

Chapter 2:

Basics of Java

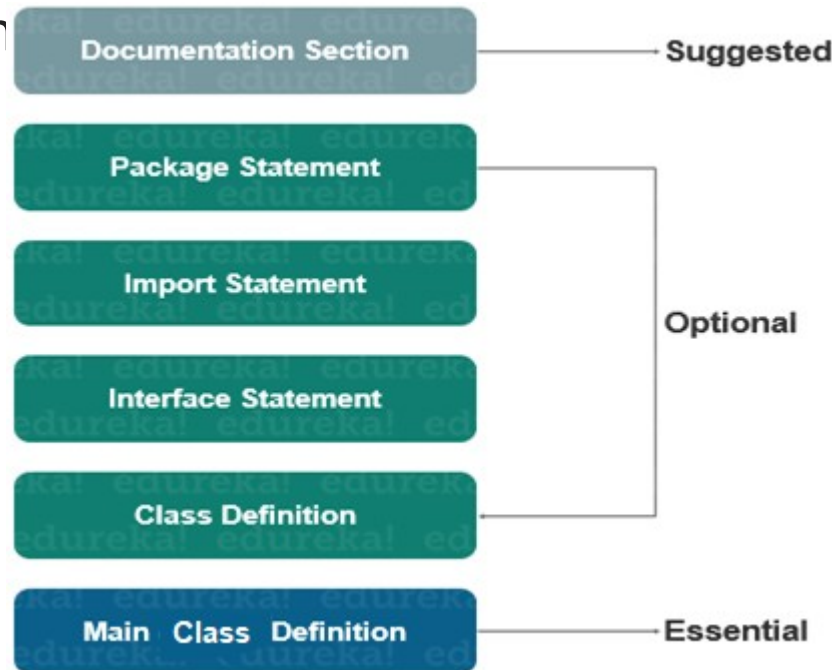
Java Program

Java program

a **Java program** can be **defined** as a **collection of objects** that communicate via invoking each other's **methods**.

- An object is an instance of a class

Structurally a Java program involves the following section



Java program...

- **Documentation Section:** Makes program understandable
- **Package Statement:** used to group classes
- **Import Statements:** to use classes from another package
- **Interface Statement:** contains method declarations
- **Class Definition:** definition of classes
- **Main Class Definition:** The class which contains **the main method definition** which is the starting point of the program. Every java program must contain such

Java Class Structure

```
class MyClass [extends AnotherClass]
```

```
{
```

Class header

Class body

```
}
```

A class definition contains two parts

1. A class header

contains the keyword **class**, the **name** of the class (MyClass) and may be derived from another class

2. A class body

contains **data(fields)** and **code(method)** enclosed in a pair of braces.

Class name must be the same as the file name where the class lives

Java Main Class

```
import java.lang.*;  
package cs224.codes;  
//Comments about the class  
public class MyProgram  
{  
    static int id;  
    //Comments about the method  
    public static void main(String [] args)  
    {  
        //Main Method body  
    }  
    static retType method1(parameters)  
    {  
    }  
}
```

Importing java language Package
Class MyProgram is in package
cs224.codes

Member Variable
Or field

The Main
Methods

Member
functions
Or Methods

Main Method header

Main Method body

retType method1(parameters)
Declaration

Java program

- **The main class** is the class which defines **the main method**
- This class should **be public (visible from outside the class)**
- **static** indicates that the `main()` method is a **class method not an instant method**. It allows ***main()*** to be called without instantiating a particular instance of the class
- **extends** allow the new class to be derived from AnotherClass

Java program

- **A java program may contains one or several classes**
- If a java source file contains many classes, **only one of the class can be public** and the name of the source file must be named after the public class.
- There must be only one **main class** in a java program source file.
- **Execution** of the program starts in the **main method** of the **main class**
- The syntax of the main method must be **strictly followed**
- The program can be created in any text editor

Your First program

```
import java.lang.*;
/**
 * My first <b>Test Program</b>
 * @author me
 * @version 1.1
 */
public class HelloWorld {
    /**
     * @param args the command line arguments
     * @since 1.0
     */
    public static void main(String[] args) {
        System.out.println("Hello World");
    }
}
```

This source code must be saved **with the same name as the class** even the case must be preserved and with an extension **.java**

Compiling and running the first program

- Open the command prompt in Windows.
 - Change the path to the directory where your files is stored
- To compile the program, type the following at the command prompt:
javac HelloWorld.java
- You have now created your first compiled Java program named **HelloWorld.class (The bytecode)**. You will get it in the same directory as **HelloWorld.java(The source file)**
- To run your first program, type the following at the command prompt:
java HelloWorld
- Although the file name includes the **.class**

Command line arguments

A **command-line argument** is an information that directly follows the program's name on the **command line** when it is executed.

