyAsync without Executor==>
supplyAsync() executed by:ForkJoinPool.commonPool-worker-3
exceptionallyAsync executed by::ForkJoinPool.commonPool-worker-3
Exception occured :java.lang.ArithmeticException: / by zero
```

Figure 4.5: *exceptionallyAsync without Executor with Exception*

Explanation

Java 12 added few more methods to handle the exceptions arising while dealing with **`CompletableFuture`** asynchronously. One of such method is **`exceptionallyAsync()`**.

The syntax of the method is as follows:

```
default CompletionStage<T> exceptionallyAsync
      (Function<Throwable,? extends T> function1)
```

The method parameters are:

- **`function1`**: The function is used to compute the value of the returned **`CompletionStage`**, if this **`CompletionStage`** is completed `exceptionally`.

The method returns new **`CompletionStage.`**

In the solution which is provided in the Listing, you can observe that when the exception is generated, the **`exceptionallyAsync()`** method is invoked. You can also observe that it uses the same worker thread which is associated with **`CompletableFuture`**.

Using exceptionallyAsyc() with executor

`Listing 4-7, ExceptionallyAsyncWithExcecutor.java` shows how to add the executor to the **`exceptionallyAsync()`** method:

```
1. //Listing 4-7
2. package com.completablefuture;
3.
4. import java.util.concurrent.CompletableFuture;
5. import java.util.concurrent.ExecutorService;
6. import java.util.concurrent.Executors;
7.
8. public class ExceptionallyAsyncWithExecutor {
9.   public void doSomeTask(int value) {
10.    System.out.println("Working with exceptionallyAsync with
          Executor==>");
11.    ExecutorService executor = Executors.newFixedThreadPool(10);
12.    int data=\··;
13.    CompletableFuture.supplyAsync(() -> {
14.     System.out.println("supplyAsync() executed by:" +
          Thread.currentThread().getName());
15.     int ans = data / value;
16.     return ans;
17.     }).thenAcceptAsync(s -> {
18.     System.out.println("acceptAsync() executed by:" +
              Thread.currentThread().getName());
19.     System.out.println("Answer from AcceptAsync()=>" + s);
20.    }).exceptionallyAsync(e -> {
21.     System.out.println("exceptionallyAsync executed by::" +
             Thread.currentThread().getName());
22.     System.out.println("Exception occured :" + e.getMessage());
```

```
23.      return null; }, executor).thenApply(s -> {
24.        System.out.println("thenApply() executed by:" +
                   Thread.currentThread().getName());
25.        return 0;
26.      }).thenAccept(s -> {
27.      System.out.println("thenAccept() executed by:" +
               Thread.currentThread().getName());
28.      System.out.println("Generated answer:" + s);
29.     });
30.    }
31. }
```

We will test this code with the exception because the executor will work only when the exception occurs.

So, we need to add the following lines to our application class:

```
ExceptionallyAsyncWithExecutor async=
                    new ExceptionallyAsyncWithExecutor();
async.doSomeTask(0);
```

Output

If you execute this block, you will get the output as shown in *Figure 4.6*:

```
Working with exceptionallyAsync with Executor==>
supplyAsync() executed by:ForkJoinPool.commonPool-worker-3
exceptionallyAsync executed by::pool-1-thread-1
Exception occured :java.lang.ArithmeticException: / by zero
thenApply() executed by:main
thenAccept() executed by:main
Generated answer:0
```

Figure 4.6: *exceptionallyAsync with Executor with Exception*

Explanation

We have seen the **`exceptionallyAsync()`** method already in our earlier demonstration. But that method does not utilize any executor if the exception occurs. Our solution in the Listing is the demonstration of the **`exceptionallyAsync()`** method which takes the executor as another argument.

The syntax of the method is as follows:

```
default CompletionStage<T> exceptionallyAsync
            (Function<Throwable,? extends T> fn1, Executor
executor\)
```

The method parameters are:

- **`fn1`**: The function is used to compute the value of the returned **`CompletionStage`**, if this **`CompletionStage`** is completed `exceptionally.`
- **`executor1`**: The executor is used for asynchronous execution.

The method returns new **`CompletionStage`**

In our solution **`Listing 4-7`**, you can observe that once the exceptions occurs, the task is executed by another executor which is provided in the **`exceptionallyAsync()`** method, which runs in a different thread. If there is no occurrence of any exception, the method returns normally.

`Using exceptionallyComposeAsync()`

Finally, we will work with the **`exceptionallyComposeAsnyc()`** method, which is shown in the **`Listing 4-8`**, **`ExceptionallyComposeAsync.java`**:

```
1. //Listing 4-8
2. package com.completablefuture;
3. import java.util.concurrent.CompletableFuture;
4.
5. public class ExceptionallyComposeAsync {
6.   public void doSomeTask(int value) {
7.                         System.out.println("Working        with
   exceptionallyComposeAsync()==>");
8.    int data=\··;
9.    CompletableFuture<Void> task3 =
            CompletableFuture.supplyAsync(() -> {
10.     System.out.println("supplyAsync() executed by:" +
              Thread.currentThread().getName());
11.     int ans = data / value;
12.     return ans;
13.   }).exceptionallyComposeAsync(
            e -> CompletableFuture.supplyAsync(() -> {
14.     System.out.println("Exception occured Propagating to
```

```
                    supplier with default value!!");
15.      return data;
16.     })).thenAcceptAsync(s -> {
17.      System.out.println("Generated value:"+s);
18.      return;
19.
20.     });
21.    }
22. }
```

Add the following lines to your application class:

```
ExceptionallyComposeAsync async=new ExceptionallyComposeAsync();
async.doSomeTask(0);
```

Output

After the execution, you will get the output as shown in *Figure 4.7*:

```
Working with exceptionallyComposeAsync()==>
supplyAsync() executed by:ForkJoinPool.commonPool-worker-3
Exception occured Propagating to supplier with default value!!
Generated value:100
```

Figure 4.7: *exceptionallyComposeAsync() with Exception*

Explanation

Another method introduced in Java 12 to deal with the exception in **`CompletableFuture`** is **`exceptionallyComposeAsync()`**. This method takes two overloaded forms, without the executor and with the executor.

Form1: without executor

The syntax of the method is as follows:

```
default CompletionStage<T> exceptionallyComposeAsync(
       Function<Throwable,? extends CompletionStage<T>> fn)
```

The method parameters:

- **`fn`**: The function is used to compute the returned **`CompletionStage`** if this **`CompletionStage`** is completed exceptionally.

The method returns the new **`CompletionStage`**

Form2: with executor

The syntax of method is as follows:

```
default CompletionStage<T>
exceptionallyComposeAsync(Function<Throwable,
     ? extends CompletionStage<T>> fn, Executor executor)
```

The method parameters:

- **`fn`**: The function is used to compute the returned **`CompletionStage`** if this **`CompletionStage`** `is completed exceptionally.`
- **`executor`**: The executor is used for asynchronous execution.

The method returns the new **`CompletionStage.`**

In our solution provided in **`Listing 4-8`**, we have used **`form1`** that is, **`exceptionallyComposeAsync()`** without **`executor`**. Once the exception occurs, it executes the supplier function which is provided as an argument to the **`exceptionallyComposeAsync()`** method. If no exception occurs, the method executes normally.

Compact Number Formatting

To handle the data like percentage and currency, Java 12 had introduced a new class **`CompactNumberFormat`** which is a subclass of the **`NumberFormat`** class. This class is used to apply different formats on the numbers.

Problem

Explain how we can use the new **`CompatNumberFormat`** class of Java 12.

Solution

`Listing 4-9`, **`CompactNfDemo.java`** shows the different functionalities of **`CompactNumberFormat`** you can use:

```
1. //Listing 4-9
2. package com.numberformat;
3.
4. import java.text.NumberFormat;
5. import java.text.ParseException;
6. import java.util.Locale;
```

```
7.
8. public class CompactNFDemo {
9.
10.            public static void main(String args[]) throws
   ParseException {
11.
12.                                        NumberFormat    nf    =
   NumberFormat.getCompactNumberInstance(
13.                 Locale.US, NumberFormat.Style.SHORT);
14.
15.          nf.setMaximumFractionDigits(2);
16.
17.          System.out.println("NumberFormat.Style.SHORT:");
18.          System.out.println("Result: " + nf.format(10000));
19.          System.out.println("Result: " + nf.format(120300));
20.
21.          nf = NumberFormat.getCompactNumberInstance(
22.                 Locale.US, NumberFormat.Style.LONG);
23.
24.          System.out.println("\nNumberFormat.Style.LONG:");
25.          System.out.println("Result: " + nf.format(10000));
26.          System.out.println("Result: " + nf.format(120300));
27.
28.
29.          nf = NumberFormat.getCompactNumberInstance(
30.                 Locale.US, NumberFormat.Style.SHORT);
31.           System.out.println("\nConverting the string to SHORT
   number");
32.        System.out.println(nf.parse("1K"));
33.          System.out.println(nf.parse("1M"));
34.      }
```

Output

If you execute the preceding code, you will get the output as shown in *Figure 4.8*:

```
NumberFormat.Style.SHORT:
Result: 10K
Result: 120.3K

NumberFormat.Style.LONG:
Result: 10 thousand
Result: 120 thousand

Converting the string to SHORT number
1000
1000000
```

Figure 4.8: *Compact Number Format*

Explanation

`CompactNumberFormat` is a concrete subclass of the **`NumberFormat`** class. This class is used to format the currency or percentage number's general format. To use **`CompactNumberFormat`**, you need to use the static method **`getCompactNumberInstance()`** of the **`NumberFormat`** class.

The syntax of the method is as follows:

```
public static NumberFormat getCompactNumberInstance
            (Locale locale, NumberFormat.Style numberformatStyle)
```

The method parameters are:

- **`locale`**: The specific locale based on which formatting should generate values.
- **`numberformatStyle`**: Formatting style for a number.

The method returns a **`NumberFormat`** instance to be used for compact number formatting.

The method throws **`NullPointerException`**, if either **`locale`** or **`numberformatStyle`** is null.

Once you get the **`NumberFormat`**, you can use the following methods to format

your numbers:

- **`format(double number)`**

  Formats the number provided in the arguments according to the format provided by **`NumberFormat`**.

- **`parse(String number)`**

  Parses the String provided in the arguments according to the format provided by **`NumberFormat`**.

In our solution **`Listing 4-9`**, at line number 12, we have set up this compact number formatting as follows:

```
NumberFormat nf = NumberFormat.getCompactNumberInstance(Locale.US,
                                                        NumberFormat.Style.SHORT);
```

This returns a **`NumberFormat`** of Locale.US and it returns the number in the **`NumberFormat.Style.SHORT`** format. In this format, the number `10000` is formatted as `10K`.

If you change the style to **`NumberFormat.Style.Long,`** then the number `10000` is formatted as `10 thousands`.

You can also observe the usage of the **`parse()`** method. We have invoked **`nf.parse('1k')`**, which returns `1000`.



You can use the CompactNumberFormat class of Java 12 to format the numbers. While formatting the numbers in the format like SHORT, LONG, and so on, you can also provide locale to denote what language you wish to use to display the formatted number.

Conclusion

In this chapter we discussed many new features introduced in Java 12. We started our discussion with **`Collectors.teeing()`** which merges the different operations performed on collections into a single group. We then explored new methods like **`indent()`**, **`transform()`**, and so on, which are introduced in the **`String`** API. We also talked about the new method **`mismatch()`** to compare two files which is introduced in `NIO`. Apart from this, we also studied the new approach of handling the exceptions in **`CompletableFuture`** and how we can format the numbers using the **`CompactNumberFormat`** class.

In the next chapter, you will learn a few new features launched in Java 13 and Java 14. Both of these versions do not provide much of the improvements as far

as development is concern, so we will cover them in a single chapter.

Keywords

- Collectors.teeing()
- indent()
- transform()
- describeConstable()
- resolveConstantDesc()
- Files.mismatch()
- exceptionallyAsync()
- exceptionallyComposeAsync()
- getCompactNumberInstance()
- format()
- parse()
- CompletableFuture
- CompletionStage
- Collectors
- Constable
- ConstantDesc
- NumberFormat
- CompactNumberFormat

Questions

1. What is the significance of the **`Collectors.teeing()`** method, which is introduced in Java 12?
2. Can we transform the String which is transformed already by using the chaining of String? How to apply it?
3. In case of **`Files.mismatch()`**, if the paths of the two files are the same, but the files do not exist, what will be the output?
4. **`Files.mismatch()`** returns a number if the files do not match. What does this number specify?

5. Can we provide the executor to **`exceptionallyComposeAsync()`** which is added in **`CompletableFuture()`** ?

CHAPTER 5

Java 13 and 14 – Friends Forever

Introduction

In the previous chapters, we discussed the new features of Java, which were launched in the respective versions. We have seen many astonishing changes in the API, which made the job of Java developers simpler and maintainable. One might think that there is enough on the plate as far as API development is concerned. But hold on! Oracle never said that they will launch a version that will be the final version. Remember the concept of a time-based version released system, which promised us to get the updated version every six months. Java 13 is the next version as a product of the same time-based versioning system which was released in September 2019. Because of the deadline of six months, there is no guarantee that every released version will have major changes in the API. The changes may be minor, or sometimes the changes may be in the preview stage but the version will be released in the given timeline. Java 13 is one such version where more features are launched in the preview stage which were finalized in the next version. Of course, there are some changes as far as API is concerned which we will study in this chapter. A few of the preview features from Java 13 are launched as a final feature in Java 14, which was launched in March 2020. Java 14 had released some features for switch-case, compact number formatting, and so on. Also, this feature mainly focuses on performance and robustness. This chapter is divided into 2 parts. Part 1 is dedicated to updates in Java 13 and Part 2 covers updates on Java 14.

Structure

In this chapter, we will cover the following concepts:

- What's new in Java 13?
    - Updates in java.nio.file.FileSystems
    - Updates in XML parsers
- Updates in Java14

- Modified switch case syntax
- Using yield in the switch case

Objectives

After studying this chapter, you will be able to use the different features of Java 13 and Java 14 in different scenarios. You will learn how to use the newly introduced static methods in the `java.nio.file.FileSystems` class and see the changes in the API for parsing the XML files in Java 13. You will also learn how to use the modified switch case with its different flavours.

Part I: What's new in Java 13?

In the first part of this chapter, we will discuss a few API improvements added in Java 13. Many of the changes in this version are part of the preview feature. As we are going to discuss a few more versions in the upcoming chapters, we will cover those in the respective chapters where they are released. Because of this, we do not have many of the concepts to study in Java 13. A few of the changes are introduced in the **`java.nio.file`** package which we will discuss.

Installation

Before you start with this part of the chapter, download the Java SE Development Kit Version 13 from the following resource:

https://www.oracle.com/java/technologies/javase/jdk13-archive-downloads.html

Updates in java.nio.file.FileSystems

The **`FileSystems`** class is available in the **`java.nio.file`** package. This class is not launched in Java 13, it is available from Java 7 version. This class now provides some static methods from Java 13 onwards, which returns the **`FileSystem`** object.

Problem

What are the different providers available that facilitate to communicate with the file system in Java? Does the Java API provide any method to use these providers?

Solution

`Listing 5-1, FilesystemOperations.java` shows how we can list out different in-built file system providers available in Java. The code also shows the use of the new static method from the **`java.nio.file.FileSystem`** class, which uses one of these providers:

```
1. //Listing 5-1
2. package com.java13.nio;
```

```
3.
4. import java.io.IOException;
5. import java.nio.file.FileSystem;
6. import java.nio.file.FileSystems;
7. import java.nio.file.Paths;
8. import java.nio.file.spi.FileSystemProvider;
9. import java.util.Map;
10.
11. public class FileSystemOperations {
12.   public static void main(String[] args) throws IOException {
13.    System.out.println("Available providers==>");
14.              for    (FileSystemProvider    builtinprovider    :
   FileSystemProvider.
                                              installedProviders())
     {
15.     System.out.println(builtinprovider);
16.    }
17.
18.    Map<String, String> attr = Map.of("create", "true");
19.    FileSystem testfile = FileSystems.newFileSystem(
                                  Paths.get("first.docx"), attr);
20.    System.out.println("Provider used :- "+testfile.provider());
21.    System.out.println("Root directory:- "+
                                 testfile.getRootDirectories());
22.                      System.out.println("Read       attribute:-
   "+testfile.isReadOnly());
23.   }
24. }
```

Output

After the execution of **`FileSystemOperations.java`**, we get the output as shown in *Figure 5.1*:

```
sun.nio.fs.WindowsFileSystemProvider@4c873330
jdk.nio.zipfs.ZipFileSystemProvider@119d7047
jdk.internal.jrtfs.JrtFileSystemProvider@776ec8df
Provider used :- jdk.nio.zipfs.ZipFileSystemProvider@119d7047
Root directory:- [/]
Read attribute:- false
```

Figure 5.1: *Updates in the FileSystems class*

Explanation

The **`FileSystem`** class that was introduced as a part of the **`java.nio.file`** package in JDK 1.7 is essentially a collection of all the static methods to generate the file system. Depending upon the URI scheme passed, it identifies the providers required for that file type. These providers are pre-installed that are loaded by the system class loader. Initially, only a few methods were added in this class which communicates with the URI and occasionally with the Path to locate the file. From version 13, this class provides a few more methods to facilitate more flexibility in generating the file system.

The following new methods are added to this API.

newFileSystem(Path)

This static method returns the **`FileSystem`** object which is mapped to the file defined by **`Path`**. Using this object, you can access the contents of the specified file.

The syntax of method is as follows:

```
public static FileSystem newFileSystem(Path path) throws
IOException
```

This method is internally connected to another overloaded method which talks to the filesystem provider and the map which defines the properties or attributes of the specified file. If the provider is found, the empty map is pushed in this method and the file system is returned.

The method parameters are:

- **`path`**: The path to the file.

The method returns a new file system.

The method throws:

- **`ProviderNotFoundException`**: If there is no provider which can support the specified file type.

- **`ServiceConfigurationError`**: If the service provider is not loaded properly which is responsible to load the required provider.
- **`IOException`**: If the file is not available.
- **`SecurityException`**: If a security manager does not give permission to process the file.

newfileSystem(Path, Map<String,?>)

Similar to the previous method, this method also returns the file system. But you can provide the special attributes in the form of a map as the second argument of this method.

The syntax of the method is as follows:

```
public static FileSystem newFileSystem(Path path,
                                       Map<String,?> envmap) throws
  IOException
```

This method works similar to the earlier method, but in addition, the attributes of the file system are specified in the map. Map, as you know, is a key value pair that will provide the attribute's name in the key and the attribute's value will be provided in the value.

The method parameters are:

- **`path`**: The path to the file.
- **`envmap`**: A map which provides the properties which configures the file system.

The method returns a new **`FileSystem`**.

The method throws:

- **`ProviderNotFoundException`**: If there is no provider which can support the specified file type.
- **`ServiceConfigurationError`**: If the service provider is not loaded properly which is responsible to load the required provider.
- **`IOException`**: If the file is not available.
- **`SecurityException`**: If a security manager does not give permission to process the file.

In our solution, that is, **`Listing 5-1`**, we have initiated the map on line number 18 as follows:

```
Map<String, String> attr = Map.of("create", "true");
```

This attribute specifies that if the file is not present, it will be created automatically. Further, we have passed this map in the **`FileSystems.newFileSystem()`** method as shown below in line number 19 of our solution:

```
FileSystem testfile = FileSystems.newFileSystem(
                                    Paths.get("first.docx"), attr);
```

As the attribute "**`create`**" is set to "**`true`**", when you execute the program for the first time, you will observe that the file **`first.docx`** is created automatically. As mentioned, depending upon the file type, the service provider will load the provider from the already installed providers. In our case, you will observe in the output that **`jdk.nio.zipfs.ZipFileSystemProvider`** provider is used to generate the file system.

newFilesystem(Path, Map<String,?>, ClassLoader)

This method can be termed as a completely customized method as you can provide **`Path`**, **`Map`**, and also the **`Classloader`** to load the file system.

The syntax of the method is as follows:

```
public static FileSystem newFileSystem(Path path, Map<String,?>
envmap,
                              ClassLoader loader) throws IOException
```

When we invoke this method, the installed providers are searched as per the file type. But if the matching provider is not found, the classloader which is provided as the third argument locates the required provider. Once the provider is loaded, the file system is returned for further file processing.

The method parameters are:

- **`path`**: The path to the file.
- **`envmap`**: A map which provides the properties which configures the file system.
- **`loader`**: The class loader which is used to locate the required provider. This can be also null to locate an installed provider.

The method returns a new **`FileSystem`** .

The method throws:

- **`ProviderNotFoundException`**: If there is no provider which can support the specified file type.
- **`ServiceConfigurationError`**: If the service provider is not loaded properly which is responsible to load the required provider.

- **`IOException`**: If the file is not available.
- **`SecurityException`**: If a security manager does not give permission to process the file.

Updates in XML Parsers

Often, we need to work with XML files in an application. Such XML files are parsed using the **`SAXParser`** and **`DOMParser`** API. Java 13 introduced some new methods to instantiate the `DOM` and `SAX` factories. These methods provide awareness about the namespaces which are provided in the XML meta information tags. The names of these methods are kept similar to traditional methods, but they are now prefixed with "**`NS`**". The prefix "**`NS`**" indicates that it is **`NamespaceAware`**.

Following are the newly introduced methods:

- **`newDefaultNSInstance()`**
- **`newNSInstance()`**
- **`newNSInstance(String factoryClassName, ClassLoader classLoader)`**

These methods are used to create the parser which will be by default NamepsaceAware.

Consider the following snippet:

```
DocumentBuilder documentBuilder =
      DocumentBuilderFactory.newDefaultNSInstance().newDocumentBuil
```

This is a replacement for the following code snippet:

```
DocumentBuilderFactory documentBuilderFactory =
  DocumentBuilderFactory.newDefaultInstance();
documentBuilderFactory.setNamespaceAware(true);
DocumentBuilder db = documentBuilderFactory.newDocumentBuilder();
```

As you can observe, the modified approach of **`DocumentBuilderFactory`** is more compact than the earlier approach.

Let us discuss the new methods in detail.

newDefaultNSInstance()

This method returns the **`NamespaceAware`** instance of **`DocumentBuilderFactory`**, with the built-in system-default implementation.

The syntax of the method is as follows:

```
public static DocumentBuilderFactory newDefaultNSInstance()
```

Once you get the factory instance, you can create the parser from this. This parser by default has the support for XML namespaces.

The method returns a new instance of the **`DocumentBuilderFactory`** built-in system-default implementation.

A similar method is available for generating the factory for **`SAXParser`**.

newNSInstance()

This method returns the NamespaceAware instance of **`DocumentBuilderFactory`**.

The syntax of the method is as follows:

```
public static DocumentBuilderFactory newNSInstance()
```

The method returns a new instance of a **`DocumentBuilderFactory`**.

The method throws **`FactoryConfigurationError`**, if the **service configuration error** occurs or if the implementation is not available or cannot be instantiated.

A similar method is available for generating the factory for **`SAXParser`**.

newNSInstance(String,ClassLoader)

This method facilitates to provide the factory class name and class loader to load the factory class with the help of two arguments.

The syntax of the method is as follows:

```
public static DocumentBuilderFactory newNSInstance(String
factoryClass,
          ClassLoader classLoader)
```

The method parameters are:

- **`factoryClass`**: A factory class name that provides implementation of **`javax.xml.parsers.DocumentBuilderFactory.`**
- **`classLoader:`** The **`ClassLoader`** used to load the factory class.

The method returns a new instance of a **`DocumentBuilderFactory`**.

The method throws **`FactoryConfigurationError`** if **`factoryClass`** is null, or if the factory class is not loaded or instantiated because of some reason.

A similar method is available for generating the factory for **`SAXParser`**.

Part II: Updates in Java 14

Java 14 comes up with a lot many updates which are more focussed towards robustness, performance, and security aspects. Many of such features execute as background processes. The following notable changes are implemented in this version:

- Thread Interrupt State is always available
- Zip File System throws **`java.nio.file.NoSuchFileException`** when the file does not get created
- Better serial filter handling
- Improved Registry Support
- Plural Support in the Compact Number Format
- Modified Switch Case Statement

In this part, we will cover the updates which are important for the implementation like modified switch case and plural support in the compact number format.

Installation

Before you start to read this part of the chapter, download the Java SE Development Kit Version 14 from the following resource:

`https://www.oracle.com/java/technologies/javase/jdk14-archive-downloads.html`

Updates in Switch-Case

Using the control statement is always an integral part of any of the business solution. The concepts like **`if..else`**, **`switch-case`** are not new to any of us. But the traditional switch case statement was not flexible enough to handle different forms in the cases. That is the reason you might have observed that the developers were reluctant to use the switch case. That was always their second choice. Java 14 modified the overall way of declaration of switch case and made it more flexible.

The modified switch case now can use:

- Multiple labels in case

- Arrow operator
- Expression in the case
- Yield statement

Let us discuss all these features in different situations.

Problem

The traditional switch-case was not allowing multiple case labels, but now it does. How to write cases having multiple labels?

Solution

We will discuss this solution with two scenarios:

Scenario 1: without the break statement:

`Listing 5-2, TestJava14SwitchMultipleCase_Label1_WithoutBreak.java` shows how we can use multiple case labels without using the break statement:

```
1. //Listing 5-2
2. package com.java14;
3. public class TestJava14SwitchMultipleCase_Label_WithoutBreak {
4.
5. public static void main(String[] args) {
6.   checkFruit("apple");
7.   checkFruit("watermelon");
8.
9.   }
10.
11. private static void checkFruit(String fruitName){
12.   switch (fruitName) {
13.   case "apricot", "apple", "banana":
14.       System.out.println(fruitName + " is full of Vitamin A,B١,B٢,B٦,
                                C and folic acid");
15.   case "Orange":
16.    System.out.println("Orange is full of Vitamin C,potassium,
```

```
                                    folate and thiamine");
17.    default:
18.     System.out.println("We don't have fruit on the list");
19.    }
20. }
21. }
```

Output

If we execute **`TestJava14SwitchMultipleCase_Label1_WithoutBreak.java`**, we will get the following output, as shown in *Figure 5.2*:

```
apple is full of Vitamin A,B1,B2,B6,C and folic acid
Orange is full of Vitamin C,potassium, folate and thiamine
We dont have fruit on the list
We dont have fruit on the list
```

Figure 5.2: *Modified Switch with Multiple Labels without break*

Scenario 2: with the break statement:

`Listing 5-3, TestJava14SwitchMultipleCase_Label_Break.java`, shows how we can use multiple case labels with the break statement:

```
1. //Listing 5-3
2. package com.java14;
3.
4. public class TestJava14SwitchMultipleCase_Label_Break {
5.
6. public static void main(String[] args) {
7.    checkFruit("banana");
8.    checkFruit("watermelon");
9. }
10. private static void checkFruit(String fruitName) {
11.    switch (fruitName) {
12.    case "apricot", "apple", "banana":
13.        System.out.println(fruitName + " is full of Vitamin
    A,B١,B٢,B٦,
```

```
                                        C and folic acid");
14.     break;
15.    case "Orange":
16.     System.out.println("Orange is full of Vitamin C,potassium,
                                          folate and thiamine");
17.     break;
18.    default:
19.     System.out.println("We don't have fruit on the list");
20.    }
21. }
22. }
```

Output

If we execute **`TestJava14SwitchMultipleCase_Label_Break.java`**, we will get the output as shown in *Figure 5.3*:

```
banana is full of Vitamin A,B1,B2,B6,C and folic acid
We don't have fruit on the list
```

Figure 5.3: *Modified Switch with Multiple Labels with break*

Explanation

In the traditional switch-case, we were only able to use a single case. This leads to write multiple unnecessary cases and we were not able to combine similar operations under the same case. This would always create redundant code. But now, with the enhanced features we can combine similar operations under the same case by providing comma separated labels.

We have created two different scenarios to explain this concept.

In scenario 1, that is, **`Listing 5-2, TestJava14SwitchMultipleCase_Label_WithoutBreak.java`**, we have declared comma separated multiple cases in the same case instead of writing different cases for the identical business logic. But in this scenario, we have not used the break statement after the cases. Because of this fall-through, the concept will be generated, that is, all the business logic after the matching case will execute without checking any further cases. You can observe in the output shown in *Figure 5.2* that, even though the case for fruitName, **`"apple"`** is matching, we are

getting the output for **"orange"** and **"default"**. Of course, when we pass **"watermelon"**, none of the cases pass because of which the **"default"** case executes.

In scenario 2, that is, **Listing 5-3**, **TestJava14SwitchMultipleCase_Label_Break.java**, we have used the same code (multiple cases are declared in the same case, if they are supposed to execute the identical business logic) but this time, we have used the break statement after each case. Because of the break statement only the matching case will execute and then the switch block will terminate. In the output shown in *Figure 5.3*, you will observe that when we pass **"banana"**, only the business logic for the case which matches with **"banana"** will execute. None of the other cases are executed. When we passed **"watermelon"**, the default case is executed as there is no case matching with **"watermelon"**.



Java 14 provides the facility to write the common code for multiple cases with more flexibility:

- Multiple cases can be declared by separating them with a comma (,) in the single case.
- Do not forget the write the break statement after each case block, else the fall-through condition would arise, which is similar to the traditional switch case.

Problem

In comparison to the traditional switch-case, how to use the new form of the switch label "**case L->**"? How to use an arrow operator in the switch case?

Solution

Listing 5-4, TestJava14SwitchWithoutBreak.java shows how to use the arrow operator in the switch case statement:

```
1. //Listing 5-4
2. package com.java14;
3.
4. public class TestJava14SwitchWithoutBreak {
5. public static void main(String[] args) {
```

```
    checkFruit("apple");
    checkFruit("watermelon");
}

private static void checkFruit(String fruitName) {
    switch (fruitName) {
    case "apricot" ->
            System.out.println("Apricot is full of Vitamin
A,B1,B2,B6,
                                    C and folic acid");
    case "apple" ->
            System.out.println("Apple is full of Vitamin
A,B1,B2,B6,C
                                    and folic acid");
    case "banana" ->
            System.out.println("Banana is full of Vitamin
A,B1,B2,B6,C
                                    and folic acid");
    case "Orange" ->
            System.out.println("Orange is full of Vitamin
C,potassium,
                                    folate and thiamine");
     default -> System.out.println("We dont have fruit on the
list");
    }
}
}
```

Output

The following output as shown in *Figure 5.4* is observed, when we execute **`TestJava14SwitchWithoutBreak.java`**:

```
Apple is full of Vitamin A,B1,B2,B6,C and folic acid
We dont have fruit on the list
```

Figure 5.4: *Use of an arrow operator (->) in the switch case*

Explanation

In the traditional switch case, the developers were using labels followed by a `colon(:)`. Now, a new form **`"case L ->"`** has been adapted. This new form signifies that the code to the right of the label will be executed when the label is matched. One case can also have multiple constants per case which are separated by commas. The most important property of using an arrow operator in case is you need not have to use the break statement after the cases.

In the output shown in *Figure 5.4*, you can observe that when we passed **`"Apple"`** to the method, the case matching with **`"Apple"`** is executed. After that case, there is no break statement. But still there is no fall-through process that was observed in earlier cases. When the case is matched, the other cases are not checked at all. If no cases are matched, the default block is executed as shown in the output.



You can now use the arrow operator (->) instead of a colon (:) to denote the cases. If you use the arrow operator, there is no compulsion to use the break statement after each case.

Problem

Can we have multiple labels in the same case while writing the switch block using the arrow operator?

Solution

`Listing 5-5, TestJava14SwitchMultipleCase_Labels.java` shows how we can combine the use of multiple labels and arrow in the switch case statement:

```
1. //Listing 5-5
2. package com.java14;
3.
4. public class TestJava14SwitchMultipleCase_Labels{
5.     public static void main(String[]args){
6.       checkFruit("apricot");
7.       checkFruit("Orange");
8.       checkFruit("Watermelon");
9.     }
```

```
10.    private static void checkFruit(String fruitName){
11.     switch(fruitName){
12.        case "apricot", "apple","banana" ->
               System.out.println(fruitName +" is full of Vitamin
    A,B1,
                                        B2,B6, C and folic acid");
13.        case "Orange" ->
               System.out.println("Orange is full of Vitamin
    C,potassium,
                                      folate and thiamine");
14.        default ->
                System.out.println("We don't have fruit on the
    list");
15.    }
16.   }
17. }
```

Output

The following output shown in *Figure 5.5* is observed, when we execute, **`TestJava14SwitchMultipleCase_Labels.java`**:

```
apricot is full of Vitamin A,B1,B2,B6,C and folic acid
Orange is full of Vitamin C,potassium, folate and thiamine
We dont have fruit on the list
```

Figure 5.5: *Multiple cases with the arrow operator*

Explanation

The developers now can write an expression, a block, or a throw statement to the right of a **`"case L ->"`**. Also, as per the new propose now it's allowed to have multiple constants per case which are separated by commas.

In the solution, shown in **`Listing 5.5`**, you will observe that we have used multiple cases in the single case statement which are separated by commas on line number 12.

In the output shown in *Figure 5.5*, we can see that if any of the cases matches with the comma separated values, then that code block is executed. If none of the cases match, the **`"default"`** case is executed.



It is possible to have multiple labels separated by commas using the arrow operator with the help of a new format 'case L->'.

Problem

What is the scope of a local variable declared in the case of the switch block?

Solution

To understand the concept of the scope of variables in the switch-case, we will go through different scenarios shown in **`Listing 5-6, TestJava14ScopeOfLocalVariable`**.

Scenario 1: Traditional switch case and local variables

The following code snippet shows the traditional approach of declaring and using local variables:

```
1. private static void traditionalSwitch(String choice, int no1,
                                                    int no2) {
2.     switch (choice) {
3.     case "addition":
4.      int value = no١ + no٢;
5.      System.out.println("Addition:-" + value);
6.      break;
7.     case "substraction":
8.      value = no١ - no٢;
9.      System.out.println("Substraction:-" + value);
10.     break;
11.    default:
12.     value = ٠;
13.     System.out.println("Wrong Option:-" + value);
14.    }
15.  }
```

Scenario 2: Local variables in the modified switch case

The following code snippet shows the first approach of declaring and using local

variables in the modified switch case:

```
1. private static void modifiedSwitchType1(String choice, int no1,
                                                    int no2) {
2.     switch (choice) {
3.     case "addition" -> {
4.      int value = no١ + no٢;
5.      System.out.println("Addition:-" + value);
6.     }
7.     case "substraction" -> {
8.      int value = no١ - no٢;
9.      System.out.println("Substraction:-" + value);
10.    }
11.    default -> {
12.     int value = ٠;
13.     System.out.println(value);
14.    }
15.    }
16.   }
```

Scenario 3: Local variable declared outside the switch block in the modified switch case

The following code snippet shows the second approach of declaring and using the local variable which is declared outside the cases:

```
1. private static void modifiedSwitchType2(String choice, int no1,
                                                    int no٢) {
2.     int value = ٠;
3.     switch (choice) {
4.     case "addition" -> {
5.      System.out.print("Addition:-");
6.      value = no١ + no٢;
7.     }
8.     case "substraction" -> {
```

```
9.      System.out.print("Substraction:-")
10.     value = no١ - no٢;
11.    }
12.    default -> {
13.     value = ٠;
14.    }
15.    }
16.    System.out.println(value);
17.   }
18. }
```

Output

The following output is observed as shown *Figure 5.6* when we execute **`TestJava14ScopeOfLocalVariable.java`**:

```
Using traditional switch case:-
Addition:-30
Using local variable inside case:-
Addition:-30
Using local variable outside case:-
Addition:-30
```

Figure 5.6: *Local variables in the switch case*

Explanation

Traditionally, we used to declare a local variable either outside the switch block and then use it inside cases or declare a local variable and use it in the preceding cases. This is the same thing we have followed in scenario 1 as shown in the method **`traditionalSwitch()`** of **`Listing 5-6`**.

The scenario 2 is targeting to write cases using the new format of **`"case L->".`** To the right side of the **`"case L ->"`**, is restricted to have expression, a block, or a throw statement only. It means if we want expressions, we can write them without any assignment operator. However, when we need an expression, the local variable needs to be declared and to be enclosed within the pair of braces. We all are aware of the variable declared inside the block has the scope limited to the block itself. By this rule in scenario 2, the variable **`'value'`** has scope

limited to the block and in each subsequent blocks belonging to the cases, we can declare another variable named **`'value'`**. You can observe this in the method **`modifiedSwitchType1()`** of **`Listing 5-6`**.

Scenario 3 is more interesting. Now, the question is if the variable has the scope of block belonging to the case, what happens if we have a variable declared outside the switch? Not to our surprise, it will give us an error. This is not the case with the modified switch case in Java 14. If you observe the method **`modifiedSwitchType2()`** of **`Listing 5-6`**, the variable **`'value'`** as declared outside the switch inside a method. In such a case, the scope of the variable is for the entire method body. Anyone who tries to declare the **`'value'`** variable will be treated as redeclaration and not as declaration.



The following key points must be noted when you consider the scope of variables in the switch case expression:

- You can declare the variable in the case which has the scope limited to that case only.
- The variable declared inside the block belonging to the 'case L->' format has the scope limited to the block.
- The variable declared outside the switch, is accessible to all the cases which belongs to that switch block.

Problem

How to use the expressions in the switch?

Solution

You can observe the different ways to handle the expressions in the modified switch in the **`Listing 5-7, TestSwitchExpression.java`**. In this class, different scenarios are defined in different methods, which are discussed as follows:

Scenario 1: Without the switch expression

```
1. private static void switchWithExpressionForm1(int dayOfWeek) {
2.   // TODO Auto-generated method stub
3.   // scenario 1 :without switch expression
4.     switch (dayOfWeek) {
```

```
    case ١ -> System.out.println("It›s Monday");
    case ٢ -> System.out.println("It›s Tuesday");
    case ٣ -> System.out.println("It›s Wednesday");
    default -> System.out.println("Incorrect value for Day");
   }
  }
```

Or, alternatively we can also write the preceding code as follows:

```
private static void switchWithExpressionForm1(int dayOfWeek) {
 // TODO Auto-generated method stub
 // scenario 1 :without switch expression
String value;
switch (dayOfWeek) {
    case ١ -> value="It›s Monday";
    case ٢ -> value="It›s Tuesday";
    case ٣ -> value="It›s Wednesday";
    default -> value="In correct value for Day");
   }
  }
```

Scenario 2: With the switch expression

```
private static void switchWithExpressionForm2(int dayOfWeek) {
 // TODO Auto-generated method stub
 // scenario 2:with switch expression
   String result = switch (dayOfWeek) {
    case ١ -> "It›s Monday";
    case ٢ -> "It›s Tuesday";
    case ٣ -> "It›s Wednesday";
    default -> "Incorrect value for Day";
   };
   System.out.println("Result:-" + result);
  }
```

Scenario 3: Switch expression and System.out.println()

```
1. private static void switchWithExpressionForm3(int dayOfWeek) {
2.  // TODO Auto-generated method stub
3.  // scenario 3 : switch expression and System.out.println
4.    System.out.println("Result :"+switch (dayOfWeek) {
5.     case ١ -> "It›s Monday";
6.     case ٢ -> "It›s Tuesday";
7.     case ٣ -> "It›s Wednesday";
8.     default -> "Incorrect value for Day";
9.    });
10.  }
```

Output

The following *Figure 5.7* is generated if we execute *TestSwitchExpression.java*:

```
Switch - case without expression:-
Result:-It's Monday
Switch - case with expression Type1:-
Result:-It's Tuesday
Switch - case with expression Type2:-
Result :It's Wednesday
```

Figure 5.7: *Switch case with the expression*

Explanation

Java 14 gives different approaches to work with the expression that we write as a part of case blocks. In scenario 1, which is written in the method **`switchWithExpressionForm1()`** of Listing, we have used **`System.out.println(expression)`**. Instead of just the print statement, it can be any other expression, logic, or manipulation. We can store the value from the expression in some local variable and print it as an alternative way. But this style is repetitive, roundabout, and more over error-prone.

Instead, we can write this with much clearer and safer way as shown in scenario 2. In the method **`switchWithExpressionForm2(), of Listing 5-7, TestSwitchExpression.java`**, we have collected the value from each expression

of the case into a single variable. So, whichever case matches, the expression for that case is evaluated and the value of that is stored into the variable which is mapped with the entire switch block.

The format of the switch expression looks like:

```
T result = switch (argument) {
    case Label1 -> expression1;
    case Label2 -> expression2;
    default -> expression3;
};
```

Another interesting way of using the expression in switch is wrapping the entire switch block into the **`System.out.println()`** statement. This scenario 3 is implemented in **`switchWithExpressionForm3() of our solution in Listing 5-7, TestSwitchExpression.java`**. This looks like anonymous inner class, *BUT IT IS NOT*!

The format of the switch expression looks like:

```
System.out.println(switch (argument) {
    case Label1 -> expression1;
    case Label2 -> expression2;
    default -> expression3;
});
```



The following are the advantages of the new switch-case statement while working with expressions:

- The switch expression allows a developer to write much clean, safe, and nonrepetitive code by returning a value by each case arm.
- You can also wrap the switch case in the output statement. So, the generated value from the matching expression will be printed automatically.

Problem

Can we perform some operations in the case and pass the result to some variable using the expression in the switch?

Solution

Let us understand this concept by different scenarios as follows:

Scenario 1: Switch expression and performing an operation without the code block:

```
int result = switch (choice) {
```

```
case "addition" -> no1 +no2;
default ->
   0;
};
```

Scenario 2: switch expression and cases with the code block:

```
int result = switch (choice) {
case "substraction" -> {
   no1 - no2;
}
default ->
  0;
};
```

Scenario 3: Switch expression and cases with the code block having the return statement:

```
int result = switch (choice) {
case "multiplication"-> {
   return no1 * no2;
}
default ->
  0;
};
```

Explanation

The switch statement is extended to use as the switch expression. In general, the switch expression will have a single expression to the right of **`case L->`**. And the value evaluated after execution of that expression is return. As discussed in the scenario 1, case **`"addition"`** will provide a value of two numbers using the plus operator and then it will return to initialize the variable result.

The second scenario will fail as we have a code block instead of having a single statement or expression. In the same way, the third scenario will lead to an error. Why? We have a complex logic so we used braces to surround the logic and on top of that, we also used the return statement to return the evaluated value. Then? Yes, we are using the switch expression and it doesn't allow the **`return`** statement. Whenever we have a case with the complex logic, we need to remember two things:

- Surround the logic and place the case in the curly braces.
- Instead of using return the value, use the **`Yield`** identifier.



The complex logic performed by the case arm must be enclosed within the pair of curly braces. Use the identifier yield so as to return the resultant value from the case block.

Problem

What is the use of **`Yield`** in the switch expression?

How to return a calculated value in the case block of switch?

Solution

We can make use of yield in both the traditional case operator, colon **`(:)`** as well as in enhanced arrow**`(->)`** operator. In our solution, that is, in **`Listing 5-8`**, we have used yield in both these types in different methods which are shown as follows:

Scenario 1: Yield with the traditional switch case:

```
1. private static void swithCaseYieldForm1(String choice, int no1,
                                            int no2) {
2.    System.out.println("Using Yield in traditional switch case:-
   ");
3.    int result = switch (choice) {
4.    case "addition":
5.     yield (no١ + no٢);
6.    case "substraction":
7.     int sub = no١ - no٢;
8.     yield sub;
9.    case "multiplication":
10.    yield (no١ * no٢);
11.   default:
12.    yield ٠;
13.   };
14.   System.out.println("Result:-" + result);
15.  }
```

Scenario 2: Yield with the modified switch case:

```
1. private static void switchCaseYieldForm2(String choice, int
   no1,
                                                int no2) {
2.     System.out.println("Using Yield in modified switch case:-");
3.     int result = switch (choice) {
4.     case "addition" -> no١ + no٢;
5.     case "substraction" -> {
6.      int sub = no١ - no٢;
7.      yield sub;
8.
9.     }
10.    case "multiplication" -> {
11.     yield no١ * no٢;
12.    }
13.    default -> ٠;
14.    };
15.
16.    System.out.println("Result:-" + result);
17.   }
```

Output

We will get the following *Figure 5.8* when you execute the **`TestSwitchExpression Yield.java`** file:

```
Using Yield in traditional switch case:-
Result:-12
Using Yeild in modified switch case:-
Result:-12
```

Figure 5.8: *Using Yield in switch*

Explanation

Most of the times when the developers use switch expressions, it will have a single line expression to the right side of the **`"case L ->"`**. However, sometimes a single line expression may not be sufficient, instead the developers might need a full block. To handle such a situation, a new yield statement is introduced. The yield value becomes the value of the enclosing the switch expression.

When we use the switch expression, we don't use the break statement. Also, the return statement is not allowed in the switch expression. Instead, we can use the new identifier, yield. The yield identifier will result in returning the calculated value of the enclosed expression. One might get confused between the break and yield statements.

The following points will differentiate them as follows:

- In the traditional switch-case, one needs to use break to avoid the fall-through concept of cases.
- In the switch expression, use of break is not allowed.
- In the switch expression, if one uses the code block, we need to use the `Yield` identifier.

In our solution, that is, **`Listing 5-8`**, we have defined the method **`swithCaseYieldForm1()`**, inside which we have used the traditional switch case with an expression. Observe that there is neither a break statement nor return statement. But the **`Yield`** returns a value from the expression, which is ideally the value of the switch block. In the same program, we have defined **`swithCaseYieldForm2()`**. In this method, we have used the case with the arrow operator which is then followed by the expression. The value of the expression is returned by the **`Yield`** statement and not by the return statement.



To return the value generated from the expression defined in the case, we can use the Yield statement:

- We can use the Yield statement in the traditional switch case, where we are expecting an expression in a case, which returns a value.
- We can use the Yield statement in the switch case which uses the arrow operator to denote the case. The expressions value is returned by the Yield statement.

Better Approach of the NullPointerException

description

One of the most serious exceptions in Java is **`NullPointerException`**. Though we can handle this exception, earlier versions of Java do not give clear information about this exception. In Java 14, the information about this exception is more explanatory.

Problem

How the **`NullpointerException`** is enhanced?

Solution

`Listing 5-9, DemoEnhancedNullpointerException.java` shows the enhanced approach of NullPointerException in Java 14:

```
1. //Listing 5-9
2. package com.java14;
3.
4. public class DemoEnhancedNullpointerException {
5.
6.   public static void main(String[] args) {
7.    String name=null;
8.    System.out.println("length of the name:-"+ name.length());
9.   }
10.
11. }
```

Output

We have to compile and execute this program by using the pre Java 14 compiler and Java 14 compiler to understand how Java 14 gives more information about this exception.

Using pre Java14 compiler

You will get the following output as shown in *Figure 5.9* when you execute the program using any Java version prior to Java 14:



Figure 5.9: *NullPointerException PreJava14 compilation*

Using the Java 14 compiler

You will get the following output as shown in *Figure 5.10* if you execute the same program using the Java 14 compiler:

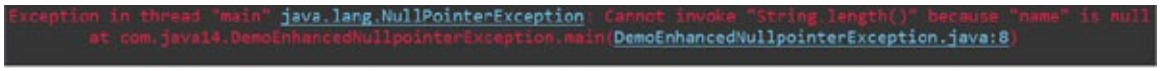


Figure 5.10: *NullPointerException Java14 compilation*

Explanation

When we develop any application in the object-oriented programming language, we play with the objects. We create objects, retrieve objects, and modify the objects. Though there is not much complexities while we create an object, when we retrieve an object from some other resources the things might not work as per expectations. The object that we are trying to process might not have initiated because of some or the other reasons. The problem is this will not be tracked while we compile the application. But it will generate **`NullPointerException`** at runtime. The exception message generated by JVM in such cases is very generic and it is bit difficult to get accurate information about the source and reason of this exception from this message.

In our output shown in *Figure 5.9*, you can see that only the type of exception and the line number is shown. At this point, you might say "*This is fine! What else do I need to handle this?*"

Well, you are correct to some extent because the code is not complex. What if the code is so complex that the single line contain more than one object reference and you are in dilemma about which reference or object is generating this **`NullPointerException`**? That's what the thought behind modifying the **`NullPointerException`** message in Java 14.

As per the updates proposed and accepted from Java 14 onward more descriptive way of specifying the cause of null pointer exceptions is adapted. Now, when the JVM throws a **`NullPointerException`** at any point in the program where code tries to refer the null, the program's bytecode instructions determine exactly which variable is referring to null. Then, it will describe the variable with a message why the variable is null. This detailed message then will be shown in the stack trace message with the method, filename, and line number. You can

observe this description in our output shown in the previous *Figure 5.10*.

The message which is described on the console has two parts:

- The first part specifies which action could not be performed because a bytecode instruction popped a null reference from the operand stack.
- The second part is more complex and in detail the reason for the **`NullPointerException`** is provided. This part recreates the part of the source code which is pushed the null reference on to the operand stack.



Java 14 provides more descriptive and detailed approach to throw the NullPointerException. Along with the exception type, it also describes which variable is responsible for this exception and what is the reason behind it.

Plural Support in the Compact Number Format

From Java 14 onwards, we get a plural support for the German language in the compact number format.

Problem

Which new feature is introduced in the compact number format in Java 14?

Solution

`Listing 5-10, PluralCompactNumberFormat.java` shows the actual invocation of the feature introduced in Compact Number Formatting:

```
1. //Listing 5-10
2. package com.java14;
3.
4. import java.text.NumberFormat;
5. import java.util.List;
6. import java.util.Locale;
7.
8. public class PluralCompatNumberFormat {
9.   public static void main(String[] args) {
10.         compactFormatTest(Locale.CANADA);
```

```
11.         compactFormatTest(Locale.GERMANY);
12.     }
13.
14.     private static void compactFormatTest(Locale locale) {
15.             List<Integer> numList = List.of(,0·····  ,\···
    \·····);
16.         System.out.println("Numbers in locale:"+ locale);
17.         numList.stream().forEach((num) -> {
18.                                         NumberFormat nf =
    NumberFormat.getCompactNumberInstance(
                    locale, NumberFormat.Style.LONG);
19.             System.out.println(nf.format(num));
20.         });
21.     }
22. }
```

Output

The following *Figure 5.11* is generated when we execute **`PluralCompatNumberFormat.java`**:

```
Numbers in locale:en_CA
1 thousand
5 million
1 million
Numbers in locale:de_DE
1 Tausend
5 Millionen
1 Million
```

Figure 5.11: *Plural Support in the Compact Number Format*

Explanation

Java 14 has modified the approach of dealing with the Compact Number Format for a specific Locale. In this version, the plural support is extended for German Locale. In our solution provided in **`Listing 5-10, PluralCompactNumberFormat.java`**, we have formatted the numbers provided

by a list as follows:

```
List<Integer> numList = List.of(1000, 5000000, 1000000);
```

When this list is formatted for Canadian Locale, you will observe that for 1000000 and 5000000, there is no different in singular and plural format. But when we formatted the same numbers for German `Locale`, you will observe that for number 5000000, the compact number format is changed to `Millionen` instead of `Million`, as this number is plural.



You can take the advantage of the new enhancement in the Compact Number Format. In this update, Java provides the support for identifying the plural numbers format it accordingly. This provides more readability for the compact number handling.

Conclusion

In this chapter, we discussed some new features launched in versions 13 and 14. Both these versions do not provide much features for implementations. But we covered a few of the new features, which are important for application development. We started the chapter with updates in Java 13, where we discussed new static methods introduced in **`java.nio.file.FileSystems`** and the new approach of generating XML parsers. In the second part of this chapter, we discussed a few features introduced in Java 14. We learned exclusively about different improvements in **`switch-case`**. We also talked about the updates in the Compact Number Formatting and a more descriptive way of throwing **`NullPointerException`**.

In the next chapter, we will learn the new features launched in Java 15 using some interesting recipes.

Key terms

- FileSystem
- FileSystems
- Switch-case
- Yield
- newDefaultNSInstance()
- newNSInstance()
- DocumentBuilder

- DocumentBuilderFactory
- NullPointerException

Questions

1. What are the different ways to create the SAXParser in Java 14?
2. How do you return the value from an expression in the modified switch-case statement?
3. What is difference between a return and yield statement?
4. What are the different providers available to generate the file system?
5. Which new features are introduced in switch-case in Java 14?

CHAPTER 6

Java 15 – I am 25 Years Old

Introduction

It is time to celebrate. Java completed 25 years in 2020. It is not simple in this competitive world to not only complete 25 years, but to also maintain the top position as a leading programming language for developers across the globe. During these 25 years, Java provided some notable features in different versions. If we want to recall a few famous concepts, we can talk about Generics from Java 5, Lambda expressions from Java 8, and of course, modules in Java 9. Moreover, the new operational concept introduced was a time-based version release from Java 10, in which a new version is released every six months. Following the same path, Oracle released Java 15 on 15th September 2020.

Java 15 consists of a few functionalities such as the Hidden class and **Edwards-Curve Digital Signature Algorithm** (**EdDSA**), which is used for better security aspects. Moreover, some of the preview features from earlier versions like text blocks and ZGC were released as the final release in this version. Though not many of the changes or updates are incorporated in this version, as a developer these are still quite noteworthy changes.

Structure

In this chapter, we will cover the following concepts:

- Using text blocks
- Hidden classes
- Improvement in the CharSequence interface

Objectives

The aim of this chapter is to learn new features which were launched in Java 15. We will discuss some interesting features such as text blocks and hidden classes, which are termed the main features of Java 15. We will also discuss some minor changes in this version, such as updates in the CharSequence interface.

Installation

Before you start reading this part of the chapter, download the Java SE Development Kit Version 15 from the following resource:

https://www.oracle.com/java/technologies/javase/jdk15-archive-downloads.html

Using text blocks

The preview feature to add text blocks to Java had been added in JDK 13, and it had been accepted as a full offering in JDK 15. A text block is considered as a multiline string literal, which avoids the need for most escape sequences and automatically formats the string in a predictable way. It also provides control over the formatting whenever desired. The text block can be used to replace a string wherever we were previously using a string literal since it is more expressive and has fewer chances of accidental complexity. Before implementing the text blocks, we first need to understand what complexity we faced prior to the text block, while still using multi-line Strings in our code.

In Java, when we need to embed an HTML, XML, SQL, or JSON in a string literal enclosed in double quotes (" "), it requires a significant amount of editing with a lot of escape sequences along with concatenation. This embedding makes the snippet often difficult to read and maintain.

Let us understand this with multiple scenarios discussed in the following sections.

Scenario 1: Handling string embedded JSON using the + operator

The following code **`Listing 6.1, TestStringWithConcatationOperator.java`** shows how we can declare and initiate JSON which is embedded in the String format:

```
1. //Listing 6.1
2. package com.java15;
3.
4. public class TestStringWithConcatationOperator{
5.     public static void main(String[] args) {
6.         String stringJSON = "{\r\n"
7.             + "\"Name\" : \"Java\","
```

```
            + "\n\"RollNO\" : \"20\"\r\n"
            + "}";
        System.out.println(stringJSON);
    }
}
```

Explanation

The Java language can be regularized by accepting such strings that may be too large to span multiple lines of a source file. Furthermore, by envisaging that escapes in their content, these strings may represent formatting and layout operations as well as individual characters.

In the aforementioned code, we are using **'+'** as the concatenation operator. It is observable that how difficult it is to read a simple JSON object.

We must improve both the readability and the writability of a large string in the program. These strings may span across multiple lines and thus, without the visual clutter of escapes, one can easily be able to read them. This means that we need a two-dimensional block of text instead of a one-dimensional sequence of characters. What we need here, is a *text block.*

Scenario 2: Handling the string embedded XML + operator

We can also declare XML as a part of the String. Normally such XML declarations need to be declared in multiple lines because of the complexity of the tags that are part of XML structures. In such cases, we need to often use the **"+"** concatenation operator between multiple lines, as shown in the following code **Listing 6.2, TestStringWithConcatationOperatorXML.java**:

```
//Listing 6.2
package com.java15;

public class TestStringWithConcatationOperatorXML{
    public static void main(String[] args) {
       String stringxml = "<XML>\n"
                  + "     <language>\n"
                  + "         <name>Java </name>\n"
   + "         <version>\0 </version>\n"
```

```
                    + "     </language>\n"
                    + "</XML>";
        System.out.println(stringxml);
    }
}
```

As you can observe, the aforementioned code looks complex to read because of a lot of concatenation operators being used between the multiple lines.

Scenario 3: Handling the string embedded HTML + operator

We can initiate the string with the HTML, as shown in the following code **Listing 6.3, TestStringWithConcatationOperatorHTML.java**:

```
//Listing 6.3
package com.java15;

public class TestStringWithConcatationOperatorHTML{
    public static void main(String[] args) {
        String stringhtml = "<html>\n"
                    + "     <body>\n"
                    + "         <p>Welcome to Java \0</p>\n"
                    + "     </body>\n"
                    + "</html>";
        System.out.println(stringhtml);
    }
}
```

Similar to earlier scenarios, you can see that you need to use the + operator to join the multiple lines to generate the single string.

Scenario 4: Handling the string embedded HTML using StringBuilder

Instead of String, we can use the StringBuilder to initiate the HTML. The following code **Listing 6.4, TestStringWithStringBuilderHTML.java** shows the usage of using StringBuilder:

```
//Listing 6.4
```

```
2. package com.java15;
3.
4. public class TestStringWithStringBuilderHTML{
5.     public static void main(String[] args) {
6.        String stringhtml = new StringBuilder()
7.              .append("<html>\n")
8.              .append("     <body>\n")
9.              .append("         <p>Welcome to Java \0</p>\n")
10.             .append("     </body>\n")
11.             .append("</html>")
12.             .toString();
13.       System.out.println(stringhtml);
14.    }
15. }
```

This code is obviously more readable than using String. You can see that instead of the **"+"** operator, we can use the chained **append()** method and go on modifying the existing **StringBuilder** object.

Scenario 5: Handling the string embedded HTML using the String.format(string) function

Another way of declaring the String on multiple lines is to use the **format()** method of String. Observe the following code **Listing 6.5, TestStringWithStringFormatHTMl.java**:

```
1. //Listing 6.5
2. package com.java15;
3.
4. public class TestStringWithStringFormatHTML{
5.     public static void main(String[] args) {
6.        String stringhtml = String.format("%s\n%s\n%s\n%s\n%s"
7.             , "<html>"
8.             , "     <body>"
9.             , "         <p>Welcome to Java \0</p>"
```

```
            , "     </body>"
            , "</html>");
        System.out.println(stringhtml);
    }
}
```

The **`format()`** method takes the format specifiers as the first argument, and then you can go on to add the comma-separated values, which match the specifiers declared in the first argument. This approach is useful, when you want to make use of multiple formats in the single string. But it is a bit complex to read and maintain.

Scenario 6: Handling the string embedded HTML using StringWriter

We can also use the **`StringWriter`** object to wrap the HTML content within it. The following code **`Listing 6.6, TestStringWithStringWriterHTML.java`** shows the usage of the same:

```
//Listing 6.6
package com.java15;

import java.io.*;
public class TestStringWithStringWriterHTML{
   public static void main(String[] args) {
    StringWriter sw = new StringWriter();
    PrintWriter pw = new PrintWriter(sw);
    pw.println("<html>");
    pw.println("     <body>");
    pw.println("         <p>Hello, World</p>");
    pw.println("     </body>");
    pw.println("</html>");
    String stringhtml = sw.toString();
    System.out.println(stringhtml);
   }
}
```

Scenario 7: Handling the string embedded HTML using String.join()

Another way to declare multiple lines is to use the **`join()`** method of the String object. The following code **`Listing 6.7, TestStringWithJoin.java`** shows how to use this method to declare the HTML text in multiple lines:

```
1. //Listing 6.7
2. package com.java15;
3.
4. public class TestStringWithJoin{
5.     public static void main(String[] args) {
6.        String stringhtml =String.join("\n"
7.                , "<html>"
8.                , "     <body>"
9.                , "         <p>Welcome to Java \0</p>"
10.               , "     </body>"
11.               , "</html>");
12.       System.out.println(stringhtml);
13.    }
14. }
```



One can therefore use the text block to write multiline strings, HTML, JSON, or XML code blocks with more effectiveness and less complexity, so as to increase readability and writability.

Problem

How to handle multiline strings containing JSON, XML, or HTML?

Solution

We will walk you through different scenarios to understand how we can use the concept of the text block from Java 15 onwards.

Scenario 1: Handling the string embedded JSON

The following code snippet shows how we can declare the JSON text using the

text block utility, as opposed to the **"+"** operator we used previously. The code **Listing 6.8**, shows the difference in declaration between the preceding traditional approach and the modified approach of text blocks:

```
1. //Listing 6.8
2. package com.java15;
3. 
4. public class TestStringWithTextBlockJSON{
5.     public static void main(String[] args) {
6.   //Traditional way
7.       /* String stringJSON = "{\r\n"
8.           + "\"Name\" : \"Java\","
9.           + "\"RollNO\" : \"20\"\r\n"
10.          + "}";
11.        */
12.    //Using Text blocks
13.      String textJSON = """
14.             {"name" : "Java", \"RollNO" : "20"}""";
15.     System.out.println(textJSON);
16.     }
17. }
```

Explanation

As discussed earlier, in Java, when we want to embed a snippet for HTML, XML, SQL, or JSON in a string literal enclosed in double quotes (**"…"**), it requires a significant amount of editing with a lot of escapes along with concatenation. This embedding makes the snippet often difficult to read and maintain.

The Java language can be regularized by accepting that strings may be large enough to span multiple lines of a source file, and by envisaging that the escapes in their content may represent formatting and layout operations as well as individual characters.

This makes large code much complex. This means that there is a need to improve both the readability and the writability of a large class for the coding to

have a linguistic mechanism, where denoting strings is literally more feasible literally than as a string literal. These strings may span across multiple lines and so, without the visual clutter of escapes, one can easily be able to read that. It means that we need a two-dimensional block of text instead of a one-dimensional sequence of characters. What we need here a text block.

Scenario 2: Handling the string embedded HTML

The following code **`Listing 6.9, TestStringWithTextBlockHTML.java`**, shows the use of a text block for declaration of HTML on multiple lines:

```
1. //Listing 6.9
2. package com.java15;
3.
4. public class TestStringWithTextBlockHTML{
5.     public static void main(String[] args) {
6.   //Traditional way
7.      /*String stringhtml = "<html>\n"
8.                  + "     <body>\n"
9.                  + "         <p>Welcome to Java \0</p>\n"
10.                 + "     </body>\n"
11.                 + "</html>"; */
12.  //Using Text blocks
13.     String texthtml = """
14.       <html>
15.                         <body>
16.                             <p>Welcome to Java \0</p>
17.                         </body>
18.                     </html>""";
19.     System.out.println(texthtml);
20.     }
21. }
```

Scenario 3: Handling the string embedded XML

The text block also simplifies the XML declaration as shown in the following

code **Listing 6.10, TestStringWithTextBlockXML.java**:

```
1. //Listing 6.10
2. package com.java15;
3.
4. public class TestStringWithTextBlockXML{
5.     public static void main(String[] args) {
6.      //Traditional Way
7.         /*String stringxml = "<XML>\n"
8.                  + "     <language>\n"
9.                  + "         <name>Java </name>\n"
10.        + "         <version>\0 </version>\n"
11.                 + "     </language>\n"
12.                 + "</XML>";
13.         */
14.         //Using Text blocks
15.      String stringxml= """
16.                          <XML>
17.                           <language>
18.                             <name>Java</name>
19.                             <version> \0</<version>
20.                           </language>
21.                          </XML>""" ;
22.        System.out.println(stringxml);
23.     }
24. }
```



One can use text blocks to write multiline strings, HTML, JSON, or XML code blocks with more effectiveness and less complexity, so as to increase readability and writability.

Problem

When we use a text block to initialize a string containing JSON or HTML, will it support string functions such as **`length()`** or **`indexOf()`**?

Solution

The code snippet **`Listing 6.11`**, shows the use of different String functions that can be used for the String which is declared by using the text block feature:

```
1. //Listing 6.11
2. package com.java15;
3.
4. public class TestTextBlockStringFunctions {
5.   public static void main(String[] args) {
6.    String textJSON = """
7.      {"name" : "Java", \"RollNO" : "20"}""";
8.   System.out.println("Original text :-"+textJSON);
9.              System.out.println("Contains    java:     "    +
   textJSON.contains("java"));
10.            System.out.println("indexOf     Java:     "     +
   textJSON.indexOf("Java"));
11.   System.out.println("Length of text: " + textJSON.length());
12.   }
13. }
```

Output

If you execute the preceding code, the output shown in *Figure 6.1* is displayed:

```
Original text :-{"name" : "Java", "RollNO" : "20"}
Contains java: false
indexOf Java: 11
Length of text: 34
```

Figure 6.1: *JSON with the Text Block*

Explanation

A text block is nothing but a constant expression of the type **`java.lang.String`**,

which is the same as a string literal. The content of a text block is processed by the Java compiler in the following three distinct steps:

1. The line terminators in the content are translated to LF, that is,\u000A.
2. The incidental white space surrounding the content, which can be added to match the indentation of the Java source code, is now removed.
3. The escape sequences in the content are interpreted.

This processed content is now recorded in the class file as a **`CONSTANT_String_info`** entry in the constant pool, similar to a string literal. This class file does not contain the information, whether an entry is derived from a text block or a normal string literal.

At the time of execution, the text block is evaluated to a String instance. In case we have two text blocks with the same processed content, then both of them will refer to the same instance of String. It means that the String literal and string text block are visually different, although internally, it is a string literal and supports all String functionalities.



A text block is evaluated to an instance of java.lang.String, which is the same as a string literal, and hence, supports all String characteristics and functionalities.

Problem

What functionalities are offered by the **`stripIndent()`**, **`translateEscapes()`**, and **`formatted(Object… args)`** methods?

Solution

Text Blocks are provided with different helper utilities or methods, which are discussed as follows:

`public String formatted()`

Let us start with the **`formatted()`** function, as shown in the following code **`Listing 6.12, TestForattedJSON.java`**:

```
1. //Listing 6.12
2. package com.java15;
3.
```

```
4. public class TestFormattedJSON {
5. public static void main(String[] args) {
6.   String textJSON = """
7.     {"name" : "%s",
8.      "age" : "%d",
9.      "contact": "%s"
10.    }""";
11.   System.out.println("String value before formatted()\n " +
                                textJSON);
12.   System.out.println("Applying formatted()function :-
                \n" + textJSON.formatted("Java15", 20, "12345"));
13.
14. }
15. }
```

Output

If you execute the preceding code, the output shown in *Figure 6.2* will be displayed:

```
String value before formatted()
 {"name" : "%s",
 "age" : "%d",
 "contact": "%s"
}
Applying formatted()function :-
{"name" : "Java15",
 "age" : "20",
 "contact": "12345"
}
```

Figure 6.2: *Using the formatted() function*

Explanation

JDK 1.5 offered the format method of String, so as to format the JSON as shown in the following code snippet:

```
1. String textJSON1 = """
2.            {"name" : "%s",
3.             "age" : "%d",
4.             "contact": "%s"
5.            }""";
6. System.out.println(String.format(textJSON1,  "Java  1.5",  20,
   "909090"));
```

In JDK 15, the **`formatted(Object… args)`** method had been introduced. This method has been added in Java 13 to the **`String`** class. The method helps in formatting the supplied arguments:

```
public String formatted(Object… args)
```

This preceding function formats the supplied arguments by using this string as the format string. This method is equivalent to the **`format(this, args)`** of String.

The method parameters are as follows:

- **`args`**: The arguments referenced by the format specifiers in this string.

The method returns the formatted string.

The **`formatted()`** method is flexible to use as it can be used to set up different values at a single line execution.

public String stripIndent()

This method returns a string, whose value is the string in which the incidental white spaces are removed from the beginning and from the end of every line.

Now, what is the incidental white space? Where does it get added? And, who adds that?

The incidental white space is often present in the text block for aligning the content with the opening delimiter.

Let us consider the following string:

```
String stringhtml = """
……..<html>
……..    <body>
……..        <p>Welcome to Java 15</p>
……..    </body>
……..</html>
…….""";
```

Now, the **stripIndent()** method treats the incidental white space as the indentation to be stripped. This method will produce a string that preserves the relative indentation of the string. Let us use **'|'** to indent the string, which will look as follows:

```
|<html>
|    <body>
|        <p> Welcome to Java 15</p>
|    </body>
|</html>
```

traslateEscapes()

This method translates the escape sequence as the name itself indicates. The following code **Listing 6.13,TestTranslateEscapes.java** shows the usage of **translateEscapes()**:

```
1. //Listing 6.13
2. package com.java15;
3.
4. public class TestTranslateEscapes {
5. public static void main(String[] args) {
6.   String text = """
7.           as\\\\df \\nThis is the new line having a tab\\t here
8.     """;
9.    System.out.println("The original text is as follow:\n" +
   text);
10.   text = text.translateEscapes();
11.    System.out.println("Text after applying the method is as
   follow:-
                            \n" + text);
12. }
13. }
```

Output

If you execute the preceding code, the output shown in *Figure 6.3* will be displayed:

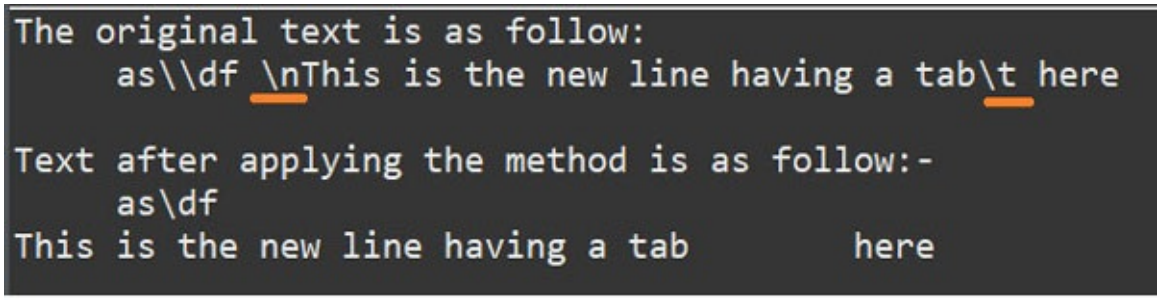


Figure 6.3: *Using translateEscapes()*

Explanation

The **`translateEscapes()`** method is used to translate the escape sequence in the given string.

The syntax of the method is as follows:

```
public String translateEscapes()
```

The method returns a string by translating the escape sequences in the original string.

The Unicode escapes are translated by the Java compiler while reading the input characters which are not part of the string literal specification. This method does not translate the Unicode escapes such as **`"\u2022"`**.

The translation of the escape sequence is as shown in the following *Table 6.1*:

Escape	Name	Translation
\b	Backspace	U+0008
\t	horizontal tab	U+0009
\f	form feed	U+000C
\r	carriage return	U+000D
\n	line feed	U+000A
\s	Space	U+0020
\"	double quote	U+0022
\'	single quote	U+0027
\\	Backslash	U+005C
\0 - \377	octal escape	code point equivalents
\<line-terminator>	Continuation	discard

Table 6.1: *Escape Sequences*



- The stripIndent() method allows you to strip away the incidental white space from the text block content.
- The method translateEscapes() allows you to translate escape sequences.
- The formatted(Object... args) method allows the value substitution in the text block.

Hidden classes

Java 15 provides the facility to use the services from the hidden class. When we do not want to expose the source code or the class in the class path, we can store the class file in some other location. Then, we can encode and decode that class and use the **`MethodHandle`** object to use the services from that class.

Problem

How to use the concept of the hidden class?

Solution

To understand this concept, we will create 2 classes.

`GreetMe.java` is the class that we would like to utilize as a hidden class. This is a simple class with one static and one instance method, as follows:

```
1. //Listing 6.14
2. package com.java15;
3.
4. public class GreetMe {
5.   public static String greet() {
6.     return "Welcome from hidden method from hidden class";
7.   }
8.   public String doTask(String task) {
9.     return "I am done with "+task;
10.  }
11. }
```

To access this class, we need to write another class, which is shown in the following code **`Listing 6.15, GreetMeTest.java`**:

```
1. //Listing 6.15
2. package com.java15;
3.
4. import java.lang.invoke.MethodHandle;
5. import java.lang.invoke.MethodHandles;
6. import java.lang.invoke.MethodType;
7. import java.lang.reflect.Method;
8. import java.nio.file.Files;
9. import java.nio.file.Paths;
10. import java.util.Base64;
11. import java.util.logging.Logger;
12. public class GreetMeTest {
13. public static void main(String[] args) throws Throwable {
14.
15.   String hiddenfilePath =
          "D:/Java9+Recipes/Java15Demos/com/java15/GreetMe.class";
16.   byte[] bytes = Files.readAllBytes(Paths.get(hiddenfilePath));
17.
18.   //Generated byte data for the class file
19.   byte[] classByteData = Baseㄱɛ.getDecoder()
          .decode(Base64.getEncoder().encodeToString(bytes));
20.
21.   //Defining lookup() for hidden class and methods
22.   Class<?> proxyClass = MethodHandles.lookup()
23.     .defineHiddenClass(classByteData, true,
            MethodHandles.Lookup.ClassOption.NESTMATE).lookupClass(
24.   System.out.println("Class Name:= "+proxyClass.getName());
25.
26.   System.out.println("Methods in class:->");
27.   for (Method method : proxyClass.getDeclaredMethods()) {
```

```
28.     System.out.println("Method Name:= "+method.getName());
29.   }
30.
31.   //Invoking static method without creating instance
32.   MethodHandle handler = MethodHandles.lookup()
                .findStatic(proxyClass, "greet",
                    MethodType.methodType(String.class));
33.   System.out.println((String) handler.invokeExact());
34.
35.   //creating instance of hidden class
36.   Object object=proxyClass.getConstructor().newInstance();
37.   //invoking the method with argument
38.   Method method=object.getClass()
                .getDeclaredMethod("doTask",String.class);
39.     System.out.println(method.invoke(object,  "Learning  Hidden
    Class"));
40. }
41. }
```

Output

The following output shown in *Figure 6.4* will be displayed if you execute **`GreetMeTest.java`**:

```
Class Name:= com.java15.GreetMe/0x0000000800bb0840
Methods in class:->
Method Name:= greet
Method Name:= doTask
Welcome from hidden method from hidden class
I am done with Learning Hidden Class
```

Figure 6.4: *Working with the Hidden class*

Explanation

Java 15 introduced a very interesting feature called the "*Hidden Class*". It is a way to restrict the access of a particular class by the class loader. Such a class

cannot be directly used by the bytecode of other classes. You can also treat them as undiscoverable. This is similar to the lambda expressions that you have been using from JDK 8.0. When you use the lambda expression at runtime, the class is instantiated behind the scenes for you.

In JDK 15, the hidden class is created by invoking the **`defineHiddenClass()`** method. In the code mentioned under **`Listing 6.15`**, you can observe the following code snippet, which starts from line number 22:

```
Class<?> proxyClass = MethodHandles.lookup()
          .defineHiddenClass(classByteData, true,
            MethodHandles.Lookup.ClassOption.NESTMATE).lookupClass(
```

The **`defineHiddenClass()`** method retrieves the hidden class against the bytes provided to it in the argument.

The parameters of this method are as follows:

- **`bytes`**: These are the bytes that make up the class data, which is in the format of a valid class file, as defined by the Java Virtual Machine Specification.
- **`initialize`**: If the value is passed as true, then the class will be initialized.
- **`options`**: These are class options. It can have one of the values from the following:
    - **`NESTMATE`**: This specifies that a hidden class can be added to the nest of a lookup class as a nestmate.
    - **`STRONG`**: This specifies that a hidden class has a strong relationship with the class loader.

The method returns the lookup object on the hidden class with original and full privilege access.

Once the class is retrieved, you can get the name of the class and methods from that class by using the process of reflection.Once the class is derived, you can invoke the static or non-static methods of that class.

Invoking the static method

To invoke the static method, you need to first create the **`MethodHandle`** reference of the method that you want to invoke. You can observe this in line 33 in the following code:

```
MethodHandle handler = MethodHandles.lookup()
              .findStatic(proxyClass, "greet",
```

```
                MethodType.methodType(String.class));
```

The **`findStatic()`** method is used to locate the static method in the retrieved class.

The method arguments are as follows:

- **`Hiddenclass name`**: The name of the derived or retrieved class from the bytes provided.
- Static method name which needs to be located.
- Return type of the method.

Once you get the **`MethodHandle`** for the method, you can invoke it by using **`invokeExact()`** as shown in line number 34 of our **`Listing 6.15`**.

Invoking the non-static/instance method

The approach of invoking the non-static method is a bit different, as you need the object of the hidden class. In our code line number 37, it shows how to create an object type for the hidden class, as follows:

```
Object object=proxyClass.getConstructor().newInstance();
```

Once you get the object, you can use the reflection process to create a method reference to the method that you would like to invoke. Line number 39 shown in the following uses **`getDeclaredMethod()`** to connect the method reference to the specific reference:

```
Method method=object.getClass()
          .getDeclaredMethod("doTask",String.class);
```

You need to provide the method name and the method argument type to **`getDeclaredMethod()`**.

After you get the method reference for the specific method, you can invoke the **`invoke()`** method on it. Observe the code snippet shown in the following from our **`Listing 6.15`**:

```
method.invoke(object, "Learning Hidden Class")
```

You don't need to provide any method name here because the method reference is already connected to the specific method. You need to just provide the object name and the argument that you would like to pass to that method.

Improvement in the CharSequence interface

`CharSequence` is not new to Java developers. This interface has been available in

Java since its 1.4 version. For years, we have been using the various services from this interface through the implementor classes like **`String`**, **`StringBuffer`**, **`StringBuilder`**, **`CharBuffer`**, and so on.

Java 15 added one more method in this interface to make the behavior of these classes more flexible.

Problem

Which new method is added to the **`CharSequence`** interface in Java 15? Explain its usage.

Solution

`Listing 6.16`, **`CharSequenceUpdate.java`** shows the usage of the **`isEmpty()`** method introduced in Java 15:

```
1. //Listing 6.16
2. package com.java15;
3. 
4. import java.util.ArrayList;
5. import java.util.List;
6. 
7. public class CharSequenceUpdate {
8. 
9. public static void main(String[] args) {
10.   // TODO Auto-generated method stub
11.   List<String> studentList=createStudents();
12.   System.out.println("List of students :->");
13.   studentList.forEach((s)->System.out.println(s));
14. 
15. }
16. 
17. private static List<String> createStudents() {
18.   // TODO Auto-generated method stub
19.   List<String> students=new ArrayList<String>();
```

```
20.   String[] studentData= {"James","David","",null, "Alan"};
21.   for (String student : studentData) {
22.
23.    if(student!=null && !student.isEmpty()){
24.     students.add(student);
25.    }
26. }
27.   return students;
28. }
29. }
```

Output

The following output in *Figure 6.5* will be displayed if we compile and execute **`CharSequenceUpdate.java`**:

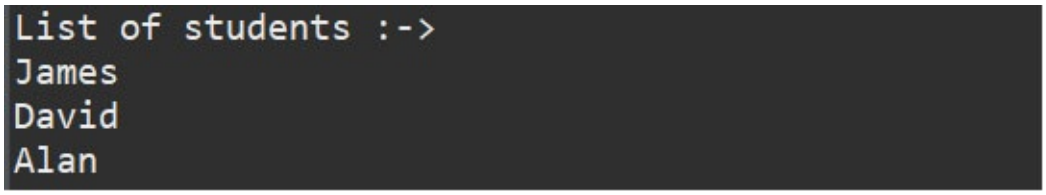


Figure 6.5: *Updates in CharSequence*

Explanation

Java 15 added another method in the **`CharSequence`** interface. Now, you can use the **`isEmpty()`** method to check whether the character sequence is empty.

The syntax of the method is as follows:

```
default boolean isEmpty()
```

The method returns true, if this character sequence is empty.

Internally, this method makes the use of the **`length()`** method. If the **`length()`** returns 0, then the method returns true, else returns false.

In our solution, that is, **`Listing 6.16`** at line number 23, we have used this method to check whether the array contains an empty string or not, as follows:

```
if(student!=null && !student.isEmpty()){
    students.add(student);
}
```



Using the isEmpty() method is more readable and maintainable than using length(). Specifically in scenarios where you would use string.length()==0, it is recommended to use string.isEmpty().

Conclusion

In this chapter, we discussed the features launched as a part of Java 15. We discussed about the text block, which is one of the major updates in this version as far as development is concerned. We discussed in depth how we can declare multi-lined strings, Html, and JSON in more flexible ways by using text blocks. We also discussed how to implement the hidden class concept and what are the different ways to invoke the static and non-static methods of the hidden class.

In the next chapter, we will learn how to implement the new features launched in Java 16.

Key terms

- format()
- stripIndent()
- translateEscapes()
- formatted()
- MethodHandles
- findStatic()
- invoke()
- invokeExact()
- isEmpty()

Questions

1. What is the need of text blocks?
2. What do you mean by the hidden class and how do you load it?
3. What are different methods to invoke the methods of the hidden class?
4. Explain the usage of the translateEscape() method.
5. Which new modification is done in the CharSequence in Java 15?

CHAPTER 7

Java 16 – Turning the Wheels

Introduction

While you all were working and understanding Java 15, the Oracle team was constantly working on their new release cycle. Following the time-based version release cycle of six months, Java 16 was released on March 16, 2021. This was said to be the last minor version before the **Long-Term Support** (**LTS**) version 17.

One of the major thought processes behind the changes recommended in this version towards the betterment was performance and the clean code approach. The concepts like the Record class, which was released as a production-ready feature in this version, reduce the boiler-plated code and instead help to create more concise and clean code. Similarly, the `instanceof` operator with pattern matching is introduced, which reduces the need for declaring the local variables inside the method. Apart from all this, there also had been also extensive updates in the Sealed classes, which was a preview feature in Java15. In this version also, this feature is kept as a preview feature. This chapter mainly focuses on production-ready features and not on preview features.

Structure

In this chapter, we will cover the following concepts:

- Default method invocation from a proxy
- Improved Date - Time API
- Modified Stream API
- Pattern matching for instanceof operator
- Record classes
- JPackage tool

Objectives

The aim of this chapter is to implement and learn the new features introduced in Java 16. In this chapter, we will discuss different concepts such as invoking the default method of interface from the proxy object, record classes, and improved Date - Time API. We will also discuss the interesting concepts of how to create the installer/packager using the jpackage tool introduced in this version.

Installation

Before you start reading this chapter, visit the following URL to download Java 16:

https://www.oracle.com/java/technologies/javase/jdk16-archive-downloads.html

Default method invocation from Proxy

As we all know from Java 8, we can have the default methods in an interface. Such default methods can be invoked by using proxy objects in Java 16.

Problem

How to invoke the default methods declared in the interface by using a proxy object?

Solution

Observe the following code snippet. **`Listing 7-1`** is an interface, **`InvokeDefaultMethodsFromPxyInstances.java`**, which contains one default method:

```
1. //Listing 7-1
2. package com.java16.proxymethod;
3.
4. interface InvokeDefaultMethodsFromProxyInstances {
5.   default String defaultFunction() {
6.    return "Default Function invoked";
7.   }
8. }
```

Listing 7-2, ProxyinstanceDemo.java shows how we can create the proxy instance for the interface declared in **Listing 7.1** and how to invoke its default method:

```
1. //Listing 7-2
2. package com.java16.proxymethod;
3. 
4. import java.lang.reflect.InvocationHandler;
5. import java.lang.reflect.Method;
6. import java.lang.reflect.Proxy;
7. 
8. public class ProxyinstanceDemo {
9.   public static void main(String[] args) {
10.    Object proxy = Proxy.newProxyInstance(
     ClassLoader.getSystemClassLoader(),
         new Class<?>[] {
     InvokeDefaultMethodsFromProxyInstances.class },
11.        (prox, method, arg) -> {
12.          if (method.isDefault()) {
13.            return InvocationHandler.invokeDefault(prox, method,
   arg);
14.          }
15.          return prox;
16. 
17.      });
18.    try {
19.                                    Method        method        =
   proxy.getClass().getMethod("defaultFunction");
20.     Object value = method.invoke(proxy);
21.     System.out.println(value.toString());
22.    } catch (NoSuchMethodException | SecurityException e) {
23.     // TODO Auto-generated catch block
24.     e.printStackTrace();
```

```
    } catch (IllegalAccessException e) {
     // TODO Auto-generated catch block
     e.printStackTrace();
    } catch (Exception e) {
     // TODO Auto-generated catch block
     e.printStackTrace();
    }
   }
}
```

Output

If you execute **`ProxyinstanceDemo.java`**, you will observe that the default method from the **`InvokeDefaultMethodsFromProxyInstance.java`** interface is invoked using a proxy.

Explanation

The developers can create and use proxy objects when they want to add a new or modify the existing functionality. As the name itself suggests, the proxy object is used instead of the original object. The proxy objects have the same methods as that of the original object. It has a handle to the original object and will be able to call the invoke method on it.

One can use **`InvocationHandler`** to perform proxy invocations. The invocation handler of a proxy instance implements the **`InvocationHandler`** interface. Each proxy instance has an associated invocation handler. Whenever a method is invoked on this proxy instance, the method invocation is encoded. After that, it is dispatched to the invoke method of its invocation handler.

Earlier to Java 16, the developers were using the **`invoke()`** method, which processes the method invocation on the proxy instance. Now in Java 16, we have the **`invokeDefault()`** method. This invokes the specified default method on the specified proxy instance:

The syntax of the method is as follows:

```
static Object invokeDefault(Object proxy,
                 Method method, Object… args) throws Throwable
```

The method arguments are as follows:

- **`proxy`**: The proxy instance on which the default method needs to be invoked.
- **`method`**: The method instance corresponding to a default method which is declared in the proxy interface of the proxy class or the method inherited from its super interface directly or indirectly.
- **`args`**: The parameters which are used for the method invocation. The parameters can be null if the number of formal parameters required by the method is zero.

The method returns the value returned from the method invocation.

The method invokes the specified default method on the given proxy instance with the specified parameters. We need to keep one thing in mind: the specified method must be a default method declared in a proxy interface of the proxy's class or it can be inherited from its super interface directly or indirectly.



The invokeDefault from the java.lang.reflect.InvocationHandler interface can be used to invoke a default method defined in a proxy interface.

Improved Date - Time API

There had been continuous efforts to make the Date -Time API more meaningful. Java 16 made some significant changes in this API, so as to make sure that the developer can generate more user-friendly Date and Time instances.

Problem

How do I use the Date - Time API to specify the value for the period of the day such as **`'in the morning'`** or **`'at night'`** instead of just specifying AM/ PM?

Solution

We can achieve this using the letter **`'B'`**, which can be used in the **`ofPattern()`** method.

Let us walk you through different scenarios to get the period of the day concept.

Scenario 1: To find the current time in the period of the day format

`Listing 7-3`, **`PeriodDateDemo.java`** shows the use of attribute ‘B' supplied to

`ofPattern()` to display the period of the day on the given date:

```
1. //Listing 7-3
2. package com.java16.dateformat;
3.
4. import java.time.Instant;
5. import java.time.OffsetDateTime;
6. import java.time.ZoneOffset;
7. import java.time.format.DateTimeFormatter;
8.
9. public class PeriodDateDemo {
10.
11.   public static void main(String[] args) {
12.  // TODO Auto-generated method stub
13.    DateTimeFormatter dateTimeFormat = DateTimeFormatter
                                          .ofPattern("hh B");
14.    for (int hour = ·; hour < ٢٤; hour+=٤) {
15.     final OffsetDateTime dateTime = Instant.now()
                        .atOffset(ZoneOffset.UTC).withHour(hour);
16.     System.out.println("Hour " + hour + ": \"" +

     dateTimeFormat.format(dateTime)+"\"");
17.    }
18.   }
19. }
```

Output

The following output shown in *Figure 7.1* will be displayed when we execute the **`PeriodDateDemo.java`**:

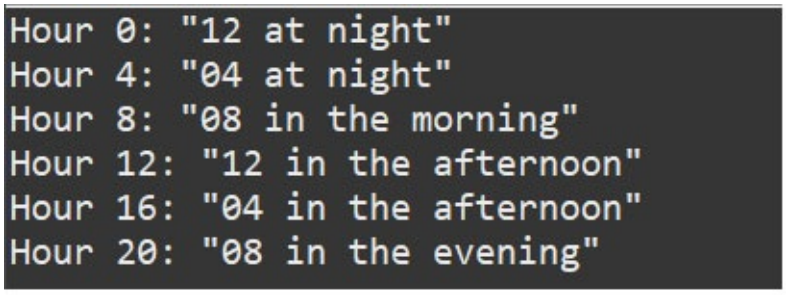


Figure 7.1: *Modified time with a period of day*

Explanation

In most of the date-related situations, we use or tend to use **ante meridiem** (**AM**) and **post meridiem** (**PM**) designators to describe the time. However, the Unicode documentation has described the rules ‘Day Period Rule Sets' for defining the period of the day. One can use ‘last night' or ‘at night' to specify 10:00 PM. This allows one to trace the situations such as the time of receiving the emails, couriers, code delivery, and so on, in proper formats such as, ‘*Your courier has arrived in the morning*,' instead of saying AM or PM.

A new formatter pattern letter **`‘B'`** along with its supporting methods has been added to the **`java.time.format.DateTimeFormatter`** and **`java.time.format.DateTimeFormatterBuilder`** classes. This pattern and its methods translate the period in the day defined in **`Unicode Consortium's CLDR`**. Using the updated methods now, the applications can express the periods in a day, such as **`‘in the morning'`**, **`‘at night'`**, and so on.

Scenario 2: To find the local time depending on the period of the day using the string provided

`Listing 7-4`, **`ParsingTimeDemo.java`** provides the solution to use the **`parse()`** method to generate the **`LocalTime`** and then pass it to the **`ofPattern()`** method to display the period of the day:

```
1. //Listing 7-4
2. package com.java16.dateformat;
3.
4. import java.time.LocalTime;
5. import java.time.format.DateTimeFormatter;
6.
7. public class ParsingTimeDemo {
```

```
8.
9.  public static void main(String[] args) {
10.    // TODO Auto-generated method stub
11.    LocalTime date = LocalTime.parse("١٥:٢٥:٠٨.٦٩٠٧٩١");
12.    DateTimeFormatter formatter = DateTimeFormatter.ofPattern("h
    B");
13.    String format=date.format(formatter);
14.    System.out.println("Time with only Hour =>");
15.    System.out.println("Fomatted Date String :-"+format);
16.
17.    formatter = DateTimeFormatter.ofPattern("hh  mm B");
18.    format=date.format(formatter);
19.    System.out.println("Time with Hour and Minute =>");
20.    System.out.println("Formatted Date String:-"+format);
21. }
22. }
```

Output

The following output in *Figure 7.2* is displayed when we execute **`ParsingTimeDemo.java`**:

```
Time with only Hour =>
Fomatted Date String :-3 in the afternoon
Time with Hour and Minute =>
Formatted Date String:-03 25 in the afternoon
```

Figure 7.2: *Modified Time with a period of the day using the parse() method*

Explanation

Here, we used the **`LocalTime.parse(CharSequence text)`** method to find the local time. Once we get the local time, we used the **`DateTimeFormatter.ofPattern()`** method to get the period of the day.

The syntax of the method is as follows:

```
public static LocalTime parse(CharSequence text,
                              DateTimeFormatter formatter)
```

The method arguments are as follows:

- **`text`**: The text to parse, not null.
- **`formatter`**: The formatter to use, not null.

The method returns instance of **`LocalTime`** from a character sequence, which is parsed using the formatter.

Scenario 3: To generate the hour, day period along with time zone details

`Listing 7-5`, **`PeriodWithZonedTimeDemo.java`** shows the process of generating hours along with the day period for the given date:

```
1. //Listing 7-5
2. package com.java16.dateformat;
3.
4. import java.time.Instant;
5. import java.time.ZoneId;
6. import java.time.ZonedDateTime;
7. import java.time.format.DateTimeFormatter;
8.
9. public class PeriodWithZonedTimeDemo {
10.
11. public static void main(String[] args) {
12.   // TODO Auto-generated method stub
13.   ZonedDateTime zonedDateTime = Instant.now()
                                          .atZone(ZoneId.systemDefa
14.     String  dateTimeStr  =  DateTimeFormatter.ofPattern("hh  B,
    zzzz")
                                       .format(zonedDateTime);
15.   System.out.println("Hour, Day Period, Zone: " + dateTimeStr);
16. }
17. }
```

Output

When we execute the **`PeriodWithZonedTimeDemo.java`**, we will get the output

as:

`Hour, Day Period, Zone: 10 in the morning, India Standard Time`

Explanation

In the previous two scenarios, we used the **`ofPattern()`** method for the local date instance. Along with this, we can also pass the zone details to this method. This will format the given time corresponding to the given zone.

In our solution at line number 14, we created the **`ZonedDateTime`** instance by invoking the method **`atZone()`** on the current instant. The **`ZoneId.systemDefault()`** generates the system's zone.

This zoned time is then passed to the **`format()`** method, which is chained to **`ofPattern()`**, as shown in line number 15.

To print the zone inside the **`ofPattern()`**, we need to provide **`"zzzz"`** specifier in the string, as shown in the solution.

Scenario 4: Getting the exact time and day period along with time zone details

We can use the **`ofPattern()`** method of **`DateTimeFormatter`** which accepts hours, minutes, the day period, and zone details, as shown in the following code **`Listing 7-6`**:

```
1. //Listing 7-6
2. package com.java16.dateformat;
3.
4. import java.time.Instant;
5. import java.time.ZoneId;
6. import java.time.ZonedDateTime;
7. import java.time.format.DateTimeFormatter;
8.
9. public class PeriodWithZonedTimeInHourMinuteDemo {
10.
11. public static void main(String[] args) {
12.   // TODO Auto-generated method stub
13.   ZonedDateTime zonedDateTime = Instant.now()
```

```
                                        .atZone(ZoneId.systemDefault(
14.   String dateTimeStr = DateTimeFormatter.ofPattern("K:mm B z")
                                        .format(zonedDateTime);
15.   System.out.println("Hour, Day Period, Zone: " + dateTimeStr);
16. }
17. }
```

Output

You will get the output as "**`Hour, Day Period, Zone: 10:26 in the morning IST`**", when you execute **`PeriodWithZonedTimeInHourMinuteDemo.java`**.

Explanation

In the second scenario, we saw how to display hours and minutes along with the period of the day. Then, in the third scenario, we learned how to display the information about the zone in the given time. In addition, we can also customize the output of the method **`ofPattern()`**, by modifying the string provided to it.

If we want to display the time in hours and minutes format along with the zone and period of the day, we can provide the specifier to the **`ofPattern()`** method as "**`K:mm B z`**" as shown in line number 14.

Scenario 5: Getting different hours to correspond to different day periods

We can use the **`ofPattern()`** method of **`DateTimeFormatter`** which accepts hours, minutes, the day period, and zone details which gives the desired output. We need to iterate this from 0th hour to 24th hour, as shown in **`Listing 7-7`**, **`PeriodThroughDayDemo.java`**:

```
1. //Listing 7-7
2. package com.java16.dateformat;
3.
4. import java.time.Instant;
5. import java.time.OffsetDateTime;
6. import java.time.ZoneOffset;
7. import java.time.format.DateTimeFormatter;
8.
9. public class PeriodThroughDayDemo {
```

```
10. public static void main(String[] args) {
11.    DateTimeFormatter dateTimeFormat = DateTimeFormatter
                                              .ofPattern("hh B");
12.    for (int hour = ٠; hour < ٢٤; hour++) {
13.     final OffsetDateTime dateTime = Instant.now()
     .atOffset(ZoneOffset.UTC).withHour(hour);
14.     System.out.println("Hour " + hour + ": \"" + dateTimeFormat
                              .format(dateTime) + "\"");
15.    }
16. }
17. }
```

Output

If you execute **`PeriodThroughDay.java`**, you will get all the hours in a day with different periods. The following *Figure 7.3* shows part of the output:

```
Hour 11: "11 in the morning"
Hour 12: "12 in the afternoon"
Hour 13: "01 in the afternoon"
Hour 14: "02 in the afternoon"
Hour 15: "03 in the afternoon"
Hour 16: "04 in the afternoon"
Hour 17: "05 in the afternoon"
Hour 18: "06 in the evening"
Hour 19: "07 in the evening"
Hour 20: "08 in the evening"
```

Figure 7.3: *Modified time throughout a day*



- Use the DateTimeFormatter.ofPattern() method to get the time in the form of a period of the day using the letter 'B'.
- To display Hours, Minutes, and so on, along with the period of the day, you can add 'h:mm' specifiers in the same String.
- You can also display the zone information by using the specifier 'zzzz' or 'z' to the ofPattern() method.

Modified Stream API

Java 16 provided an additional feature in the Stream API using which you can convert the existing stream to an unmodified list.

Problem

How to use the Stream API to collect the elements after filtering in a list in such a way that no further element can be added to that list?

Solution

We can use the **`toList()`** method of stream to create **`List`** from the existing stream. This List is unmodifiable as shown in **`Listing 7-8`**, **`StreamtoListDemo.java`**:

```
1. //Listing 7-8
2. package com.java16.stream;
3.
4. import java.util.Arrays;
5. import java.util.Comparator;
6. import java.util.List;
7. import java.util.stream.Collectors;
8.
9. public class StreamtoListDemo {
10.
11. public static void main(String[] args) {
12.   // TODO Auto-generated method stub
13.
14.     List<String> fruits = Arrays.asList("Banana", "Mango",
   "Apple",
                                          "Papaya");
15.   List<Integer> fruitlengthList = fruits.stream().map((fruit) -
   > {
16.                                                          return
   fruit.length();}).collect(Collectors.toList());
```

```
17.   System.out.println("Original List Generated (Java\0) :");
18.   System.out.println(fruitlengthList);
19.   fruitlengthList.add(\·);
20.   System.out.println("Modified List: " + fruitlengthList);
21.
22.   List<Integer> intsEquivalentOfFruits =
23.                       fruits.stream().map((fruit)->     {return
    fruit.length();}).toList();
24.   System.out.println("Original List Generated (Java\ㄱ) :");
25.   System.out.println(intsEquivalentOfFruits);
26.   intsEquivalentOfFruits.add(\·);
27.            System.out.println("Modified     List:     "     +
    intsEquivalentOfFruits);
28. }
29. }
```

Output

If you execute the preceding code, the following output is observed. Please make a note of the exception as shown in the output in *Figure 7.4*:

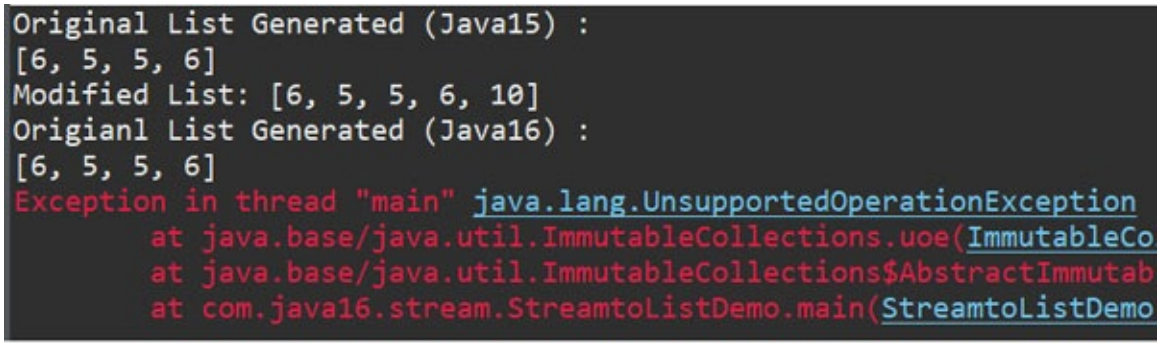


Figure 7.4: *Unmodifiable List from the Stream*

Explanation

Prior to Java 16, we were using Stream's **`collect(Collectors.toList())`** method to collect the element in an ordered fashion. Now, Java 16 provided a more concise syntax for collecting the data in the list for all frequent operations.

Though the code is more concise, we need to keep one thing in mind. The list returned by the **`toList()`** method of the stream is immutable and so cannot be further modified as well as sorted.

The signature of method is as follows:

```
default List<T> toList()
```

The method returns a list, containing the stream elements.

This is the terminal operation that can be performed on the stream to accumulate its elements into a list. The list returned by the method cannot be modified further and the elements will have the same order as that in the stream. Any calls to any kind of the mutator methods will always cause throwing of **`UnsupportedOperationException`** to be. We can have the returned instance containing value-based elements. The identity-sensitive operations on these instances are unreliable.



Use the toList() method to obtain the elements in the list format when you do not need any further addition or sorting of its elements.

Pattern matching in the instanceof operator

As a developer, it is necessary to understand the important role that `the` **`instanceof`** operator plays for all of us. We have used this operator to check the type of instance. One of the major drawbacks (unless you observe it as a drawback when you learn the **`instanceof`** operator in Java 16) of the traditional instance of the operator is the local variable type casting, which we need to use to extract the data in the method. The modified version of the **`instanceof`** operator in Java 16 helps you to write more readable and compact code for such boiler plated code of typecasting.

Problem

How can we use the **`instanceOf`** operator using pattern matching?

Solution

Listing 7-9 is a collection of different classes; **`Employee`**, **`Manager`**, and **`DemoInstanceOf`**, which is an illustration of the modified **`instanceof`** operator:

1. `//Listing 7-9`

```
package com.java16.instanceOf

class Employee {
private int empId;
        //default and parameterised constructor
        //getters, setters, toString functions
public double calculateSalary() {
  return 1000;
}
}

class Manager extends Employee {
double experience;
        //default and parameterised constructor
        //getters, setters, toString functions
@Override
public double calculateSalary() {
  double base_sal = super.calculateSalary();
  if (experience >= 5) {
   return base_sal + 5000;
  } else
   return base_sal + 2000;
}
}

public class DemoInstanceOf {
public static void main(String[] args) {
  Manager m = new Manager(12, 120);
  checkInstance1(m);
}
public static void checkInstance1(Employee e) {
```

```
33.   if (e instanceof Manager m) {
34.    System.out.println("I am manager:"+m.getExperience() +
      m.getEmpId()+":"+m.calculateSalary());
35.   }
36. }
37. }
```

Output

If you execute **`DemoInstance.java`**, you will observe that the information of the Manager is displayed on the console.

Explanation

The pattern matching performs checking whether an object has a particular type or not. Earlier versions of Java already have the **`instanceof`** operator. But now, pattern matching provides a more concise and robust way to enable conditional extraction of data from objects.

A pattern is a combination of a predicate which performs testing and a set of local variables, which is known as pattern variables. When the predicate matches successfully, this pattern variable will have the values which are obtained from the predicate. Now, the **`instanceof`** operator takes a type pattern instead of just a type.

In JDK 16, the pattern matching for the **`instanceof`** operator allows common logic in a Java program to be expressed more concisely and safely. The **`instanceof`** operator matches the specified object to the type provided. When the type of the specified object matches, then it is cast to that particular type. And after that, its value is assigned to the variable declared.

In our solution, you will observe line number 33 as: `if (e instanceof Manager m)`

In this case, if **`e`** is an instance of **`Manager`**, then we do not need any further casting. The local variable **`m`** is assigned as a type of **`Manager`**, which can be used to extract the data from the instance provided.



The updated instanceof operator will now perform the type checking and if the type matches, it will straightforwardly go with the casting process without any explicit casting operation.

Problem

How can we use the **`instanceof`** operator in the inheritance hierarchy, including its super class using pattern matching?

Solution

Let us modify the **`checkInstance1()`** definition from the earlier code snippet to check whether the instance is of the type **`Employee`**:

```
1. public static void checkInstance1(Employee e) {
2.    if (e instanceof Employee e1) {
3.     System.out.println("i am employee"+e1.getEmpId() +":"+
                                                e1.calculateSalary())
4.    } else if (e instanceof Manager m) {
5.     System.out.println("i am manager:"+m.getExperience() +
                              m.getEmpId() +":"+
     m.calculateSalary());
6.    }
7. }
```

This code will lead to "**`Expression type cannot be a subtype of the Pattern type`**" as a compile-time error. Thus, let us add another definition as follows:

```
1. public static void checkInstance(Object e) {
2.    if (e instanceof Employee e1) {
3.     System.out.println("I am an employee"+e1.getEmpId() +":"+
                                                e1.calculateSalary())
4.    } else if (e instanceof Manager m) {
5.     System.out.println("I am a manager:"+m.getExperience() +
                             m.getEmpId() +":"+
     m.calculateSalary());
```

6. }

This code works without any compilation error now.

Explanation

We straightforwardly cannot use the **`instanceof`** operator to compare against the super class type and that is the reason why we got an error on the first modification. But in the second modification, the argument of the method is changed to **`Object`** and not **`Employee`**. Now, considering Java fundamentals, we all know **`Employee`** is a subclass of **`Object`**.

Thus, the compiler can easily check whether the received instance is of type **`Employee`** or **`Manager`**, and so on.



- When we use the instanceof operator, we will get a compile-time error for a pattern where the instanceof expression is compared against its subtype.
- It is also important to understand the scope of the reference variable created in the instanceof operator used in the pattern matching. Such references are only available for the block for which it is initialized.
- Refer to the following example:

```
public void doSomething(Object o) {
  if (o instanceof String str) {
    // here, str is in scope
  }
  // str does not exist
}
```

Record classes

The primary building block of object-oriented programming is a class. We need to probably write hundreds of classes while we work with software development. While every class is used as a different model in different scenarios, quite often we need to write the boilerplate code as a part of the class. The constructors, accessor-mutator methods, and also the methods like **`toString()`**, **`equals()`**, and so on are utilized in almost every class. The record class is one of the new features launched in Java 16, which creates all such methods automatically.

Problem

How can we declare a record class?

Solution

The following line shows the basic structure of a record class declaration:

```
public record SimpleRecordClass(data_type argument1, ..) { }
```

Explanation

Now, Java has record classes as a new kind of class. The record classes are to support plain data aggregates. A record class is a declaration that primarily consists of a declaration of its state in terms of its data member declaration. When we declare a record class, we provide its name, optional type parameters, a header, and a body. The headers are nothing but the data members which define the state.

Each record will acquire the following functionalities:

- A canonical constructor whose signature is the same as the declared headers. It will assign each data member to the corresponding argument.
- A public accessor method for each declared header with the same name and return type as the component.
- The **`toString()`** method which will return a string representation of the record.
- The record will get equals and **`hashCode()`** methods for checking the two record values are equal or not.

The following *Figure 7.5* displays the methods for the class **`SimpleRecordClass`**:

```
public records.SimpleRecordClass(int);
public final java.lang.String toString();
public final int hashCode();
public final boolean equals(java.lang.Object);
public int value();
```

Figure 7.5: *Methods of Record Class*



The record classes are classes acting as data-holding classes with less boilerplate code, along with providing freedom of decoupling the class's API from its internal representation.

Problem

Is it possible to define a record class as a child of any other class, or it is acting as a super class of any other class?

Solution

Let us assume that you have some super class, and your record class tries to extend that class, as follows:

```
1. class Super_class{   }
2.
3. record Record_as_Super_Class(String name,int id)
        extends Super_class{   }
4.
5. class Sub_class extends Record_as_Super_Class{     }
```

This declaration is not permitted because of some special features of the record class.

Explanation

When we try to use the record class acting either as a super class or subclass, we will get an error. The record cannot act as a subclass because these classes are already extended by the **`java.lang.Record`** class. In the same way being the final class, they cannot act as a super class.

The following *Figure 7.6* provides a definition of the record class which will clear all our doubts:

```
final class records.Record_as_Super_Class extends java.lang.Record {
  records.Record_as_Super_Class(java.lang.String, int);
  public final java.lang.String toString();
  public final int hashCode();
  public final boolean equals(java.lang.Object);
  public java.lang.String name();
  public int id();
}
```

Figure 7.6: *Structure of a Record class*



- The record classes cannot extend any other class.
- The record classes are always final.

Problem

It is legal to implement one or multiple interfaces by a class. Can a record class also implement an interface?

Solution

The following code snippet shows how the record class can implement multiple interfaces:

```
1. interface Interface1 { }
2.
3. interface Interface2 { }
4.
5. record Record_interface_impl(long value) implements Interface1,
                Interface2, Serializable {    }
```

Explanation

A record class is allowed to implement anywhere between one to multiple interfaces as per the scenario.



A record class can implement any number of interfaces, which is the same as that of normal classes.

Problem

Is it possible to define extra instance fields in the record class apart from the headers declared in the definition?

Solution

Following code snippet shows how to add the instance field in the record class.

```
1. public record Record_extra_instance_fields(double data) {
2.     int instance_id;
3. }
```

Explanation

Whenever we will try to declare any extra instance field in the record class, we will get **"User declared non-static fields instance_id are not permitted in a record"** as an error.



The record class cannot declare extra data members apart from the header declaration.

Problem

When it comes to recording, what are the restrictions laid on headers?

Solution

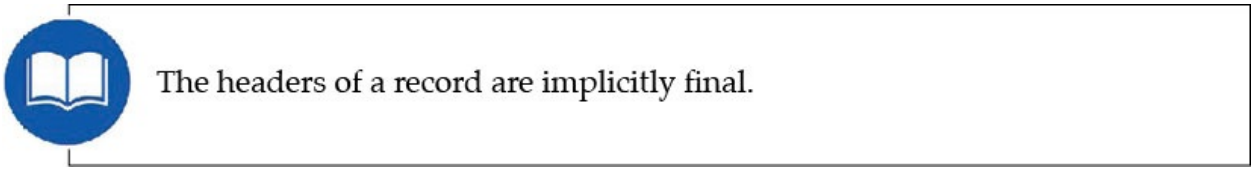
Consider the following record class:

```
1. public record Record_Header_Restrictions(String name,int value)
   {
2. String new_instance_name;
3. public void changeName(String name) {
4.   this.name=name;
5. }
6. }
```

This class fails at compilation because of some restrictions provided to the headers provided in the declaration of the record.

Explanation

As we already discussed, the record class cannot declare an extra instance field. In the same way, we cannot change the value of any header declared using mutator methods, as the headers are final. When we try to add the mutator method, we will get an error as, "**`The final field Record_Header_Restrictions.name cannot be assigned`**".



Problem

How can we declare a static field, block, and function in the record class?

Solution

The following code snippet shows the declaration and use of the static field, block, and function in the record class:

```
1. public record Record_static(String name, int value) {
2. private static int COUNT;
3. 
4. static {
5.   COUNT = 0;
6. }
7. 
8. public static void displayCount() {
9.   System.out.println("counter:-" + COUNT++);
10. }
```

Explanation

When it comes to static, it is very much similar to our normal classes. We can

declare a static field, static block for initializing the field, static block for initializing the static field, and static function, for some extra functionalities belonging to that class.



Declaration and use of a static field, block, and function is legal.

Problem

How can we instantiate in the record classes?

Solution

Let us consider the record class **`SimpleRecordClass.java`** having a single integer type header. The following code snippet shows how to instantiate it:

```
1. class TestSimpleRecordClass {
2. public static void main(String[] args) {
3.         SimpleRecordClass object=new SimpleRecordClass(10);
4. }
5. }
```

Explanation

The instantiation process of record classes is very much similar to normal classes. However, we need to keep one thing in mind. A normal class without any explicit constructor definition always gets the default constructor definition. However, the record classes will get a canonical constructor, which will assign all the private fields to the corresponding arguments. That is the reason why statement 3 is using the parameterized constructor and not default.



By default, the record class gets a canonical constructor for field initialization.

Problem

How can we define parameterized constructors in the record classes? What are the restrictions on them?

Solution

We will discuss this with a couple of scenarios.

Scenario 1: Defining the explicit canonical constructor

The following snippet shows the use of the constructor to initiate all the headers/fields declared in the record:

```
1. record Demo_explicit_constructor(int empId, String empname,
                                                        long
   salary) {
2. public  Demo_explicit_constructor(int empId, String empname,
                                                        long
   salary) {
3.   this.empId = empId;
4.   this.empname = empname.toUpperCase();
5.   this.salary = salary;
6. }
7. }
```

Explanation

We all know that in a normal class without any constructor, we get a default constructor automatically. However, in a record class, when there is no explicit constructor declared, it gets a canonical constructor automatically, which assigns all the private fields to the corresponding arguments of the new expression, which instantiated that record.

According to Java 16 when the developer declares a canonical constructor explicitly, then its access modifier must provide at least as much access as the record class. So, in this example, we declared an explicit canonical constructor with the public. We can also use the default access specifier instead of the public.



It is legal to have an explicit canonical constructor in the record class.

Scenario 2: Explicit canonical constructor with some/none field initialization

Let us add the following constructor definition instead of the earlier one:

```
public  Demo_explicit_constructor(int empId, String empname,
                            long salary) {   } // this will give
error
```

Explanation

The explicit canonical constructor should initialize all the fields of the class. In case it fails to do so, we will get **"`The blank final field value may not have been initialized`"** error. So, the code from the preceding snippet will not work.

Problem

Which are the legal access specifiers while declaring an explicit canonical constructor in the record class?

Solution

Let us discuss this with different scenarios. We will declare the record class with the default access specifier. Let us see what are valid or legal access specifiers for canonical constructors for the same:

```
1. record Demo_explicit_constructor1(int empId, String empname,
              long salary) {
2. //scenario 1
3. protected Demo_explicit_constructor1(int empId, String empname,
                                                    long salary) {
4.    this.empId = empId;
5.    this.empname = empname;
6.    this.salary = salary;
7. }
```

```
}

//scenario 2
record Demo_explicit_constructor2(int empId, String empname,
                                                    long salary) {
Demo_explicit_constructor2(int  empId,  String  empname,  long
salary) {
  this.empId = empId;
  this.empname = empname;
  this.salary = salary;
}
}

//scenario 3
record Demo_explicit_constructor3(int empId, String empname,
                                                    long salary) {
private Demo_explicit_constructor3(int empId, String empname,
                                                    long salary) {
  this.empId = empId;
  this.empname = empname;
  this.salary = salary;
}
}
```

Explanation

We can define a canonical constructor as having the same access specification as that of the class or more of the access specifier. However, it is not legal to use the restricted access specifier for canonical constructor definition. So, the code from scenario 3 will give an error.



The explicit canonical constructor is legal with the same or more broader access specifier; however, we cannot use restricted access specifiers.

Problem

How can we define overloaded constructors or non-canonical constructors in the record classes?

Solution

You can provide the overloaded constructor to the canonical constructor of record, as follows:

```
1. record Demo_explicit_constructor4(int empId, String empname,
                                                    long salary) {
2. public Demo_explicit_constructor4(int empId) {
3.   this(empId, "empname", 1234l);
4. }
5.
6. protected Demo_explicit_constructor4(int empId,String empname)
   {
7.   this(empId, empname, 1234l);
8. }
9.
10. protected Demo_explicit_constructor4(String empname) {
11.   this(0, empname, 1234l);
12. }
13.
14. protected Demo_explicit_constructor4(long salary) {
15.   this(0, "empname", salary);
16. }
17. }
```

Explanation

It is very usual to have multiple overloaded constructors to support various ways of instantiating a normal class. In the same way, a record class can have multiple non-canonical constructors. But we cannot overlook the fact that the fields of record classes are final, and so there can be no question of reinitializing them.

Thus, instead of directly going for initialization using this operator, we need to invoke the canonical constructor from the non-canonical constructor to perform the initialization task. It means that we can write the non-canonical constructors as shown in the following code:

```
1. public Demo_explicit_constructor4(int empId) {
2.   this(empId, "empname", 1234l);
3.   this.empId=empId;
4. }
5.
6. public Demo_explicit_constructor4(int empId) {
7.    this.empId=empId;
8. }
```



- The explicit non-canonical constructors are perfectly legal.
- The explicit non-canonical constructor is not allowed to initialize the declared headers.
- The explicit non-canonical constructor has to invoke the explicitly canonical constructor for the initialization using the custom values.

Problem

How to make the use of compact constructors for the field validation of a record class?

Solution

The following code shows the working of a compact constructor for a record class:

```
1. //Listing 7-10
2. package com.java16.records;
3.
4. import java.util.Objects;
5.
6. public record Demo_explicit_constructor6(int empId, String
```

```
    empname, long salary) {
7.  //compact constructor
8.  public Demo_explicit_constructor٦
9.  {
10.   Objects.requireNonNull(empname, "Needs employee name!");
11.   if (empId <= ٠) {
12.    throw new IllegalArgumentException("The empid must
                                              be a non-zero!!");
13.   }
14. }
15. public static void main(String[] args) {
16.
17.   Demo_explicit_constructor٦ demo =
              new Demo_explicit_constructor6(1, "name 1", 1234l);
18.   System.out.println("Instance value is:-" + demo);
19.   try {
20.    demo = new Demo_explicit_constructor١-)٦, "name ١٢٣٤ ,"١l);
21.    System.out.println("Instance value is:-" + demo);
22.   } catch (Exception e) {
23.    // TODO: handle exception
24.    e.printStackTrace();
25.   }
26.   demo = new Demo_explicit_constructor١)٦, null, ١٢٣٤l);
27.   System.out.println("Instance value is:-" + demo);
28. }
29. }
```

Output

The following output shown in *Figure 7.7* is displayed when we execute the preceding code:

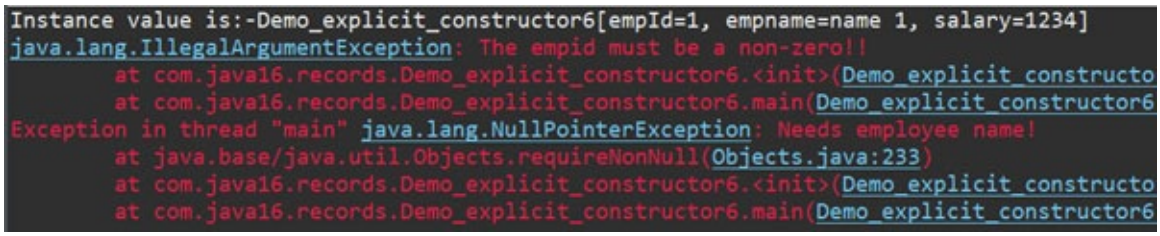


Figure 7.7: *Compact Structure in Record class*

Explanation

We have previously discussed about the canonical constructor. Let us try to understand what limitation we face when we use the canonical constructor to perform validation of fields of a record.

To check this, let us add the explicit canonical constructor, as follows:

```
1. public record Demo_explicit_constructor6(int empId, String empname,
                                                    long salary) {
2. public Demo_explicit_constructor6(int empId, String empname,
                                                 long salary) {
3.    if (Objects.nonNull(empname)) {
4.     this.empname = empname;
5.    }
6.    this.salary = 0;
7.    this.empId = empId;
8. }
9. }
```

When we add such a constructor, we will get **`"The blank final field empname may not have been initialized"`** as an error, as **`empname`** is not yet initialized. We are trying to use it in the validation process prior to initialization.

This implies that performing straightforward validation in the canonical constructor is a big headache. This issue can be resolved using the compact constructor, as shown in the preceding code.



Use a compact constructor to perform encryption, decryption or validation, and so on.

Problem

Can we define accessor and instance methods in the record class?

Solution

The following code snippet shows how to use the accessor and instance methods in a record class:

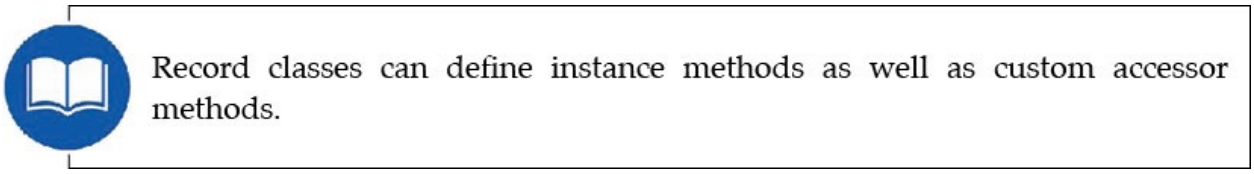
```
1. //Listing 7-11
2. package com.java16.records;
3.
4. public record Demo_record_instance_methods(int empId, String
     empname, long salary) {
5. //inbuilt getter method of record
6. public long salary() {
7.   return salary < ١٠٠٠٠ ? ٩٩٩٩ : salary;
8. }
9.
10. //customized getter method
11. public long getSalary() {
12.   return salary < ١٠٠٠٠ ? ٩٩٩٩ : salary;
13. }
14.
15. //instance method
16. public void display() {
17.   System.out.println("name:-" + empname + "\t" +
              "id:-" + empId + "\t" + "having salary:-" + salary);
18. }
19.
20. }
```

Explanation

Adding the instance method is very much common to record classes, as we can do it with normal classes. When we talk about the accessor method, the record

class provides the default accessor method. The default accessor method is shown in line number 6. This method is just a header or fieldname, followed by the rounded bracket.

Sometimes, the default accessor method may not fulfil the actual requirements. Under such conditions, we can add the custom accessor method, even though it is quite possible to customize the behaviour of the predefined accessor method by writing an explicit one. One should not forget to ensure that the signature of the custom accessor method must be same as that of the predefined method. One can also think of writing a completely new method to access the fields, instead of customizing the defined ones. However, we will try to avoid it as it is unnecessary.



Problem

How to define record classes inside a class?

Solution

We can declare the record class as an inner class. We can declare it within another class or we can declare it as a method-local inner class. Let us discuss both these scenarios.

Scenario 1: Record class declared locally inside a class

The following code snippet shows the nested record class declared inside another class:

```
1. public class Demo_nested_record {
2. record Nested_Record(int value) {
3.   void display() {
4.    System.out.println("value:-" + value);
5.   }
6. }
7.
```

```
public static void main(String[] args) {
  Demo_nested_record.Nested_Record n = new Nested_Record(100);
  n.display();
}
}
```

Scenario 2: Record class declared locally inside a method

The following code snippet shows the nested record class declared inside a method:

Solution

```
public int method_local_record(int experience) {
  record Nested_Record1(int experience) {
   int calculate_incentive() {
    if(experience<=2)
     return 5000;
    return 7000;
   }
  }

  Nested_Record1 n=new Nested_Record1(experience);
  int incentives=n.calculate_incentive();
  return incentives;
}
```

Scenario 3: Record class declared locally inside another record class

We can also declare the record inner class inside another record class, as shown in the following code:

```
record Outer_record( int value) {
static int data=100;

record Inner_Record(int value1) {
```

```
  public void display_Inner_Record() {
   System.out.println("value1:-"+value1);

   System.out.println("value of static member is:-"+data);
   static_function();
  }
}

public void display_Outer_Record() {
  System.out.println("value1:-"+value);
}
public static void static_function() {
  System.out.println("accessing static method");
}

}
```

Explanation

It is quite possible to define record classes inside a class. In the normal classes, when it comes to nesting of a class, we can define them as either static or non-static. But the record classes when defined inside a class are always static. This is shown in the first scenario.

A method local record class is very much similar to the method local inner class, as shown in the code snippet of the second scenario.

We can define nested record classes with the same ease as that of nested classes. Whenever we need to access the inner classes, we need to provide the reference of the outer class. In the same way for nested records also, reference of the outer record needs to be provided.



- It is legal to define record classes inside a class, a method, or inside another record.
- Implicitly, the nested record classes are static.
- Nested record classes cannot refer to the non-static fields and methods of outer class members.

Problem

How can we use record classes as a member of the inner class?

Solution

The code shown in **`Listing 7-12`** shows the use of record classes as a member of the inner class:

```
1. //Listing 7-12
2. package com.java16.records;
3.
4. record Employee(int empId, String empname, long salary) {
5. }
6.
7. class OuterClass {
8. class InnerClass {
9.   Employee e;
10.
11.   public InnerClass() {
12.    // TODO Auto-generated constructor stub
13.    e = new Employee(\, "emp name", \···l);
14.   }
15.
16.   public void display() {
17.    System.out.println("employee :" + e);
18.   }
```

```
}
}

public class DemoRecord_Inner_class {
public static void main(String[] args) {
  // TODO Auto-generated method stub
  OuterClass.InnerClass innerClass = new OuterClass()
                                          .new InnerClass();
  innerClass.display();
}
}
```

Output

When you execute **`DemoRecord_Inner_class.java`**, you will get the output as information of the record class as follows:

```
Employee[empId=1, empname=emp name, salary=1000].
```

Explanation

The earlier version of Java 16 was not allowed to define record classes as members of the inner class. However, according to JEP 395, record classes can now be members of inner classes.



Now, one can declare a record class as a data member of inner classes.

Problem

How to declare a generic record class?

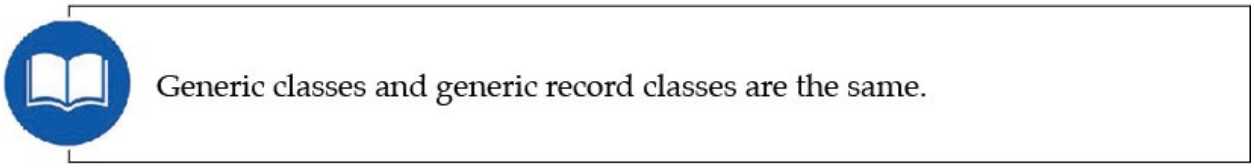
Solution

To explain this, let us create a hierarchy as **`Employee->WageEmployee`**. Once we have this hierarchy, we can create a generic record class containing the field which is a subtype of **`Employee`** as shown in the following code:

```
1. class Employee {   }
2.
3. class WageEmployee extends Employee {    }
4.
5. class Manager extends Employee {   }
6.
7. record Generic_Record<E extends Employee>(E employee) {    }
```

Explanation

All of us already know how the generic class worked in different programming languages. Generic record classes too work exactly the same as other generic classes. Line number 7 from the preceding code snippet shows how we can create the generic record class from Java 16 onwards.



Packager tool

Since its inception, Java lacked in providing proper utility to create the packaged setup for the application. Traditionally, we were required to use the jar utility provided by Java to execute Java files on native operating systems. Java 16 released the Packager tool for compiling, packaging, signing, and deploying Java and JavaFX application.

Problem

If our application is ready for production, can we deliver it as an installable package suitable for the native platform?

Solution

Let us first develop a Java application which we will further package using the Java tool:

```
1. public class MyFrame {
```

```
2.
3. public static void main(String[] args) {
4.
5.         JFrame frame = new JFrame("Packaging");
6.         frame.setMinimumSize(new Dimension(400, 400));
7.         frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
8.
9.          JLabel lblText = new JLabel("Hello !!!!Welcome to Java
   exe
                                    packaging!",
     SwingConstants.CENTER);
10.         frame.getContentPane().add(lblText);
11.
12.         frame.pack();
13.         frame.setVisible(true);
14.
15.     }
16.
17. }
```

Let us now package it together as a JAR after a successful compilation using the following command:

```
jar cvf FrameApp.jar MyFrame.class
```

Once we have the jar, let us use the tool to create a platform specific executable file named as follows

```
jpackage -i . -n MyFrame --main-jar FrameApp.jar --main-class
MyFrame.
```

After the execution of the command, we will get an executable file named **`MyFrame-1.0.exe`**

This file now can be used as installed by just double clicking on it. You can find the installation at **`C:/Program Files`**.

Explanation

The jpackage tool was introduced as an incubating tool in JDK 14 by JEP 343. Currently in JDK 16, it is ready to be promoted to a production-ready feature. To

support this transition, the name of the jpackage module will change from **`jdk.incubator.jpackage`** to **`jdk.jpackage`**.

The legacy **`JavaFX`** javapackager tool supports the following:

- Native packaging formats to give the end users a natural installation experience. It includes formats such as msi, exe on Windows, pkg, dmg on macOS, and deb, rpm on Linux.
- Specifying launch-time parameters at packaging time.
- For invocation directly from the command line, programmatically, or via the ToolProvider API.

After application development for the distribution purpose, creating a JAR is a very common practice for many Java applications. However, the users are willing to install it on native platforms rather than simply placing them on the class path or the module path. It means now that it is just not sufficient for the application developer to deliver a simple JAR file; the developers must deliver it in the form of an installable package and that too, as one suitable for the native platform. In this way, Java applications will be distributed, installed, and uninstalled in an end user familiar way.

Value based classes

Java API now contains a new type of class termed as value based class. Along with wrappers, the classes like **`java.util.Optional`**, **`java.time.LocalDateTime`** are designed as value-based classes.

Problem

What are the value-based classes? How has the behaviour of wrappers changed because of this modification?

Solution

The following code snippet explains the modification in wrappers:

```
1. //Listing 7-13
2. package com.java16.wrapper;
3.
4. public class DemoWrapper {
```

```
5. public static void main(String[] args) {
6.   // TODO Auto-generated method stub
7.
8.   Integer i=٢٠;
9.   Integer i١=new Integer(٢٠);  //warning
10.
11.  synchronized (i) {  //warning
12.  }
13. }
14. }
```

Explanation

The primitive wrapper classes are now designated as value-based. Since JDK 9, the construction of such value-based classes has changed. Now, we can expect the warning for the following scenarios:

1. Since Java 9, the primitive wrapper class constructors have been deprecated for removal. Now, whenever we use the constructors in the source code **`javac`**, it produces removal warnings by default:

   ```
   Integer i1=new Integer(٢٠);
   ```

   On compilation of this preceding code, we will get the warning as follows:

   ```
   The constructor Integer(int) has been deprecated since version
   9 and marked for removal.
   ```

   We can identify the usage of deprecated APIs in binaries using the **`jdeprscan`** tool as follows:

   ```
   jdeprscan DemoWrapper.class
   ```

   We will get an output as follows:

   ```
   class DemoWrapper uses deprecated method
   java/lang/Integer::<init>(I)V(forRemoval=true)
   ```

2. We also get a warning while working with synchronization of wrappers. The **`javac`** identifies usages of the synchronized statement used with a value-based class type or any of the type, whose subtypes are all specified as value-based:

   ```
   synchronized (i) {

     }
   ```

On compilation of this preceding code, we will get the warning as following:

Integer is a value-based type which is a discouraged argument for the synchronized statement.



- The primitive wrapper classes are designated as value-based.
- The construction using their constructors is deprecated for removal along with prompting new deprecation warnings.
- New warnings are introduced for improper attempts to synchronize on instances of any value-based classes.

Conclusion

In this chapter, we discussed a few interesting features launched in Java 16. We learned many features with scenarios. We discussed how improving the Date Time API helps us to get more descriptive and meaningful time. Apart from the modified **`instanceof`** operator, we also talked about the record class in detail.

In the next chapter, we will walk through the features of Java 17.

Keywords

- invokeDefault()
- InvocationHandler
- ofPattern()
- LocalTime.parse()
- toList()
- UnSupportedOperationException
- instanceof
- record

Questions

1. What is the use of the invokeDefault() method?
2. How can you display the date with the period of the day?

3. How to create non modifiable list using features of Java 16?
4. Explain the use of modified instanceof operator used in Java 16.
5. What is scope of reference variables created by using modified instance of operator?
6. What is a record class and how to create it?
7. What are the different functionalities available in a record class?
8. Can a record class extend another record class?
9. Can a record class implement an interface?
10. What restrictions are applied on the headers of a record class?
11. What is a canonical constructor and how to declare it in a record class?
12. What is the behaviour of a record class when used inside an inner class?

CHAPTER 8

Java 17 – Journey is Not Over Yet

Introduction

Finally, the wait for another **Long-Term Support** (**LTS**) version is over for Java developers. The last LTS version was Java 11, which was launched in Sept 2018. After Java 11, many new features were introduced to make Java more reliable and robust. Java 17, which was launched in September 2021, comes with thousands of performances and security-related updates. Along with fourteen enhancements, this version also launched three incubator modules and one preview feature. This version provides a more meaningful way to generate random numbers. One of the important updates in the new random number generator algorithm is the ability to generate random numbers for parallel computing internally, which improves the performance of an application. Along with this, the version also provides an additional security feature for the deserialization process, which was missing from Java 9. Moreover, you can now use the switch-case statements in a more flexible way. In this chapter, we will learn the usage of all these features. Apart from these features which are released as the final release, we will also learn about the Vector API, which is still the incubator module. But in the near future, it will be definitely released as the final release because of its services of distributed computing. So, it is always a wise decision to learn this feature.

Structure

In this chapter, we will cover the following concepts:

- Improvements in Random Number Generator algorithms
- Deserialization filtering
- Modified switch case
- Reflection API for a sealed class
- Vector API

Objectives

The aim of this chapter is to get well-versed with Java 17 features. In this chapter, we will cover important features of the new random number generator algorithms, restricting the deserialization process by using the **`allowFilter()`** and **`rejectFilter()`** methods, and so on. We will also learn to integrate the reflection API on the sealed classes and interfaces, as well as this version's new switch case features. At the end, we will talk about the Vector API, which is still part of the incubator module in Java 17.

Installation

Before you start reading this chapter, visit the following URL to download Java 17:

https://www.oracle.com/java/technologies/javase/jdk17-archive-downloads.html

Improvements in the Random Number Generator Algorithm

Random numbers are used in many business algorithms. Earlier, we used legacy **Pseudo Random Number Generator** (**PRNG**) classes to generate random numbers. Though the API was rich in functionalities, there were some limitations observed in this API. One of the major concerns was the algorithms provided in the different classes were not easily interchangeable. Also, there was not much support to use streams along with these algorithms. To achieve this, Java 17 launched some additional interfaces and classes to this API. Along with this, the algorithms are now modified in such a way that they are very easily interchangeable.

Problem

Along with the legacy **`Pseudo Random Number Generator (PRNG)`** classes such as **`Random`**, **`ThreadLocalRandom`**, and **`SplittableRandom`**, what are the new ways available to obtain a random number?

Solution

`Listing 8.1, DemoRandomGenerator.java` shows the working of the RandomGenerator interface to generate the random number:

```
1. //Listing 8.1
2. package com.java17.random;
3. import java.util.random.RandomGenerator ;
4.
5. public class DemoRandomGenerator{
6.     public static void main(String[]args){
7.           System.out.println("Use of RandomGenerator interface
   =>");
8.       RandomGenerator generator = RandomGenerator.of("Random");
9.       int randomNumber=generator.nextInt();
10.      System.out.println("The generated random number :"
                                        +randomNumber);
11.    }
12. }
```

Output

As this is a random number generator, you will get different outputs for different executions. In this case, you may get the output shown in *Figure 8.1*:

```
Use of RandomGenerator interface =>
The generated random number :-881498324
```

Figure 8.1: *Using the RandomGenerator interface*

Explanation

Java already has legacy PRNG classes such as **`Random`**, **`ThreadLocalRandom`**, and **`SplittableRandom`**. However, we currently are facing the following difficulties:

- It is very difficult to replace classes such as **`Random`**, **`ThreadLocalRandom`**, and **`SplittableRandom`** with each another. Though they all support pretty much the similar set of methods, it is practically difficult even to replace them by one another.
- The **`Random`**, **`ThreadLocalRandom`**, and **`SplittableRandom`** classes have completely independent methods like **`nextInt()`**, **`nextDouble()`**,**`ints()`**, and **`longs()`**. However, if we look at the implementation within all these classes, we can see that they very much identical, as if they are almost

copy pasted.

- The literature has many PRNGs. However, some of them are not splitable, but are jumpable or some are leapable. The available PRNG implementations in Java makes it difficult to take advantages of the jumpable property.

The **`RandomGenerator`** interface is designed to provide a common protocol for objects that generate random numbers, pseudorandom sequences of numbers, or even the **`boolean`** values. One can implicitly or explicitly assign a specific range of values for random number generation. Each value would be chosen independently and uniformly from the specified range. This interface provides the methods to obtain a sequence by repeatedly invoking a method, which returns a single pseudorandomly chosen value. The interface also provides the method that returns a stream of pseudorandomly chosen values.

One of the interesting methods of this interface is the **`of()`** method.

The syntax of method is as follows:

```
static RandomGenerator of(String name)
```

The method returns an instance of **`RandomGenerator,`** which is dependent on the name of algorithm specified in the argument.

The method parameter is:

- **`name`**: name of the random number generator algorithm.

The supported algorithm names are **`L128X1024MixRandom`**, **`L128X128MixRandom`**, **`L128X256MixRandom`**, **`L32X64MixRandom`**, **`L64X1024MixRandom`**, **`L64X128MixRandom`**, **`L64X128StarStarRandom`**, **`L64X256MixRandom`**, **`Random`**, **`SplittableRandom`**, **`ThreadLocalRandom`**, **`Xoroshiro128PlusPlus`**, and **`Xoshiro256PlusPlus`**.



The new PRNG algorithms generation is fast, easier to implement, and appear to completely avoid the weakness of the currently available PRNG generation legacy classes.

Problem

How can we find information about all available random number generators initiated from **`RandomGeneratorFactory`**?

Solution

`Listing 8.2` shows the way to list and display all the random generators which are types of

`RandomGeneratorFactory`:

```
1. //Listing 8.2
2. package com.java17.random;
3. import java.util.random.RandomGenerator ;
4. import java.util.random.RandomGeneratorFactory ;
5. import java.util.stream.Stream;
6.
7. public class DemoRandomGeneratorFactory{
8.     public static void main(String[]args){
9.       System.out.println("List of available generators ==>");
10.      Stream<RandomGeneratorFactory<RandomGenerator>> stream=
    RandomGeneratorFactory.all();
11.      stream.forEach(value->System.out.println(value.name()));
12.    }
13. }
```

Output

The following output shown in *Figure 8.2.1* will be displayed if you execute the preceding program:

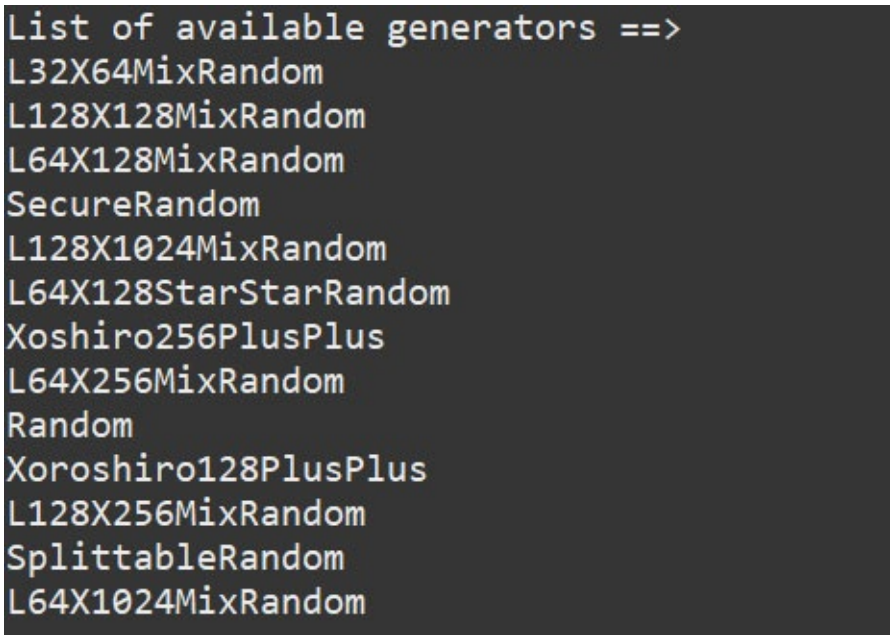


Figure 8.2.1: *The RandomFactoryGenerator factory class*

These generators are grouped as per their algorithm. If we execute the following statement:

```
RandomGeneratorFactory.all().map(factory->factory.group()+" - "

+factory.name()).sorted().forEach(System.out::println);
```

We will get the output as shown in *Figure 8.2.2*:

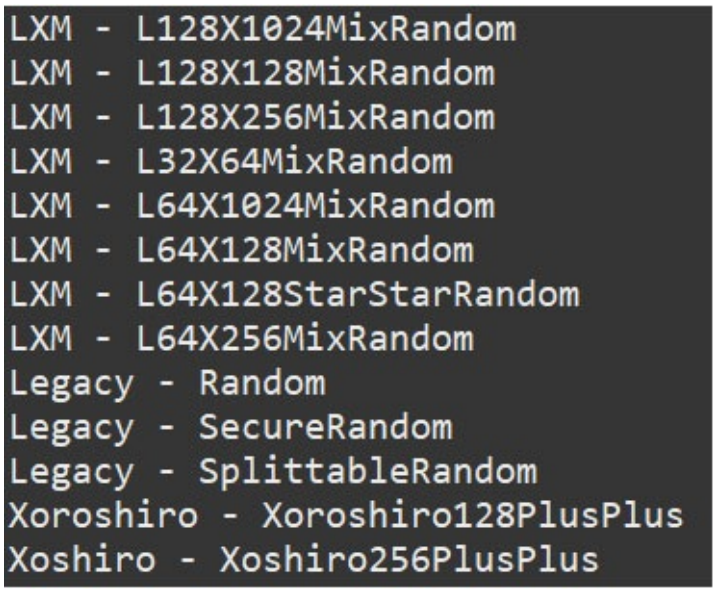


Figure 8.2.2: *The RandomFactoryGenerator factory grouping*

Here **`LXM`**, **`Legacy`**, **`Xoroshiro`**, and **`Xoshiro`** are the names of the factory group

in which the generators are stored.

Explanation

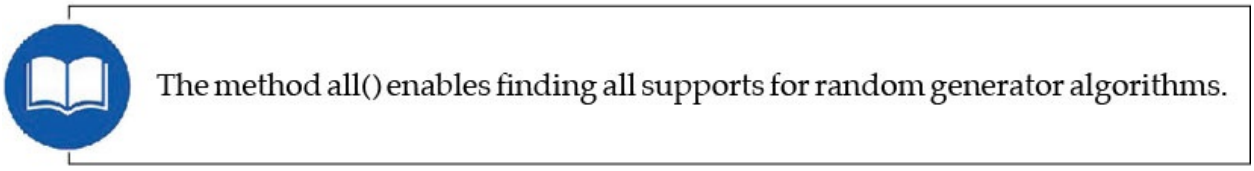
The **`RandomGeneratorFactory`** is a factory class for generating multiple random number generators of the specific algorithm. It provides methods for selecting random number generator algorithms by using the method **`of(String)`** which takes the argument as the name of the algorithm. It also provides the method **`all()`** which produces a non-empty Stream of all available **`RandomGeneratorFactory`**.

The syntax of the method is as follows:

```
public static Stream<RandomGeneratorFactory<RandomGenerator>> all()
```

The method returns non-empty stream of all the available **`RandomGeneratorFactory`**.

We have invoked this method in our **`Listing 8.2`** in line number 12. The name of the **`RandomGenerator`** generator can be retrieved by invoking the **`name()`** method, as shown in the code in line number 13.



Problem

What are the ways provided by **`RandomGeneratorFactory`** to get an instance of **`RandomGenerator`**?

Solution

`Listing 8-3, DemoRandomGeneratorFromFactory.java` shows the creation of a random number using the **`SplittableRandom`** algorithm using the **`of()`** method:

```
1. //Listing 8-3
2. package com.java17.random;
3. import java.util.random.RandomGenerator ;
4. import java.util.random.RandomGeneratorFactory ;
5. public class DemoRandomGeneratorFromFactory{
6.     public static void main(String[]args){
```

```
7.        System.out.println("Generating the SplittableRandom
                                                    Algorithm ==>");
8.        RandomGenerator generator=RandomGeneratorFactory
                                        .of("SplittableRandom").creat
9.        System.out.println("RandomGenerator Class:"+
                                        generator.getClass());
10.       int number = generator.nextInt();
11.       System.out.println("Generated Random Number:"+number);
12.
13.     }
14. }
```

Output

On executing the preceding program, the output shown in *Figure 8.3* is displayed:

```
Generating the SplittableRandom Algorithm ==>
RandomGenerator Class:class java.util.SplittableRandom
Generated Random Number:359414487
```

Figure 8.3: *Splittable using of()*



As this is a random number generator, you may get a different random number on your system.

Explanation

As we know, the **`RandomGeneratorFactory`** is a factory class for generating multiple random number generators of the specific algorithm. It provides three methods for constructing a **`RandomGenerator`** instance, depending on the type of initial seed required as discussed in the following:

`public T create(long seed)`

The method creates an instance of **`RandomGenerator`** depending on the algorithm chosen, and provides a starting long seed. In case, the long seed is not supported by the specified algorithm, no-argument form of create, that is, **`create()`** is used.

The method parameters:

- **seed**: Long random seed value.

The method returns a new instance of **`RandomGenerator`**.

`public T create(byte[] seed)`

The method creates an instance of **`RandomGenerator`** depending on the algorithm chosen and provides a starting byte[] seed. In case, the byte[] seed is not supported by the specified algorithm, no-argument form of create, that is, **`create()`** is used.

The method parameters:

- **seed**: **`byte[]`** array random seed value.

The method returns a new in instance of **`RandomGenerator`**.

The method throws **`NullPointerException`**, if the **`seed`** is null.

`public T create()`

The method creates an instance of **`RandomGenerator`** based on the algorithm specified.

The method returns a new instance of **`RandomGenerator`**.



The create() method of the RandomGeneratorFactory enables developers to get an instance of RandomGenerator.

Problem

How to use **`RandomGenerator`** to choose a PRNG algorithm for generating random integers, within a specific range?

Solution

The code **`Listing 8-4, DemoRandomGeneratorRange.java`** shows how to use the random PRNG algorithm to generate numbers between 0-15:

```
1. //Listing 8-4
2. package com.java17.random;
3.
4. import java.util.random.RandomGenerator;
5.
```

```
6. public class DemoRandomGeneratorRange {
7.   public static void main(String[]args){
8.    System.out.println("Generating random
                          numbers for the given range :");
9.    generateRandomNumberWithinRange(15);
10.  }
11.
12.  private static void generateRandomNumberWithinRange(int i) {
13.   // TODO Auto-generated method stub
14.   RandomGenerator generator = RandomGenerator.of("Random");
15.   int counter = ·;
16.   System.out.println("Random numbers in the range 0-"+i);
17.   while(counter<=\·){
18.     int number = generator.nextInt(i);
19.     System.out.println("Generated number:-"+number);
20.     counter++;
21.   }
22.  }
23. }
```

Output

As we are passing the value 15, for the function, you will observe that the random numbers from 0-15 are generated, as shown in *Figure 8.4*:

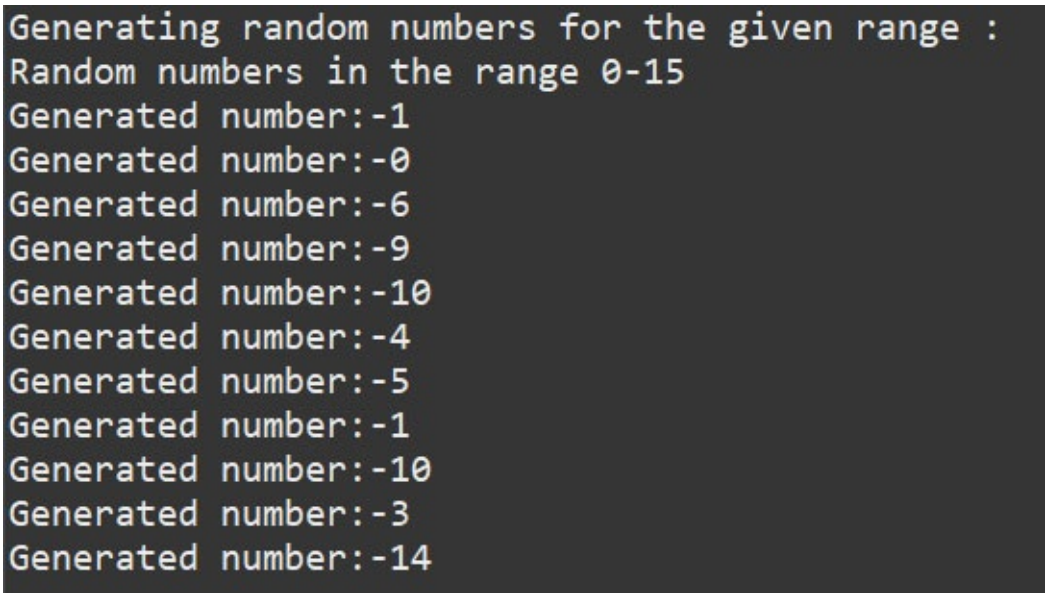


Figure 8.4: *Random Numbers in the range*

Explanation

`The RandomGenerator` interface has the **`nextXXX()`** method which returns a pseudorandomly chosen value of a specific type. Here, we have used the **`nextInt()`** method, which will return the int value. There are overloaded forms of this methods as well, which are discussed below:

`default int nextInt(int bound)`

The method returns a pseudorandomly chosen value of the type integer between zero and the specified bound (exclusive).

The method parameters:

- **bound**: The upper bound which must be a positive numeric value. The generated number will be exclusive of the specified bound value.

The method returns a pseudorandomly chosen value of type integer, between zero and the bound.

The method throws **`IllegalArgumentException`**, if the specified bound is not positive.

`default int nextInt(int origin, int bound)`

The method generates a pseudorandomly chosen value of the type integer between the specified origin (inclusive) and the specified bound (exclusive).

The method parameters:

- **origin**: The least value which can be returned.

- **Bound**: The upper bound (exclusive) for the value to return.

The method returns a pseudorandomly chosen integer value, between the origin (inclusive) and the bound (exclusive).

The method throws **`IllegalArgumentException`**, if the value of origin is greater than or equal to bound.

`default int nextInt()`

The method returns a pseudorandomly chosen value of the type integer. This does not take into consideration any origin or bound as in the earlier method declarations.

Let us now discuss how to use different algorithms to generate the random number. Keep in mind that the only difference is the way that we choose the algorithm. The rest of the code will remain the same as shown the **`Listing 8-4`**.

Case 1: Working with the SplittableRandom Algorithm

```
RandomGenerator generator=RandomGeneratorFactory
                                    .of("SplittableRandom").create();
```

Case 2: Working with the SplittableRandom Algorithm with a seed value

```
RandomGenerator generator=RandomGeneratorFactory
                               .of("SplittableRandom").create(100);
```



We can generate the random number in the range using the nextXXX() method which is provided as the parameter. The overloaded form of this method can also take another argument, using which, you can exclude the values from the generated group of random numbers.

Problem

How to use the **`split()`** method of **`SplittableGenerator`**?

Solution

`Listing 8-5, DemoRandomGeneratorThreads.java` shows the working of the **`split()`** method from **`SplittableGenerator`**:

```
1. //Listing 8-5
2. package com.java17.random;
3. import java.util.random.RandomGenerator ;
4. import java.util.random.RandomGenerator.SplittableGenerator ;
5. import java.util.random.RandomGeneratorFactory ;
```

```
import java.util.stream.IntStream;

public class DemoRandomGeneratorThreads{
    public static void main(String[]args){
       System.out.println("Generating the random numbers using
                                  SplittableGenerator ==>");
       generateAndSplit(5);
    }

    private static void generateAndSplit(int NUM_PROCESSES) {
     // TODO Auto-generated method stub
       RandomGeneratorFactory<SplittableGenerator> factory =
             RandomGeneratorFactory.of("L128X1024MixRandom");
                  SplittableGenerator splittableGenerator =
factory.create();

               IntStream intStream= IntStream.rangeClosed(\,
NUM_PROCESSES);

       intStream.parallel().forEach(p -> {
                RandomGenerator generator =splittableGenerator
.split();
       System.out.println(p + "-> Random Number: " +
          generator.nextInt(15)+ ", "
                                          +  "Thread  name:-
"+Thread.currentThread().getName());});

     }
}
```

Output

The following output in *Figure 8.5* is displayed when we execute the program:

```
Generating the random numbers using SplittableGenerator ==>
3-> Random Number: 11, Thread name:-main
5-> Random Number: 5, Thread name:-main
4-> Random Number: 9, Thread name:-main
2-> Random Number: 14, Thread name:-ForkJoinPool.commonPool-worker-1
1-> Random Number: 3, Thread name:-main
```

Figure 8.5: *Using the split() method*

Explanation

The **`SplittableGenerator`** interface is designed to provide a common protocol for objects, which generate the sequences of pseudorandom values. This sequence can be split into two objects where both objects have the same protocol. Both these objects are statistically independent of one another, individually uniform, and have the same statistical properties. The objects which implement **`RandomGenerator.SplittableGenerator`** are not cryptographically secure. So, one should use

`SecureRandom` to get a cryptographically secure pseudo-random number generator whenever the applications are security-sensitive.

The **`SplittableGenerator`** interface provides the **`split()`** method which is used to split the generated numbers:

The syntax of the method is as follows:

```
RandomGenerator.SplittableGenerator split()
```

The method returns a new pseudorandom number generator, split off from this one which implements the **`RandomGenerator`** and **`RandomGenerator.SplittableGenerator`** interfaces.

In our program, we have used:

```
IntStream intStream= IntStream.rangeClosed(1, NUM_PROCESSES);

    intStream.parallel().forEach(p -> {
          RandomGenerator generator =splittableGenerator .split();
    System.out.println(p + "-> Random Number: " +
          generator.nextInt(15)+ ", "
           + "Thread name:-"+Thread.currentThread().getName());});
```

The **`NUM_PROCESS`** argument in our solution has the value 5, which means that we are going to generate 5 random numbers. These 5 numbers will be from the range of 0 to 15, as the **`nextInt()`** method takes argument 15.

However, these 5 numbers will be split into different subtasks, as we have invoked the **`split()`** method, as shown in the output.



SplittableGenerator is a special kind of random generator, which is used to generate the random values that can be split. Such values can be used in parallel computations where each task is divided into the different subtasks.

Problem

What are the characteristics of **`RandomGeneratorFactory`** and how to find them?

Solution

The characteristics of **`RandomGeneratorFactory`** can be found out by invoking different methods on it, as shown in the following code, **`Listing 8-6, DemoRandomGeneratorFromFactoryCharacteristics.java`**:

```
1. //Listing 8-6
2. package com.java17.random;
3. import java.util.random.RandomGenerator ;
4. import java.util.random.RandomGeneratorFactory ;
5.
6. public class DemoRandomGeneratorFromFactoryCharacteristics{
7.    public static void main(String[]args){
8.         System.out.println("Characteristics of
                                RandomGeneratorFactory ==>");
9.         RandomGeneratorFactory<RandomGenerator> greatest =
10.            RandomGeneratorFactory
11.                .all()
12.                                        .sorted((f,   g)   ->
   g.period().compareTo(f.period()))
13.                .findFirst()
14.                .orElse(RandomGeneratorFactory.of("Random"));
15.        System.out.println("Name   :"+greatest.name());
16.        System.out.println("Period :"+greatest.period());
17.        System.out.println("Group  :"+greatest.group());
18.        System.out.println("Random Number :"+greatest.create()
```

```
                                                    .nextInt(15)
    }
}
```

Output

If you execute the preceding program, you will get the status of different characteristics of the **`RandomGeneratorFactory`**, as shown in *Figure 8.6*:

```
Characteristics of RandomGeneratorFactory ==>
Name   :L128X1024MixRandom
Period :61172327492847069472032393719205726809135813743440 7
Group  :LXM
Random Number :10
```

Figure 8.6: *Characteristics of RandomGeneratorFactory*

Explanation

`RandomGeneratorFactory` provides different methods, which return the characteristics of the specific implementor. This will facilitate to gather the information about the **`RandomGeneratorFactory`** at runtime. The following is a list of some important methods:

```
public String name()
```

The method returns the name of the algorithm, used by the random number generator.

```
public int stateBits()
```

The method returns the number of bits, used by the algorithm to maintain the state of seed.

```
public BigInteger period()
```

The method returns the period which is used by the random number generator algorithm. It returns **`BigInteger.ZERO`**, in case of the period is not determinable.

```
public String group()
```

The method returns String which indicates the group name of the algorithm used by the random number generator.

Deserialization filtering

We know that Java 9 had already provided the security implementations to

serialize the classes. But there were no restrictions on deserialization. JEP 415 from Java 17 now provides the mechanism to control the vulnerabilities in deserialization by applying different methods to **`ObjectInputFilter`**.

Problem

How to write a custom filter using a pattern for deserialization of an object?

Solution

Let us write a custom filter to set the maximum number of bytes in the input stream = 1024, and allowing classes from **`com.java17.filter`** as well as from the **`java.base`** module, but will reject all other classes.

The following code snippet from **`Listing 8-8`** shows how to apply the filter for the deserialization:

```
ObjectInputStream objectInputStream =
        new ObjectInputStream(inputStream);
ObjectInputFilter filter = ObjectInputFilter.Config.
 createFilter("maxbytes=1024;com.java17.filter.*;java.base/*;!*"
objectInputStream.setObjectInputFilter(filter);
try {
 Object object = objectInputStream.readObject();

 System.out.println("Deserialized Object : " + object);
} catch (ClassNotFoundException e) {
 e.printStackTrace();
} catch (IOException e) {
 e.printStackTrace();
}
```

Output

This code block applies a filter for the process of deserialization. As in the code repository, we have the serialized object in the **`com.java17.filter`**, and the deserialization happens without any issue.

But if we change the filter to deserialize only the objects from **`com.java.17.filter1`**.* as shown:

```
ObjectInputFilter filter = ObjectInputFilter.Config.
    createFilter("maxbytes=1024;com.java17.filter1.*;java.base/*;!*"
```

We will then get an exception, as shown in *Figure 8.7*:

```
java.io.InvalidClassException: filter status: REJECTED
```

Figure 8.7: *Using filters in Deserialization*

Explanation

To begin with, let us revise what we mean by serialization. According to the definition given by Oracle, *"Serialization is a process of converting an object state to a byte stream. So that the byte stream can be reverted back into a copy of the object."* In Java, whenever we want to serialize an object, either the class itself on any one of its parent classes must implement either the **`java.io.Serializable`** interface or its sub interface, that is, **`java.io.Externalizable`**. The Java serialization uses reflection to pull all the data of the object fields which we want to serialize. In case the object has private fields or final fields; they are also included in the process of serialization. When the object has a field of an object type, then the object is serialized recursively.

On the other side, deserialization is the opposite process as that of serialization. It is the process of converting the serialized form of an object back into the copy of the object. When the deserialization takes place, it creates an empty object and then uses reflection to write the data to the fields.

The process of deserialization has a security vulnerability. The security vulnerability occurs when a malicious user tries to insert a modified serialized object into the system. This eventually compromises the system or system data. Now the question is, how we can prevent this? The best way to prevent Java deserialization vulnerability is to prevent Java serialization overall. It means that if our application does not accept serialized objects at all, it cannot harm our system. However, it's not a practical solution. According to **`JEP2901`**, one can decide to allow the incoming streams of object serialization to be filtered in order to improve the security as well as robustness.

The **`ObjectInputFilter`** is a functional interface added in JDK 9. The **`Filter`**, **`CompositeFilter`**, and **`FilterFactory`** are the parts of deserialization filtering. Every filter performs checks on the classes and resource to determine to set the status as **`REJECTED`**, **`ALLOWED`**, or **`UNDECIDED`**. We can have more than one filter and also merge or combine their results. The responsibility of establishing and updating the filter for each **`ObjectInputStream`** is taken care of by the **`FilterFactory`**. The **`ObjectInputFilter.Config,`** is the utility class to set as well as get the JVM wide deserialization filter factory and the static JVM wide filter. It also enables the creation of a filter from a pattern string using the **`createFilter()`** method.

The syntax of the method is as follows:

```
public static ObjectInputFilter createFilter(String pattern)
```

The method returns an **`ObjectInputFilter`**, from a string of patterns. This is a filter to apply the check on the object being serialized. If no patterns are provided, it returns **`null`**.

The method parameters are:

- **pattern:** The pattern string to parse; not null.

The method throws **`IllegalArgumentException`**, when the pattern string is illegal, malformed, or it cannot be parsed.

We must consider the following points while applying the pattern of the filter:

- We can combine the patterns which are separated by `";"`.
- We can have whitespace considered as the part of the pattern.
- The `"="` can be used to set a limit and when a limit appears more than once, only the last value is used.
- When the pattern starts with `"!"`, then the class is rejected if the remaining pattern is matched; otherwise, the class is allowed if the pattern matches.
- When the pattern contains `"/"`, the non-empty prefix up to the `"/"` is the module name.
- When the module name matches the module name of the class, then the remaining pattern is matched with the class name. If there is no `"/"`, the module name is not compared.
- When the pattern ends with `".**"`, it matches any class in the package and all subpackages.
- When the pattern ends with `".*"`, it matches any class in the package.
- When the pattern ends with `"*"`, it matches any class with the pattern as a prefix.
- When the pattern is equal to the class name, it matches. Otherwise, the pattern is not matched.

The resulting filter performs the limit checks and then tries to match the class, if any. If any of the limits are exceeded, the filter returns **`Status.REJECTED`**. When the class is an array type, the class to be matched is the element type. Arrays of any number of dimensions are treated the same as the element type.

The **`Status.ALLOWED`** or **`Status.REJECTED`** is returned when the first pattern matches while working from left to right.

When the limits are not exceeded and no pattern matches, the class result is said to be **`Status.UNDECIDED`**.



Use the createFilter() method to apply the filter on the ObjectInputFilter. config. The parameters used in this method decides which object can be deserialized. The different conditions applied for deserialization are provided in the String format.

Problem

How to set a filter for allowing deserialization from some particular packages and rejecting classes from other packages?

Solution

`Listing 8-9, DemoAllowFilter.java` shows how to use the **`allowFilter()`** method to allow the classes from a particular package in the process of serialization:

```
1. //Listing 8-9
2. package com.java17.filter;
3.
4. import java.io.FileInputStream;
5. import java.io.FileOutputStream;
6. import java.io.IOException;
7. import java.io.ObjectInputFilter;
8. import java.io.ObjectInputStream;
9. import java.io.ObjectOutputStream;
10.
11. public class DemoAllowFilter {
12.
13.   public static void main(String[] args) throws IOException,
                                    ClassNotFoundException {
14.    var filter\ = ObjectInputFilter.allowFilter(cl ->
15.
    cl.getPackageName().contentEquals("com.java17.filter"),
```

```
            ObjectInputFilter.Status.REJECTED);
    ObjectInputFilter.Config.setSerialFilter(filter1);
    ObjectInputFilter.Config.setSerialFilterFactory((f1, f2) ->
            ObjectInputFilter.merge(f2, f1));
    serialize(new DemoPojo("Demonstration"), "demo.txt");
    deserialize("demo.txt");
  }

  public static void serialize(Object value,
                        String filename) throws IOException {
    System.out.println("Serialization Process Initiated =>");
    FileOutputStream fileOut = new FileOutputStream(filename);
    ObjectOutputStream out = new ObjectOutputStream(fileOut);
    out.writeObject(value);
    out.close();
    fileOut.close();
    System.out.println("Object Serialized!");
  }

  public static void deserialize(String filename) throws
                       IOException, ClassNotFoundException {
    System.out.println("Deserialization Process Initiated =>");
    FileInputStream fileIn = new FileInputStream(filename);
    ObjectInputStream in = new ObjectInputStream(fileIn);
    try {
     Object o = in.readObject();
     System.out.println("Object Deserialized!");
     if (o instanceof DemoPojo) {
      DemoPojo demo = (DemoPojo) o;
      System.out.println(demo);
     }
```

```
45.     } catch (java.io.InvalidClassException e) {
46.      // TODO: handle exception
47.      e.printStackTrace();
48.     }
49.    }
50. }
```

Output

The following output as shown in in *Figure 8.8* is displayed when we execute the preceding program:

```
Serialization Process Initiated =>
Object Serialized!
Deserialization Process Initiated =>
Object Deserialized!
DemoPojo :Demonstration
```

Figure 8.8: *Controlling Deserialization*

Observe that the object is serialized and successfully deserialized.

Explanation

The process of serialization and deserialization is the classical process, which we have already discussed. The change we incorporated here is to set the filter. The filter will allow the deserialization, depending on some conditions and rejecting from other using the **`allowFilter()`** method.

The syntax of the method is as follows:

```
static ObjectInputFilter allowFilter(Predicate<Class<?>> predicate,
                                     ObjectInputFilter.Status otherStatus)
```

The method returns a filter which returns **`Status.ALLOWED`**, when the predicate on the class is true.

The filter can return any of the following:

- **`ALLOWED`**, when the predicate condition matches.
- **`UNDECIDED`**, if the class is **`null`** and the **`otherStatus`**, based on the predicate of the **`non-null`** class.

The method parameters are:

- **Predicate**: A predicate to test a class non-null.
- **`otherStatus`**: A status to use if the predicate returns false.

In the following code, we have used:

```
var filter1 = ObjectInputFilter.allowFilter(cl ->
                    cl.getPackageName().contentEquals("com.java17.f
```

We are setting the deserialization to allow all the classes which belong to the **`com.java17.filter`** package. Because of this the object is successfully deserialized, as shown in *Figure 8.8*.

But when we try to perform deserialization from **`DemoPojo.class`**, which belongs to **`com.java17.filter1`**, we will get an exception as shown in *Figure 8.9*:

```
Serialization Process Initiated =>
Object Serialized!
Deserialization Process Initiated =>
java.io.InvalidClassException: filter status: REJECTED
```

Figure 8.9: *Rejected filters*

As you can observe, the object is serialized but the exception is raised when we tried to deserialize it.



The allowFilter(), rejectFilter(), rejectUndecidedClass(), and merge() methods are added to the ObjectInputFilter interface to create faster deserialization filters in Java 17. Typically, these methods are used to check the package to which the class belongs to, or the type of the class, or object of which you wish to deserialize.

Problem

How to set criteria so that a specific class from the package will stop deserialization? How to use the **`rejectFilter()`** method to set criteria in deserialization?

Solution

In the earlier code, we used the **`allowFilter()`** method to specify the package from which (in our case, **`com.pojo`**) the classes can be deserialized. Now, on top of that, we can set the criteria from the same package so that the **`Employee`** class will not be deserialized, as follows:

```
1. public static void deserialize(String filename) throws
                  IOException, ClassNotFoundException {
2.   System.out.println("---deserialize");
3.   FileInputStream fileIn = new FileInputStream(filename);
4.   ObjectInputStream in = new ObjectInputStream(fileIn);
5.   ObjectInputFilter empFilter =
           ObjectInputFilter.rejectFilter(cl ->
     cl.equals(Employee.class),
6.                        ObjectInputFilter.Status.UNDECIDED);
7.   in.setObjectInputFilter(empFilter);
8.
9.   try {
10.    Employee employee = (Employee) in.readObject();
11.    System.out.println(employee);
12.  } catch (java.io.InvalidClassException e) {
13.    e.printStackTrace();
14.  }
15. }
```

Explanation

Here, we have used the **`rejectFilter()`** method to specify the classes which we want to stop from deserialization.

The method signature is as follows:

```
static ObjectInputFilter rejectFilter(Predicate<Class<?>>
predicate,
        ObjectInputFilter.Status otherStatus)
```

The method returns:

- A filter which returns **`Status.REJECTED`** when the specified predicate returns true. The filter can return any of the status, which are listed in the following:
    - **`UNDECIDED`** when the **`serialClass`** is null.
    - **`REJECTED`** when the predicate on the class returns true.

  Otherwise, the method returns **`otherStatus`**.

The method parameters are:

- **`predicate`**: The predicate to test the class is non-null.
- **`otherStatus`**: A status to use when the specified predicate is false.



The process of deserialization can be controlled not only for the package, but also for the classes within the package. The rejectFilter() method can be used to apply filters so that if the predicate is matched, the deserialization is rejected.

Problem

We can prevent deserialization of certain classes. Is it possible to prevent deserialization of record classes as well? How to prevent deserialization of certain record classes?

Solution

Records are no exceptions when they are connected to the filters. The following code snippet shows how to use the record classes to filter the deserialization.

In the snippet, two record classes are created in the **`com.pojo.records`** package, as **`Employee.java`** and **`Student.java`**.

Create the **`Employee`** record as follows:

```
package com.pojo.records;
import java.io.Serializable;
public record Employee(int empId,String empName)
               implements Serializable{    // code goes here }
```

Create the **`Student`** record as follows:

```
public record Student(int rollNo,String name) implements
Serializable {
// code goes here
}
```

Now, let us set the filter to allow deserialization from the **`com.pojo.records`** package as follows:

```
var filter1 = ObjectInputFilter.allowFilter(cl ->
            cl.getPackageName().contentEquals("com.pojo.records"),
                        ObjectInputFilter.Status.REJECTED);
ObjectInputFilter.Config.setSerialFilter(filter1);
ObjectInputFilter.Config.setSerialFilterFactory((f1, f2) ->
                          ObjectInputFilter.merge(f2, f1));
```

We can also prevent deserialization of the **`Employee`** record class as follows:

```
FileInputStream fileIn = new FileInputStream(filename);
ObjectInputStream in = new ObjectInputStream(fileIn);
ObjectInputFilter intFilter = ObjectInputFilter.rejectFilter
                          (cl -> cl.equals(Employee.class),
                        ObjectInputFilter.Status.UNDECIDED);
in.setObjectInputFilter(intFilter);
```

Explanation

We can use the same way that we use the **`allowFilter()`** and **`rejectFilter()`** methods to allow or reject certain classes for the normal classic version of classes, just as we can use the methods for record classes with the same ease.

We have already written the code using the **`rejectFilter()`** method to prevent deserialization of the **`Employee`** record class. So, when we try to perform deserialization of the **`Employee`** class as follows:

```
var filename1 = "file1.ser";
Employee employee = new Employee(1, "emp name");
serialize(employee, filename1);
deserialize(filename1);
```

We will get **`InvalidClassException`** as the output.



We can use the allowFilter() and rejectFilter() methods to allow or stop deserialization of certain record classes in the same way, and with the same ease as that of normal classes.

Modified switch case

Java 17 has provided additional features to the switch case block, which we will discuss in the following sections using different scenarios.

Problem

We can use the **`if…else`** condition to test several possibilities and then execute the logic when the condition matches. How to use the switch-case to perform similar conditions?

Solution

The following code shows the switch case approach used as an alternative to the **`if.. else`** block:

```
1. //Listing 8-10
2. package com.java17.switchcase;
3.
4. public class DemoSwitchCase{
5.      public static void main(String[]args){
6.         System.out.println(testSwitchCase("testing if else"));
7.         System.out.println(testSwitchCase(100l));
8.      }
9.
10.     public static String testSwitchCase(Object object){
11.       String result = "";
12.         switch(object){
13.            case Integer i :
14.                 result = result.concat("You have passed an
                                                integer:-"+object);
15.                 break;
16.            case Long l :
17.                result= result.concat("You have passed a
                                                 long:-"+object);
18.                 break;
19.            case Double d :
20.                 result = result.concat("You have passed
                                               a double:-
    "+object);
21.                 break;
22.            case String string:
23.             result= result.concat("You have passed
                                               a string:-"+object);
24.             result=result.toUpperCase();
25.                 break;
26.            default :
27.                 result =" no match";
```

```
28.             }
29.             return result;
30.         }
31. }
```

Output

The following output shown in *Figure 8.10* is displayed if you execute the preceding code:

```
YOU HAVE PASSED A STRING:-TESTING SWITCH CASE
You have passed a long:-100
```

Figure 8.10: *Modified switch*

Explanation

We know when we use the switch block, the value of the selector expression is compared to the labels of the cases. When the label matches the code associated with that label, it will be executed. So, what is the change? Now, we do not have normal cases. Rather, we have the case labels with patterns. The selection of which case block to execute is determined by the pattern matching, rather than the equality check. In our case, the value of object matches the pattern **`Integer I`**, **`Long l,`** or **`String string`**. When we pass the value of object as **`"testing if else"`**, that is, of type string, then the code associated with the case of type **`String`** will be executed.

A similar solution can be generated by using the arrow **`(->)`** operator in the switch case. The code snippet for the same is shown as follows:

```
public static String testSwitchCase(Object object){
      switch(object){
         case Integer i -> { return "You have passed an integer:-
"+i; }
         case Long l ->{ return "You have passed a long:-"+l;}
         case Double d ->{ return "You have passed a double:-"+d;
}
         case String string-> { return "You have passed a
                                                    string:-
"+string;}
         default -> { return" no match";}
       }
}
```



Now, we can have switch to perform pattern matching in cases. Pattern matching facilitates object type checking at runtime.

Problem

In traditional programming, when the selector expression evaluates to **`null`**, the switch statements as well as expressions throw **`NullPointerException`**. Is it possible to integrate the null-test into the switch?

Solution

`Listing 8-11, DemoSwitchCaseWithNull.java` shows how we can handle the null case in the switch case from Java 17 onwards:

```
1. //Listing 8-11
2. package com.java17.switchcase;
3.
4. public class DemoSwitchCaseWithNull {
5.   public static void main(String[] args) {
6.    System.out.println(testSwitchCase(null));
7.    System.out.println(testSwitchCase(100l));
8.   }
9.
10.  public static String testSwitchCase(Object object){
11.    switch(object){
12.          case null -> { return "You didn›t pass anything"; }
13.          case Long l ->{ return "You have passed a long:-"+l;}
14.          case String string-> { return "You have passed a
                                                  string:-
    "+string;}
15.          default -> { return" no match";}
16.         }
17.
18.    }
```

19. }

Output

In the following output, observe that even if we passed null, the switch case did not throw the **NullPointerException**. In fact, it is considered as a null case and that is evaluated. Refer to *Figure 8.11*:

```
You didn't pass anything
You have passed a long:-100
```

Figure 8.11: *Checking null in switch case*

Explanation

Traditionally, when we have the selector expression which evaluates to **null**, it throws **NullPointerException** and one must perform testing for null outside the switch as follows:

```
if (s == null) {
       //some code goes here
       return;
}
```

This was quite reasonable when switch supported only a few reference types. But now, switch allows a selector expression of any type. It also enables case labels to have type patterns. In such a scenario, you might wonder, why it is at all required to perform a standalone null test and add up the boilerplate which leads to an opportunity for error. Considering this the switch case enables performing the null-test, as shown in the preceding code snippet.

The switch case is always determined by its case labels. Now, we can put a case label as **null** and have the code block associated with it to execute. When we do not have a case labelled as **null**, the switch will tend to throw the **NullPointerException** as **null** passed to the switch expression.

We may wish to handle both the **null** value and all **String** values at the same time, as we can update the condition as follows:

```
public static String testSwitchCase1(Object object){
         switch(object){
         case null, String string -> { return "You have passed a
string:-
                                                  "+string; }
         case Long l ->{ return "You have passed a long:-"+l;}
         default -> { return" no match";}
      }
```

```
}
```



The switch now can have a null expression test to avoid boilerplate code for testing nulls outside the switch…case block.

Problem

Is it possible to perform pattern matching for inherited classes using switch… case?

Solution

We can use the switch case in the inheritance hierarchy as well. Observe the following code snippet which has the labels of the **`Manager`** instance and **`WageEmployee`** instance. You can observe the entire code in the **`Listing 8-12, DemoSwitchCaseInheritance.java`**:

```
1. private static void calculateSalary(Employee e) {
   // TODO Auto-generated method stub
2.   switch(e){
3.         case Manager m :
4.              double sal=m.calculateSalary();
5.              if(sal<=3000) {
6.                  System.out.println("Your
                      salary is "+sal+" chance of revision in
   salary");
7.              }
8.              else{
9.                System.out.println("Well paid, salary:-"+sal);
10.             }
11.            break;
12.          case WageEmployee wage :
13.              sal=wage.calculateSalary();
14.                   System.out.println("Salary on hourly basis:-
   "+sal);
```