They are stored as strings in the **String array** passed to `main( )`.

```
public class CommandLine {  
    public static void main(String args[]) {  
        for(int i = 0; i<args.length; i++) {  
            System.out.println("args[" + i + "]: " + args[i]);  
        }  
    }  
}
```

What will be produced if the preceding program executed as follows?

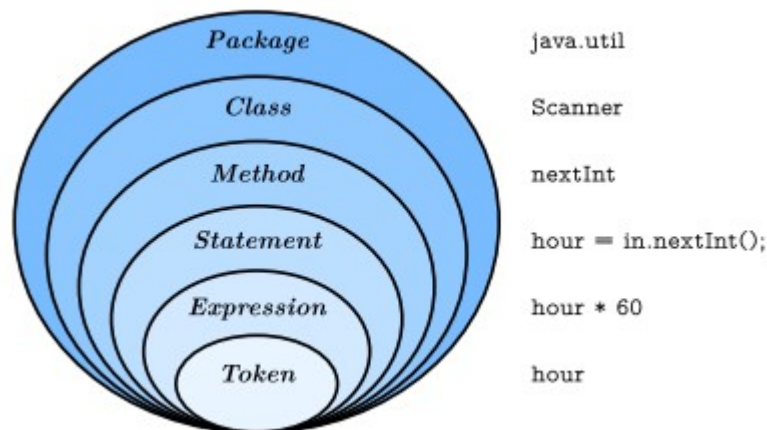
```
java CommandLine 200 -100 apple dog
```

Java Lexical

Java basic lexical

**Java basic lexical elements are
comment and tokens**

Java program = comment + tokens



Java Comments

Comments can be used to explain Java code, and to make it more readable

Three type of comments:

1. line comment

Starts with two forward slashes(//)

Any text following the // to the end of the line is comment.

Ignored by the compiler

`System.Out.println("Hello world"); //prints hello world on the console`

2. Multi-line comment

It starts with /* and ends with */

Any text between /* and */ is a comment. Ignored by java

`/*The next line print hello world on the screen.`

`It is amazingly simple*/`

`System.Out.println("Hello world");`

3. Documentation Comments

This is a documentation comment and in general its called **doc comment**. The **JDK javadoc** tool uses doc comments when preparing automatically generated documentation.

Example:

```
/**  
 * The <b>HelloWorld</b> program implements an  
 * application that  
 * simply displays "Hello World!" to the standard output.  
 *  
 * @author Thomas  
 * @version 1.0  
 * @since 2014-03-31  
 */
```

Java tokens

Java program is constituted from the following atomic unit(token):

- White space
- Identifiers
- Literals
- Separators
- Keywords
- Operators

Whitespaces

Whitespaces such as **space, tab and newline** is used **to separate tokens** in java statement.

At least one whitespace should be used to separate tokens which are not already delineated by an operator or separator

Java Identifiers

Identifiers are used to **name programming objects** such as variable, classes and methods.

An identifier may be any descriptive sequence of uppercase and lowercase letters, numbers, or the underscore and dollar-sign characters. (**The dollar-sign** character is not intended for general use.)

They must **not begin with a number**.

Remember java is **case sensitive** temp, Temp, and TEMP are different identifiers.

Valid identifiers

Value, value, _val, Sale, sale_rate, birthDate

Invalid identifiers

1name, copy#, @time, #tag

Naming convention

Type	Rule for naming	Examples
packages	Use lower case	com.example.mypackage
classes	Class names should be nouns , in mixed case with the first letter of each word capitalized .	class Table class Student class ImageSprite
Interface	Similar to class	interface Shape Interface RasterDelegate
Methods	Methods should be verbs, in mixed case with the first letter lowercase , with the first letter of each internal word capitalized	run() talk() drawBackground()
variable	The same as the method but noun . But should not start with underscore _ or dollar sign \$ characters, even though both are allowed	count numberOfNames

Java Separators

Special symbol having special meaning or use in

Symb ol	Name	Description
;	semicolon	Terminate statement
()	Parenthesis	used to contain list of parameters in method definition and invocation . And also used for defining precedence in expression , containing expression in control statements, and surrounding type cast
[]	Bracket	used to declare array and dereferencing array values
{ }	Braces	Used to contains the values automatically initialized array , used to define a block of code , for classes, methods and local scopes
,	comma	Used to separate identifiers in a variable declaration and also to chain statements in a for loop
.	period	Used to separate package name for sub package and classes and to separate an instance variable or method from reference variable
::	colons	Used to create a method or constructor reference
...	Ellipsis	Indicate a variable-arity parameter
@	Ampersand	Begins annotations

Java Keywords

Keywords are words that are reserved by java. These words can be used in other context. Some of the keywords are

abstract	continue	for	new	switch
assert***	default	goto*	package	synchronized
boolean	do	if	private	this
break	double	implements	protected	throw
byte	else	import	public	throws
case	enum****	instanceof	return	transient
catch	extends	int	short	try
char	final	interface	static	void
class	finally	long	strictfp**	volatile
const*	float	native	super	while

Java Data Types

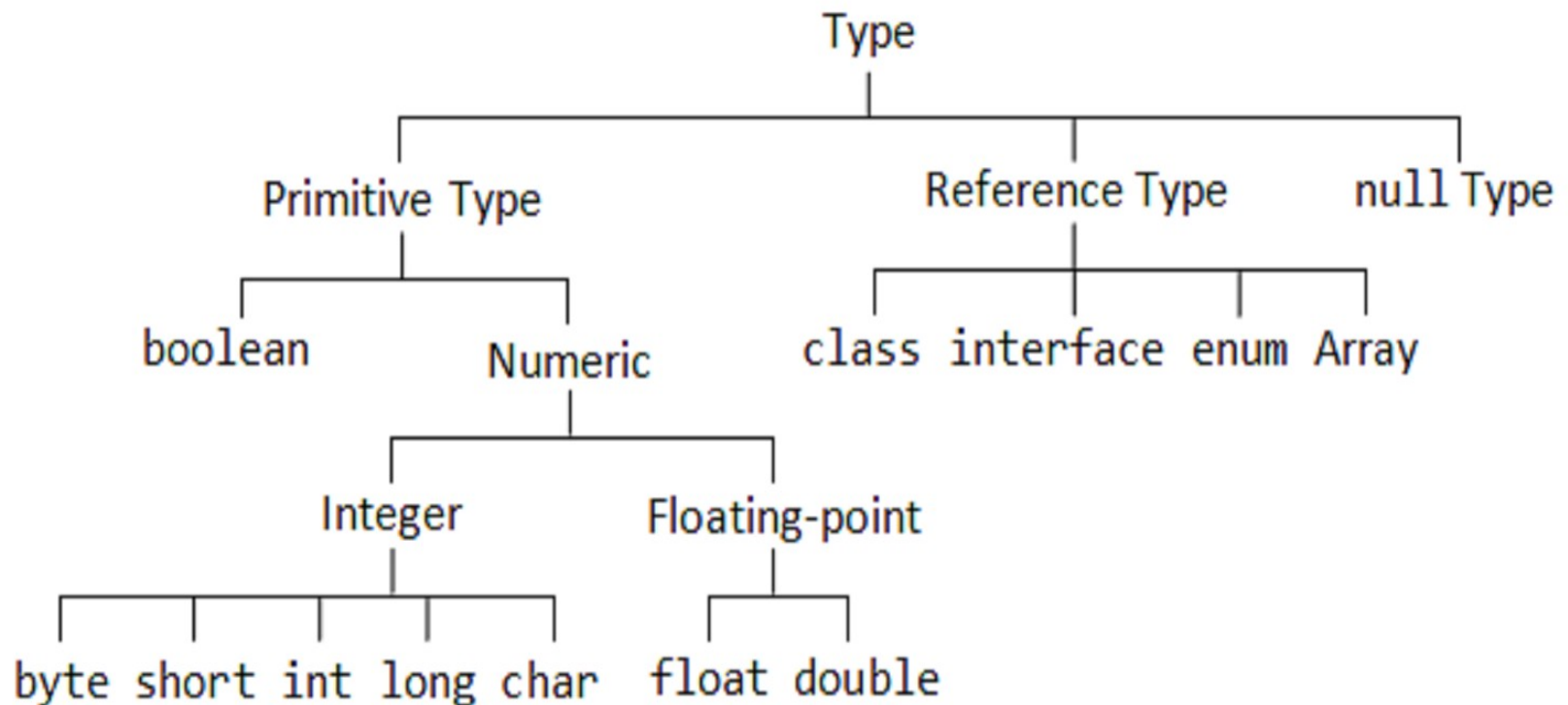
Java Data types

Java is strongly **typed language**. The type of objects must be specified before use.