```
15.                 break;
16.           case null :
17.                  System.out.println("Can't perform calculations
                                          no object provided");
18.                 break;
19.            default :
20.                System.out.println("No Match");
21.
22.       }
23. }
```

You will observe that once we get the matching instance, we can even perform some more logical execution with or without a logical condition as well.

Explanation

Here, we have considered the cases to match the instance of either the **`Manager`** or **`WageEmployee`** type. If you give a closer look at the first case, you will observe that the case is dealing with **`Manager`** instances type. In the same case, we are checking the salary as well. The intent of this code is to have a special case for checking the salary range and then either perform some logic or take some decision. As you can observe in the code, we use a fall through to get the correct behaviour when the manager has a salary greater than 3,000.

The problem here is that when we use a single pattern to discriminate among many cases, it does not scale beyond a single condition. It means we need some way to express a refinement to the pattern. One way to achieve this is to refine the case labels. When we perform such a refinement, many other programming languages call that as a ‘guard'.



The way we can use cases for predefined classes such as String, Integer and Long, one can use user-defined classes as cases.

Problem

How to implement the guarded pattern in Java 17?

Solution

To implement the guarded pattern, we need to modify the **`calculateSalary()`** method implemented in the **`Listing 8-12`**, as follows:

```
1. public static void calculateSalary(Employee e){
2.   switch(e){
3.         case Manager m && (m.calculateSalary()<=3000)-> {
4.                   //business logic
5.          }
6.          case WageEmployee wage ->{
7.                 //business logic
8.          }
9.         case null ->
                          //business logic
10.        default ->
11.            System.out.println("No Match");
12.
13.  }
14. }
```

Explanation

Let us revise the existing code shown in **`Listing 8-12`** for the method **`calculateSalary()`**.

Here, we have written the case for the Manager instance. When we get one, we check whether the salary is greater than 3,000 or not. Accordingly, we perform some logic.

This can be expressed in a more impressive way. Instead of choosing the extension of the functionality of case labels, we can choose to extend the language of patterns. Here, we can use the guarded pattern which can be written as **`p && condition`**, which enables us to refine the pattern **`p`** by an arbitrary expression expressed by the condition. This results in the **`boolean`** value.

So, the modified case looks as follows:

```
case Manager m && (m.calculateSalary()<=3000)-> {
        //some business logic
```

```
}
```

In the code, the value of e is matched against the pattern **`Manager m && (m.calculateSalary() <= 3000)`**. First of all, it matches the type pattern **`Manager m`** and when it matches, the expression **`m.calculateSalary() <= 3000`** is evaluated.

At the same time, the refined pattern matching, that is, the guarded pattern and the non-refined pattern can be part of the same switch expression. In the preceding code snippet, we have used the guarded pattern for **`Manager`** and the non-refined pattern is used for the **`WageEmployee`** object.



- We can use the guarded pattern written as p && condition2, which enables us to refine the pattern p and match an arbitrary expression expressed by the condition.
- We can use the guarded pattern and the normal pattern matching in the same switch expression.

Problem

How to use the sealed class with the modified switch expression in Java 17?

Solution

Let us start with declaring a sealed class as follows:

```
sealed class Employee permits WageEmployee, Manager,Developer{   }
```

Now, let us declare the permitted classes **`WageEmployee`**, **`Manager`**, and **`Developer`** definitions as follows:

```
final class WageEmployee extends Employee{  }
non-sealed class Manager extends Employee {  }
final class Developer extends Employee{  }
```

The following snippet shows how to write the switch expression, which has the case labels for the permitted classes generated from the sealed class:

```
public static void test(Employee e){
  switch(e){
       case Manager m -> {
                System.out.println("I am manager");
            }
        case WageEmployee wage ->{
                System.out.println("I am wage employee");
        }
        case Developer develpoer->{
```

```
                System.out.println("I am a developer");
        }
        case Employee employee->{
                System.out.println("I am an employee");
        }
    }
}
```

Explanation

As seen in the preceding code snippet, we can use the sealed classes in the switch case as well. We already know what the sealed classes are. In our solution, we have declared a sealed class **`Employee`**. The **`Employee`** class permits the **`WageEmployee`**, **`Manager,`** and **`Developer`** as its type. So, all these three classes are types of Employees as shown in the solution.

When we want to check the objects of types **`WageEmployee`**, **`Manager,`** and **`Developer`** in the switch, we can use the switch expression of the type sealed class **`Employee`**. Once we declare this, the rest part of the switch case works similar to the pattern matching approach that we have seen in the earlier demonstrations.



We now can use sealed classes in case labels to perform pattern matching.

Problem

Can we use switch case pattern matching with classes implementing sealed interfaces as well?

Solution

We can use the sealed interface in the switch case pattern matching as discussed in the following:

First, we declare the sealed interface as follows:

```
    sealed interface MyInterface permits Employee {    }
```

Then, write the sealed class implementing the sealed interface as follows:

```
sealed class Employee implements MyInterface permits
WageEmployee,
      Manager, Developer{   }
```

The permitted class definitions will be as shown in the following code:

```
final class WageEmployee extends Employee{  }
non-sealed class Manager extends Employee {  }
final class Developer extends Employee{  }
```

Once all the definitions are ready, we can use the switch block for pattern matching, as shown in the following:

```
public static void test(MyInterface e){
  switch(e){
       case Manager m -> {
                System.out.println("I am manager");
            }
        case WageEmployee wage ->{
                System.out.println("I am wage employee");
        }
case Developer develpoer->{
                System.out.println("I am a developer");
        }
        case Employee employee->{
                System.out.println("I am an employee");
        }
    }
}
```

Explanation

The approach of using the sealed interface type in the switch is no different than using the sealed classes in the switch case. The only difference is to use the sealed interface, rather than using the sealed classes, as shown in the preceding solution.



We can use sealed classes implementing the sealed interface in the pattern matching of switch.

Reflection API for a sealed class

Java 17 added some features for the reflection process, which can be used with sealed classes. Using this, you will able to find the runtime information about the sealed class.

Problem

In the earlier versions of Java, one can be able to find out metadata information about a particular class using reflection. In the same way, is it possible to find the characteristics of sealed classes?

Demonstrate the use of reflection API to find characteristics of sealed classes.

Solution

Let us first define a couple of sealed classes used by the reflection API, as follows:

```
sealed interface MyInterface permits Employee {    }
sealed class Employee implements MyInterface permits WageEmployee,
                                                    Manager,Developer{
}
final class WageEmployee extends Employee{  }
sealed class Manager extends Employee permits SalesManager{  }
final class Developer extends Employee{  }
final class SalesManager extends Manager { }
```

Now, it is time to use the reflection API to find the characteristics of sealed classes using the code as follows:

```
public class TestSealedClassWithReflection{
public static void main(String []args) throws Exception {
    Class cls=Class.forName("Employee");
    System.out.println(cls.getName());
    Class [] classes=cls.getPermittedSubclasses() ;
    List<Class>sealedclasses=new ArrayList();
    for(Class c: classes){
         System.out.println(c.getName());
         if(c.isSealed())
             sealedclasses.add(c);
   }

     System.out.println("list of sealed classes:-");
      if(sealedclasses.size()>0){
             for(Class c: sealedclasses){
                  System.out.println(c.getName());
            }
       }
  }
}
```

Output

You will get all the characters of the record classes as shown in *Figure 8.12* if you execute the preceding program:

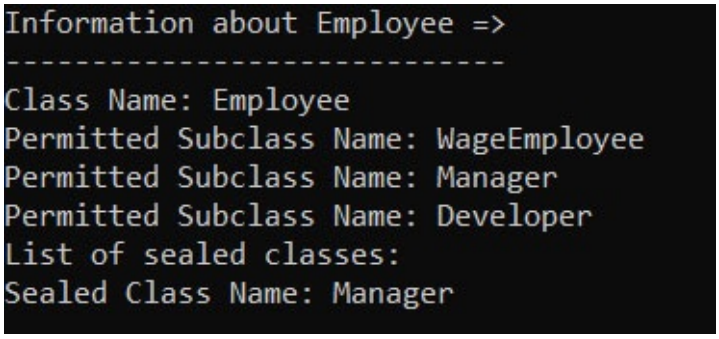


Figure 8.12: *Characters of the Record class using Reflection*

Explanation

Java 17 has added two methods to support the characterization of sealed classes as follows:

`getPermittedSubclasses()`

The method syntax is as follows:

```
public Class<?>[] getPermittedSubclasses()
```

The method returns:

- An array containing the **`Class`** objects, which are the direct subclasses or subinterfaces permitted to extend or implement this class or interface.
- If in case the interface or class is not sealed, it will return null. The elements ordered in this array are unspecified. The array will have size equal to zero, when this sealed class or interface has no permitted subclass.

For each class or interface which is specified as a ‘permitted direct subinterface' or ‘subclass' of this class or interface, the **`getPermittedSubclasses()`** method attempts to obtain the **`Class`** object. All such **`Class`** objects which can be obtained and which are the direct subinterfaces or subclasses of this class or interface are then added as elements of the returned array. When the **`Class`** objects cannot be obtained, it is silently ignored and is not included in the resultant array.

The method throws the **`SecurityException`**.

`isSealed ()`

The syntax of the method is as follows:

```
public boolean isSealed()
```

The method returns:

- True when the **`Class`** object represents a sealed class or an interface. However, when this **`Class`** object is of type primitive, void, or an array, the

method returns value as false.

- False when the **`Class`** object is of type primitive, void, or an array.



Now, Java reflection for the sealed classes is supported with addition of the getPermittedSubclasses() and isSealed() methods in the java.lang.Class.

Vector API

The **`Vector`** API may be new to developers, but it is not new for Java. In fact, the **`Vector`** API is already present from Java 16 version. However, it is still in the incubator module. The **`Vector`** API facilitates parallel programming in Java, which improves the performance.

Problem

What is the **`Vector`** API is used for? How is it used to improve the performance of an application?

Solution

To understand how the **`Vector`** API works, we will take a classic scenario. Let us assume that we have two arrays containing integers, representing the salary of 5 employees. Also, we have another array which stores the incentives of 5 employees. Now, we would like to add the salary and the incentive for each employee in the third array, so that the total salary of each employee will be calculated.

Traditionally, we can achieve this by using simple addition of array elements, as follows:

```
public static int[] calculateSalary(int[] salary, int[] incentives)
{
    int[] totalSalary = new int[salary.length];
    for (int i = ·; i < salary.length; i++) {
        totalSalary[i] = salary[i] + incentives[i];
    }
    return totalSalary;
}
```

Nothing is really wrong in this code snippet. We are adding the individual array elements and storing it into the third array. However, you can achieve the same using the **`Vector`** API, as shown in the following code snippet:

```
public static int[] calculateSalary(int[] salary, int[] incentives,
                                    VectorSpecies<Integer> species)
{
    int[] totalSalary = new int[salary.length];
    int limit = species.loopBound(salary.length);
    var i = ·;
    for (; i < limit; i += species.length()) {
        IntVector vectorSalary = IntVector.fromArray(species, a,
i);
        IntVector vectorIncentive = IntVector.fromArray(species, b,
i);
        IntVector vectorTotalSalary =
vectorSalary.add(vectorIncentive);
        vectortotalSalary.intoArray(totalSalary, i);
    }
    for (; i < salary.length; i++) {
        totalSalary[i] = salary[i] + incentives[i];
    }
    return totalSalary;
}
```

Explanation

As mentioned earlier, the **`Vector`** API was introduced in Java 16. Although it is still in the incubator mode, Java is constantly working on improving the performance of the algorithm. **`Vector`** is the result of one of such efforts from Java API developers.

`Vector` is an abstract class provided to achieve the data-parallel computations. In many situations, we have a single operation to perform on multiple data. Such operations are performed traditionally by the approach of **Single Instruction Single Data** (**SISD**). But vectors provide the lanes for multiple data combinations, where the operations are performed in each lane to produce the scalar results. At the end, the vector results are generated. Such approach is termed as **Single Instruction Multiple Data** (**SIMD**).

Terminologies in the Vector API

- **Shape of Vector**: The shape of the vector defines the information capacity of the vector. It is also termed as **`VSHAPE`**. All the possible values of **`VSHAPE`** are stored in **`VectorShape Enumeration`**.
- **Species of Vector**: The unique element type and the vector shape of vector is responsible in determining vector species.

Subclasses of the Vector

There are six abstract subclasses of the **`Vector`** and they are, **`ByteVector`**, **`ShortVector`**, **`IntVector`**, **`LongVector`**, **`FloatVector`**, and **`DoubleVector`**.

As you can guess, these are type of specific subclasses. They provide the operations which are supported by that particular type. Along with this, these classes also provide the facility to generate the vector values based on existing values used for scalar calculation.

In our solution, we have used the **`IntVector`** because we want to create a vector from array elements.

Observe the following method that we have written in the code snippet:

```
public static int[] calculateSalary(int[] salary, int[] incentives,
                                        VectorSpecies<Integer>
species){
//Business logic
}
```

The third parameter of this method takes the **`VectorSpecies`**. The possible values for **`VectorSpecies`** for **`IntVector`** are as follows:

- SPECIES_64
- SPECIES_128
- SPECIES_256
- SPECIES_512
- SPECIES_MAX
- SPECIES_PREFERRED

These are all constants present in the **`IntVector`**. You can create species by using the syntax:

```
IntVector.SPECIES_TYPE
```

For example, to create the species of type preferred, you can use:

```
IntVector.SPECIES_PREFERRED
```

Once you get the **`VectorSpecies`**, you can get the maximum length of the species by invoking the **`loopBound()`** method on it.

There are many methods present in the **`Vector`** class as well as the **`IntVector`** class. Let us discuss the methods that we have used in our solutions:

`add()`

This method is used to add the vector for the broadcasting of the corresponding scalar. This is also termed as lane-wise binary operation. This applies the

primitive addition operation to each of the lane.

The syntax of the method is as follows:

```
public final IntVector add(int e)
```

The method returns the result of adding each lane of this vector to the scalar.

`fromArray()`

This method is used to load the vector from the values of the provided array starting at an offset.

The syntax of the method is as follows:

```
public static IntVector fromArray(VectorSpecies<Integer> species,
                                                    int[] a, int
offset)
```

The method parameters are:

- **`species`**: Species of the desired vector.
- **`a`**: The array.
- **`offset`**: The offset into the array.

The method returns the **`Vector,`** returned from the given array.

`intoArray()`

This method stores the **`Vector`** into an array.

The method signature is as follows:

```
public final void intoArray(int[] a, int offset)
```

The method parameters are:

- **`a`**: The array of int type to which the vector has to converted.
- **`offset`**: The offset into an array.



The Vector API in Java 17 can convert the scalar elements into the vectors. Using this, we can create the parallel operations through the concepts of lane, where the elements are pushed to the lanes to provide some common operations. The operations are speed up by using the Vector API.

Conclusion

In this chapter, we discussed features introduced in Java 17. We learned how to generate the random numbers by using different predefined algorithms. Specifically, we talked about the **`SplittableRandom`** generator algorithm and observed how the **`split()`** method is used to split the generated random

numbers. We also learned to control the deserialization process by checking the packages of the serialized object and also the type of the same. We also invoked the reflection API on the sealed classes and interfaces to get the runtime information about it. Finally, we discussed the interesting concept of the `Vector` API which facilities faster processing of the scalar values by converting them into vectors.

Keywords

- RandomGenerator
- RandomGeneratorFactory
- ThreadLocalRandom
- SplittableRandom
- SplittableGenerator
- FilterFactory
- rejectFilter()
- rejectUndecidedClass()
- merge()
- allowFilter()
- getPermittedSubclassses
- isSealed()
- Vector
- IntVector

Questions

1. How to use the RandomGenerator interface?
2. What are the different algorithms to generate random numbers?
3. How to use the SplittableRandom algorithm?
4. What are the overloaded methods of nextInt() to generate the random integer values?
5. What is the significance of the split() method of SplittableGenerator?
6. How the createFilter() works in the deserialization?

7. Explain how to work with the allowFilter() method.
8. What are the possible values of filters applied for deserialization?
9. How to handle the null in the switch case?
10. How to use the switch case in pattern matching?
11. What is the Vector API used for?
12. What are the different subclasses of Vector?

Index

A

add() method, Vector class 301
Application Class Data Sharing 94-100
  dump of classes, creating 96
  dump of shared archive file, creating 98
  shared classes list, creating 96
  shared dump, for app launch 98
Arrays class 21
  compare() method 23, 24
  equals() method 23
  mismatch() method 24-28
  using 21-24

B

BodyHandler object 129

C

cancel() method 49
CharSequence interface 215
  improvement 215-217
checkInput() method 40
Collection API, Java 9 13
  using 14-21
Collection API, Java 10 78-83
  List.copyOf() method 83, 84
  Map.copyOf() method 85, 86
  Set.copyOf() method 84, 85
Collectors.teeing()
  using 136-140
Collectors.toUnModifiableList() method 88, 89
Collectors.toUnmodifiableMap() method 89, 90
Collectors.toUnModifiableSet() method 89
Compact Number Formatting, Java 14
  plural support 192, 193
CompatNumberFormat class, Java 12 157-160
CompletableFuture API 53, 56
  methods 57-60
CompletableFuture interface
  exceptionallyAsyc(), using with executor 153-155
  exceptionallyAsyc(), using without executor 150-153

exceptionallyComposeAsync() 155-157
updates, in Java 12 150

D

Date - Time API 223
scenarios 223-230
using 223
default int nextInt(int bound) method 270
default int nextInt(int origin, int bound) method 270, 271
default int nextInt() method 271
default methods
invoking, from proxy 220-223
defineHiddenClass() method 213
describeConstable() method 145
deserialization filtering 276-285

E

exceptionallyAsyc()
using, with executor 153-155
using, without executor 150-153
exceptionallyComposeAsync()
using 155, 156
exports directive 66
exports to directive 67

F

findStatic() method 214
Flow API 42-47
fromArray() method 301

G

generic record class
declaring 254
getDeclaredMethod() 214
getNestHost() method 109
getNestMembers() method 110, 111
getPermittedSubclasses() method 297

H

hidden class 210
using 211-214
HttpClient API 124-127
advantages 127-129
HttpURLConnection API

issues 127

I

ifPresentOrElse() method 29
using 32, 33
indent() method 143
instanceof operator, Java 16
using 233-236
intoArray() method 301, 302
IntVector 300
isBlank() method 117
isNestmateOf() method 110
isSealed() method 298

J

Java 8
Stream API 6
Java 9
Arrays class 21
Collection API 13
CompletableFuture API 53
Flow API 42
Java Platform Module System 60
ObjectInputFilter interface 37
Optional interface 29
private method interfaces 2
Java 10
Application Class Data Sharing 94
Collection API 78
local variable type inference 72
Optional interface 92
Stream API 86
Java 11
HttpClient 124
installation 104
nest-based access control 104
nested class 119
single-file source-code program 130
String API 112
Java 12 135
Collectors.teeing() 136
CompatNumberFormat class 157
CompletableFuture interface updates 150
installation 136
NIO updates 147
String API 140
Java 13 164

installation 164
java.nio.file.FileSystems update 165
XML parsers update 169
Java 14
Compact Number Formatting, plural support for 192
installation 171
NullPointerException 189
switch case updates 172
updates 171
Java 15 195
hidden class 210
installation 196
text blocks, using 196
Java 16
Date - Time API 223
installation 220
Packager tool 254
Stream API 230
Java 17
installation 262
Random Number Generator Algorithm improvements 262
Reflection API 295
switch case 285
Vector API 298
java.nio.file.FileSystems
methods 166-169
newfileSystem(Path, Map<String,?>)
method 167, 168
newFileSystem(Path) method 166, 167
updates 165, 166
Java Platform Module System 60
implementing 60-65
multiple modules, working with 65-67
Java SE Development Kit Version 9
installation 2
jshell 35, 36

L

lines() method 117, 118
List.copyOf() method 83, 84
local-variable syntax
for lambda parameters 121-124
local variable type inference
using 72
var keyword, using 72
var keyword, using in anonymous class 75
var keyword, using in loops 74
var keyword, using in polymorphism 73, 74

M

Map.copyOf() method 85, 86
multiline strings
  handling 201
  string embedded HTML, handling 203
  string embedded JSON, handling 201, 202
  string embedded XML, handling 203

N

nest-based access control, Java 11 104
  Java 9 compiler, using 106
  Java 11 compiler, using 106
  working 105
nested class, Java 11
  reflective access 119-121
newDefaultNSInstance() method 170
newfileSystem(Path, Map<String,?>)
method 167
newFileSystem(Path) method 166
newNSInstance()
method 170
newNSInstance(String,ClassLoader) method 170, 171
NIO updates, in Java 12 147-150
non-static/instance method
  invoking 214
NullPointerException, Java 14 189-191
  Java14 compiler, using 190
  pre Java14 compiler, using 190

O

ObjectInputFilter interface 37-41
onComplete() method 49
onError(Throwable throwable) method 48, 49
onNext(T item) method 48
onSubscribe(Subscription subscription) method 47, 48
Optional<T> class 29
Optional interface, Java 9 29-34
Optional interface, Java 10 92-94
or() method 29-34

P

Packager tool, Java 16 254,-256
pattern matching 234
PluralCompactNumberFormat.java 192, 193
private method interfaces 2

using 3-5
provides with module specifier 67
Pseudo Random Number Generator (PRNG) 262
public BigInteger period() method 275
public int stateBits() method 275
public static interface Flow.Subscriber<T>
onComplete() method 49
onError(Throwable throwable) method 48, 49
onNext(T item) method 48
onSubscribe(Subscription subscription) method 47, 48
public static interface Publisher<T> 49
subscribe() method 49
public static interface Subscription 49
cancel() method 49, 50
request(long n) method 49
public String formatted() 206, 207
public String group() method 275
public String name() method 275
public String stripIndent() method 208
public T create(byte[] seed) method 268
public T create(long seed) method 268
public T create() method 268

R

RandomGeneratorFactory 266
characteristics 274, 275
RandomGenerator interface 264
Random Number Generator Algorithm
improvements 262-270
Read-Eval-Print-Loop (REPL) 34
record class
accessor method, defining 248, 249
as inner class member 252, 253
block, declaring 241
canonical constructor, defining 244
compact constructors, using 246-248
declaring 236, 237
defining, in class 250-252
explicit canonical constructor, defining 242, 243
explicit canonical constructor, with some/none field initialization 243
extra instance fields, defining 239
function, declaring 241
generic record class, declaring 254
instance method, defining 248, 249
instantiation 241, 242
multiple interfaces, implementing 239
overloaded constructors, defining 245
restrictions, on headers 240

static field, declaring 241
using 238
reflection API, Java 11 107-109
getNestHost() method 109, 110
getNestMembers() method 110, 111
isNestmateOf() method 110
reflection API, Java 17
for sealed class 295-298
repeat() method 118
request(long n) method 49
requires directive 65
requires transitive directive 66
resolveConstantDesc() method 146

S

serialization 37
Set.copyOf() method 84, 85
single-file source-code program
launching 130-132
Single Instruction Multiple Data (SIMD) 300
Single Instruction Single Data (SISD) 299
split() method
using 271-273
SplittableGenerator interface 273
SplittableRandom Algorithm 271
static method
invoking 214
Stream API, Java 9 6
dropWhile(), using 9-11
iterate() method, using 11-13
takeWhile(), using 6-9
Stream API, Java 10 86-88
Collectors.toUnModifiableList() method 88, 89
Collectors.toUnmodifiableMap() method 89, 90
Collectors.toUnModifiableSet() method 89
toUnModifiableXXX() method 88
Stream API, Java 16 230
using 230, 232
stream() method 29-34
String API, Java 11 112-114
isBlank() method 117
lines() method 117, 118
repeat() method 118
stripLeading() method 116
strip() method 114-116
stripTrailing() method 116
String API, Java 12 140
describeConstable() method 145, 146

indent() method 143, 144
methods 141-143
resolveConstantDesc() method 146
transform () method 144, 145
stripLeading() method 116
strip() method 114
stripTrailing() method 116
SubmissionPublisher 50
constructors 50, 51
offer() method 51-53
submit(T item) method 51
switch case, java 17 285-295
switch case statement, Java 14
arrow operator, using 175, 176
expressions, using 182
local variable, using 178-181
multiple cases, using 178
multiple labels and arrow operator, using 177, 178
operations, performing 185-189
scenarios 174, 175
switch expression and System.out.println() 183, 184
updating 172
with break statement 173, 174
without break statement 172, 173
without switch expression 182
with switch expression 183

T

text blocks
string embedded HTML + operator, handling 198
string embedded HTML, handling with StringBuilder 199
string embedded HTML, handling with String.format() function 199, 200
string embedded HTML, handling with String.join() 201
string embedded HTML, handling with StringWriter 200
string embedded XML + operator, handling 197, 198
using 196, 197, 204-206
toUnModifiableXXX() method 88
transform () method 144
traslateEscapes() method 208-210

U

uses module specifier 67

V

value-based classes 256-258
var keyword

usage basics 72
using, in anonymous class 75
using, in loops 74
using, in polymorphism 73, 74
variable declaration examples 76, 77
Vector API 298
shape of vector 300
species of vector 300
subclasses 300-302
working 298-300
VectorSpecies 300

X

XML parsers, Java 13
newDefaultNSInstance() method 170
newNSInstance()
method 170
newNSInstance(String,ClassLoader) method 170
updating 169, 170