There three classes of type in Java

- 1. The primitive types:** are the basic types of data which are *pre-defined by the system*
- 2. Reference Types:** those which contains reference/address of dynamically created objects
- 3. Null types:** contains null reference

Java Data types taxonomy



Boolean Type

Java defines a Boolean type that has a value true or false.

Type Name	Width	Range
boolean	1 bit	Stores true or false values

This is the type returned by all relational operators, as in the case of `a < b`

Integers

Java defines four integer types: byte, short, int, and long. All of these are **signed, positive and negative values**. Java **does not support unsigned**, positive only integers.

Type Name	Width	Range
byte	1 Byte	-128 to 127
short	2 Byte	-32,768 to 32,767
int	4 Byte	-2,147,483,648 to 2,147,483,647
long	8 Byte	-9,223,372,036,854,775,808 to 9,223,372,036,854,775,807

int type is the **natural choice** for a computer. Byte and short values in an expression will be converted to int before evaluation.

Java uses 2's complement notation to represent negative numbers

Floating point

Java defines two floating types: `float` and `double`. All of these are signed, positive and negative values

Type Name	Width	Range
float	4 Byte	Stores fractional numbers. Sufficient for storing 6 to 7 decimal digits
double	8 Byte	Stores fractional numbers. Sufficient for storing 15 decimal digits

Character type

Java defines a char type for character. In java char is **unicode**

Type Name	Width	Range
char	2 Byte	Stores a single character/letter or ASCII values

Although char is **designed to hold Unicode characters**, it **can also be used as an integer type** on which you can perform arithmetic operation

Reference Types

Reference types hold **references to objects and provide a means to access those objects stored somewhere in memory**. The memory locations are irrelevant to programmers

```
int number;
```

```
number = 237;
```

```
number = 35;
```

number

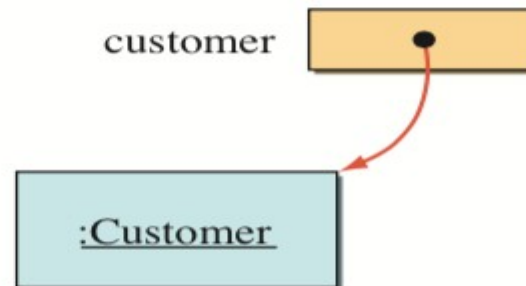
237

```
Customer customer;
```

```
customer = new Customer();
```

```
customer = new Customer();
```

customer



The value of a reference type variable, in contrast to that of a primitive type, is a reference to **(an address of)** the value or set of values represented by the variable

Reference Types...

The five Java reference types are:

1. **Array:** Provides a fixed-size data structure that stores data elements of the same type.
2. **Class:** Designed to provide inheritance, polymorphism, and encapsulation.
3. **Enumeration:** A reference for a set of objects that represents a related set of choices.
4. **Interface:** Provides a public API and is “implemented” by Java classes
5. **Annotation:** Provides a way to **associate metadata (data about data) with program elements.**

Java String

A string is a sequence of characters delimited by a double quote. Strings in Java fall under the category of **Reference type**. The String type is used to declare string variables.

Declaration Syntax

**String identifier[=value]
[,identifier[=value]...];**

Example

```
String fname="Hana", lname="Dawit";
```

```
System.out.println("My name is " + fname + " " +  
lname + ".");
```

The Java String is immutable which means it cannot be changed. Whenever we change any string, a new instance is created. For mutable strings, you can use StringBuffer and StringBuilder

Some Important Java String Methods

Method	Description	Interface
charAt()	Returns the character at the specified index (position)	char charAt(int <i>index</i>)
compareTo()	Compares two strings lexicographically	int compareTo(String <i>string2</i>)
compareToIgnoreCase()	Compares two strings lexicographically, ignoring case differences	int compareToIgnoreCase(String <i>string2</i>)
concat()	Appends a string to the end of another string	String concat(String <i>string2</i>)
contains()	Checks whether a string contains a sequence of characters	boolean contains(CharSequence <i>chars</i>)
equals()	Compares two strings. Returns true if the strings are equal, and false if not	boolean equals(Object <i>anotherObject</i>)

Some Important...

Method	Description	Interface
indexOf()	Returns the position of the first found occurrence of specified characters in a string	<code>int indexOf(String str)</code> <code>int indexOf(String str, int fromIndex)</code> <code>int indexOf(int char)</code> <code>int indexOf(int char, int fromIndex)</code>
length()	Returns the length of a specified string	<code>int length()</code>
replace()	Searches a string for a specified value, and returns a new string where the specified values are replaced	<code>String replace(char searchChar, char newChar)</code>
trim()	Removes whitespace from both ends	<code>String trim()</code>
substring()	Extracts characters, beginning at a specified start position, and through the specified number of character	<code>String substring(int startI ndex)</code> <code>String substring(int startI ndex, int endIndex)</code>

Null type

Java has a special null type. The type **has no name**. As a consequence, it is **impossible to declare a variable of the null** type or to cast to the null type.

The null represents a null reference, one that does not refer to any object.

The null is the default value of reference-type variables.

Primitive types cannot be assigned a null literal

Java Literals

Java literals

There are five literal types in Java:

1. Integral literals
2. Floating Point Literals
3. Character literals
4. Boolean Literals
5. String Literals
6. Null literal

Integral literals

Any whole number value is an integer literal
1, -13, 455556666 are integral literals.

The **default type** for an integral type is an **int**.
if you want store the integral with a long type
append l or L at the end

1l, -13L, 455556666L have long type,

An integral literals can be also represented in
binary, octal, or hexadecimal by adding prefix
0b(0B), 0, or 0x(0X).

17	Decimal
0b10001	binary
021	octal
0xF1	hexadecimal

Floating Point Literals

Floating-point numbers represent decimal values with a **fractional component**. They can be expressed in either **standard or scientific notation**.

121.31, -0.0023 standard or fixed notation

1.2131e2, -2.3E-3 scientific notation

The default type for floating point is double. if you want store the floating point with a float type **append f or F** at the end

121.31f , -2.3E-3F have float type,

Boolean Literals

Boolean literals supports only two value: **true** or **false**.

True is not equal to numerical value 1 and false is not equal 0 as the case in c++. The values of true and false do not convert into any numerical representation.

Character Literals

A character literal represents a value in **Unicode character set**. They can be considered as integers and manipulated with addition, subtraction, multiplication like integers.

A literal character is represented by enclosing it in a pair of single quote.

If the character has a graphics associated, it can be directly represented by putting it inside pair of single quote.

`'a', '@', 'B', '>'`

If no associated graphics, use escape sequence

`'\n'` new line `'\t'` tab `'\v'` vertical tab

`'\\'` back slash `'\b'` backspace `'\"'` double quote

String Literals

String literals in Java are specified by enclosing a sequence of characters between a pair of double quotes.

Example

`"Hello gays", "Java is the best"`

The escape sequence characters can be also added in the string

`"\"This is in quotes\""`

Note Java string literals must begin and end on the same line.

Null Literals

Null literals supports only one value: **null**.
It can be cast to any reference **type**, but not to any primitive **type** such as int or boolean.

Variable

Variables

Variables are containers for storing data values.

They are placeholders for memory location whose content may change as the program executes.

Declaration of variables

In Java, all variables must be declared before they can be used. The syntax for a variable declaration is :

type identifier[=value][, identifier[=value]...];

type-javas elementary types or name of a class or interface

Identifier: name of the variable

optionally the variable can be given initial value using
assignment(=)

Example

```
int x=0, y=0, z;
```

```
float rate=0.26, sale;
```

Note trying to use uninitialized or unassigned variable results in runtime error

Default values

Data Type	Default Value (for fields)
byte	0
short	0
int	0
long	0L
float	0.0f
double	0.0d
char	'\u0000'
String (or any object)	null
boolean	false

Final variable

If you make any variable as final, you cannot change the value of final variable(It will be constant or read only). The syntax for declaring a final variable is

final type identifier[=value][,identifier[=value]...];

```
final int MAX=40; //MAX is read only
```

```
...
```

```
x=14; //generate compilation error
```

a final variable that have no value it is called **blank final variable or uninitialized final variable**

Note: A final value must be given a value on the **declaration time or in the class constructor**. Assigning it in any other place **generate compilation error**.

Type inference with Local Variable

It is possible **to declare a variable without specifying the type** if the variable is initialized and its type can be inferred from its initialization.

Syntax

var identifier=initialization;

Example

var i=3; //i is int type inferred from its initialization value

var y=3.25; //x is double type inferred from its initialization value

var x=3.25f; //x is float type inferred from its initialization value

Note in other context var can be used as identifier

int var=4; //here var is not a context-sensitive-identifier. It is a normal identifier.

It is illegal to use var for reference types. It can be used for simple types and arrays. Only one variable can be declared at

Exercise

Write a simple java program that declare
Two integers variable myVar1 and myVar2 initialized with a value
10 and 20;

Then displays the following output

Original Value

myVar1 = 10, myVar2=20

Then change the content of myVar1 to 20 and myVar2 to 40 and
display

After doubling

myVar1 = 10, myVar2=20

Java Operators

Operators

- **An operator** is a language specific syntactical token that requires an action to be taken.
- C++ provides several operators for composing expression:
 - Basic Arithmetic
 - Increment and decrement
 - Relational and equality
 - Logical
 - Assignment
 - Compound Assignment
 - Bit-wise Logical
 - Ternary
 - Explicit Type Casting

Basic Arithmetic Operators

Operator	Name	Type	Example
- Or +	sign	unary	-x //Gives the negative of the number stored in x
+	Addition	binary	2 + 3.51 //Gives 5.51
-	Subtraction	binary	2 - 3.51 //Gives -1.51
*	Multiplication	binary	2*3.51 //Gives 7.01
/	Division	binary	21 / 4.0 //Gives 5.25
%	Modulo	binary	15 % 6 //Gives 3

Basic Arithmetic...

- All arithmetic operators can accept a **mix of integer and real operands**.
- AS a general rule, if both operands are of the same type then the result will be also of that type.
- Thus, when both operands of the division operator (/) are integers then the division is performed as an **integer division and not the normal division we are used to**. Integer division always results in an integer outcome (i.e., the result is always **rounded down**). For example:
 - 9 / 2 // gives 4, not 4.5!
 - 9 / 2 // gives -5, not -4!
- It is possible for the outcome of an arithmetic operation to be too large for storing in a designated variable. This situation is called an **overflow**. **The outcome** of an overflow is machine-dependent and therefore undefined

Increment/Decrement Operators

The **auto increment (++)** and **auto decrement (--)** operators provide a convenient way of, respectively, adding and subtracting 1 from **a numeric variable**. These are summarized in the table below. The examples assume the following variable definition:

int k = 5;

Operator	Name	Example
++	Prefix increment	Y = ++k + 3 //k will be 6 and y will be 9
++	Postfix increment	Y = k++ + 3 //k will be 6 and y will be 8
i--, --i	Prefix decrement	Y = --k + 3 //k will be 4 and y will be 7
i++, ++i	Postfix decrement	Y = k-- + 3 //k will be 4 and y will be 8

Both operators can be used in prefix and postfix form. The difference is significant. When used in prefix form, the operator is first applied and the outcome is then used in the expression. When used in the postfix form, the expression is evaluated first and then the operator

Bitwise Operators

C++ provides six bitwise operators for **manipulating the individual bits** in an **integer quantity**. Bitwise operators expect their operands to be **integer quantities** and treat them as **bit sequences**.

1. Bitwise **NEGATION(~)** is a **unary operator** which reverses the bits in its operands.
2. Bitwise **AND(&)** **compares the** corresponding bits of its operands and produces a 1 when both bits are 1, and 0 otherwise.
3. Bitwise **OR (|)** **compares the corresponding bits of its** operands and produces a 0 when both bits are 0, and 1 otherwise.
4. Bitwise **EXCLUSIVE OR(^)** **compares the corresponding bits of its operands** and produces a 0 when both bits are 1 or both bits are 0, and 1 otherwise 0.

Bitwise...

5. The bitwise **LEFT SHIFT (<<)** shifts all of the bits in a value to the left a specified number of times. It has the general form
value << num;
num specifies the number of positions to left-shift the value in value
For each shift left, the high-order bit is shifted out (and lost), and a zero is brought in on the right
Shifting left by n places is the same as multiplying by 2^n .
6. The bitwise **RIGHT SHIFT (>>)** does the same thing in the other direction. When a variable is shifted to the right, Java needs to fill the space on the left side with something. In order to preserve the sign of the number, the sign bit is used to fill the gap.
7. The bitwise **Unsigned Right Shift(>>>)** does the same thing as right shift but shifts zeros into the high-order bit

Bitwise...

These are summarized in Table below. In the table below, The examples assume the following variable definition:

```
int cl = 0x45;
```

//000000000000000000000000000000001000101 (69)

```
int c2 = 0x71;
```

Operator	Name	Example
~	Bitwise Negation	Y = ~c1 //Y will be 0xFFFFFBA
&	Bitwise And	Y = c1 & c2 //Y will be 0x41
	Bitwise Or	Y = c1 c2 //Y will be 0x75
^	Bitwise Exclusive Or	Y = c1 ^ c2 //Y will be 0x34
<<	Bitwise Left Shift	Y = c1 << 4 //Y will be 0x450
>>	Bitwise Right Shift	Y = (c1 >> 4) //Y will be 0xFFFFFBA

Relational Operators

Java provides six relational operators for comparing numeric quantities. These are summarized in Table below. The result of a relational operation is a Boolean value that can only be true or false , according to its Boolean result. The relational operators are most frequently used in the expressions that control the if statements and the while loop statements.

Operator	Meaning	Example
==	Equality	3 == 3 //Gives true
!=	Inequality	3 !=3 //Gives false
<	Less Than	3 < 3 //Gives false
<=	Less Than or Equal	3.5 <= 3 //Gives false
>	Greater Than	3.5 > 3.5 //Gives false

Relational Operators...

Note if you are comparing reference objects using the relational equality operator(==), the result is always false and always true for != because you are comparing references, not the objects.

Example:

```
Integer n1 = new Integer(47);  
Integer n2 = new Integer(47);  
System.out.println(n1==n2)  
System.out.println(n1.equals(n2));
```

Output:

```
false  
true
```

Logical Operator

All of the binary logical operators combine two boolean values to form a resultant Boolean value.

Operator	Type	Name	Example			
!	unary	Logical NEGATION	!(3>5) //gives true			
&	binary	Logical AND	7>5&6<4 //gives true			
	binary	Logical OR	7>5 6<4 //gives true			
^	binary	Logical XOR	7>5^6>4 //gives false			
&&	binary	Short Circuit AND	7>5 && 2< 4 //gives true			
	binary	Short Circuit OR	2<5 2 > 4 //gives true			
A	B	A	B	A & B	A B	A ^ B
false	false	true	false	false	false	
true	false	false	true	false	true	
false	true	true	true	false	true	
true	true	false	true	true	false	

Logical...

The difference between the logical and the short-circuit logical operator: Java will evaluate both the left and the right side operand in case of the logical operator but will not bother to evaluate the right-hand operand when the outcome of the expression can be determined by the left operand alone in case of the short-circuit logical operator
what is the output in the following cases?

```
int x=4, y=5, z=6;
```

Case 1

```
System.out.println(x<5 & z++ >y);
```

```
System.out.println("z=" + z);
```

Case 2

```
System.out.println(x<5 && z++ >y);
```

```
System.out.println("z=" + z);
```

Assignment Operator

The assignment operator(=) *is used for storing a value at some*

memory location (typically denoted by a variable).

The part at the left of the assignment operator (=) is known as the ***lvalue (left value)*** and the right one as the ***rvalue (right value)***.

The lvalue has to be a variable whereas the rvalue can be either a **constant**, a **variable**, the result of an operation or any combination of these.

In an assignment statement, first the expression on the right-hand side of the equal sign is evaluated and then the variable on the left-hand side of the equal sign is set equal to this value.

SYNTAX

Variable = Expression;

EXAMPLES

distance = rate * time;

a = 5 //assign 5 to the variable a

5 = a // invalid assignment left side should be an

Compound Assignment

When we want to modify the value of a variable by performing an operation on the value currently stored in that variable we can use compound assignment operators. Its general format is:

variable operator = expression

which is equivalent to

variable = variable operator (expression)

Where variable is a valid variable, operator is an **arithmetic, bitwise or logical operators**, and expression is a valid arithmetic, bitwise or logical expression.

Example

sum /= 2.0;	is equivalent to	sum = sum/2;
n &= 0xF2F2	is equivalent to	n = n & 0xF2F2;
x *= y + 3;	is equivalent to	n = x * (y + 3);

Ternary operator

The conditional operator is a **ternary operator** the takes three operand. Its general format is:

expression1?expression2:expression3

Where expression1 is any expression that evaluates to a boolean value.

First the conditional expression1 is evaluated, which is treated as a logical condition. If the result is false then expression2 is evaluated and its value is the final result. Otherwise, expression3 is evaluated and its value is the final result.

Example 1:

```
int m = 1, n = 2;
```

```
int min = (m < n ? m : n); // min receives 1
```

```
int min = (m < n ? m++ : n++);    //result?
```

Explicit type Casting Operator

Java permits explicit type conversion of variables or expressions using the **type cast operator**. The types should **be compatible** for casting.

Syntax

(type - name) expression;

Examples:

```
int sum=45, i=7;
```

```
float average;
```

```
average = sum / (float) i;
```

Example

```
int i;
```

```
double x=5.1;
```

```
float f = 3.14;
```

```
i = (int) f;           //i contains?
```

```
x = (double) (4.5f*f); //x contains?
```

Explicit type...

Casting can be classified in to two:

1. If you are casting to larger type, it is called **widening casting**
byte -> short -> char -> int -> long -> float -> double
(no explicit casting is required). No loss of information
2. If you are casting to smaller type, it is called **narrowing casting**
double -> float -> long -> int -> char -> short -> byte
May results loss of information

In case of **reference types**, when we assign a **child type object to a parent type**, **no explicit cast is required**.

When assigning a parent type variable to child type variable, an explicit cast is required.

Java Statements

Java Statement

a Java statement is just an instruction that explains what should happen.

Java supports three different types of statements:

- **Expression statements** change values of variables, call methods, and create objects.
 - ; //null statement
 - aValue = 8933.234; //assignment statement
 - aValue++; //increment statement
 - System.out.println(aValue); //method call statement
 - Integer integerObject = new Integer(4); //object creation
- **Declaration statements** declare variables.
 - double aValue = 8933.234; // declaration statement
- **Control-flow statements** determine the order that **statements** are executed.
 - If for while statements

Expression

Expression is a sequence of **operands** and **operators** that reduce to a **single value**.

Example

$5 + 3 * x;$

If the variable x contains 4, the expression reduces to a single value 11.

$y = 5 + 3 * x + \text{Math.sqrt}(x);$

The expression reduces to a single value 19 and y is assigned 19

An operand is the one that receives the operator action. It can be **a constant or a variable or a method call**

Side Effects

When the very act of evaluating an expression changes the value of data in memory, we say the evaluation has a **side effect**.

Thus, evaluating an assignment expression has the side effect of changing the assignee's value.

`x=y + 2.5*y;      // the expression has a side effect`
`y=--a + (3 + b) /2 - c++;    //The expression has`
`three side effects`
`y*x + 3.5 //?`

Operator Precedence

You can specify the order of operations in a complex expression by inserting parentheses. If you omit parentheses, the computer will follow rules called **precedence rules that determine the order in which the operations** are performed. Operators in higher levels take precedence over operators in lower levels. The **associativity** determines the order in which operators with the same precedence are evaluated in a complex expression.

Operator Precedence

Precedence	Operator	Type	Associativity
15	() [] .	Parentheses Array subscript Member selection	Left to Right
14	++ --	Unary post-increment Unary post-decrement	Right to left
13	++ -- + - ! ~ (type)	Unary pre-increment Unary pre-decrement Unary plus Unary minus Unary logical negation Unary bitwise complement Unary type cast	Right to left
12	* / %	Multiplication Division Modulus	Left to right
11	+ -	Addition Subtraction	Left to right
10	<< >> >>>	Bitwise left shift Bitwise right shift with sign extension Bitwise right shift with zero extension	Left to right
9	< <= > >= instanceof	Relational less than Relational less than or equal Relational greater than Relational greater than or equal Type comparison (objects only)	Left to right
8	== !=	Relational is equal to Relational is not equal to	Left to right
7	&	Bitwise AND	Left to right
6	^	Bitwise exclusive OR	Left to right
5		Bitwise inclusive OR	Left to right
4	&&	Logical AND	Left to right
3		Logical OR	Left to right
2	? :	Ternary conditional	Right to left
1	= += -= *= /= % =	Assignment Addition assignment Subtraction assignment Multiplication assignment Division assignment Modulus assignment	Right to left

Java Automatic Type Conversion

An expression that involves more than one data type is called **mixed expression**.

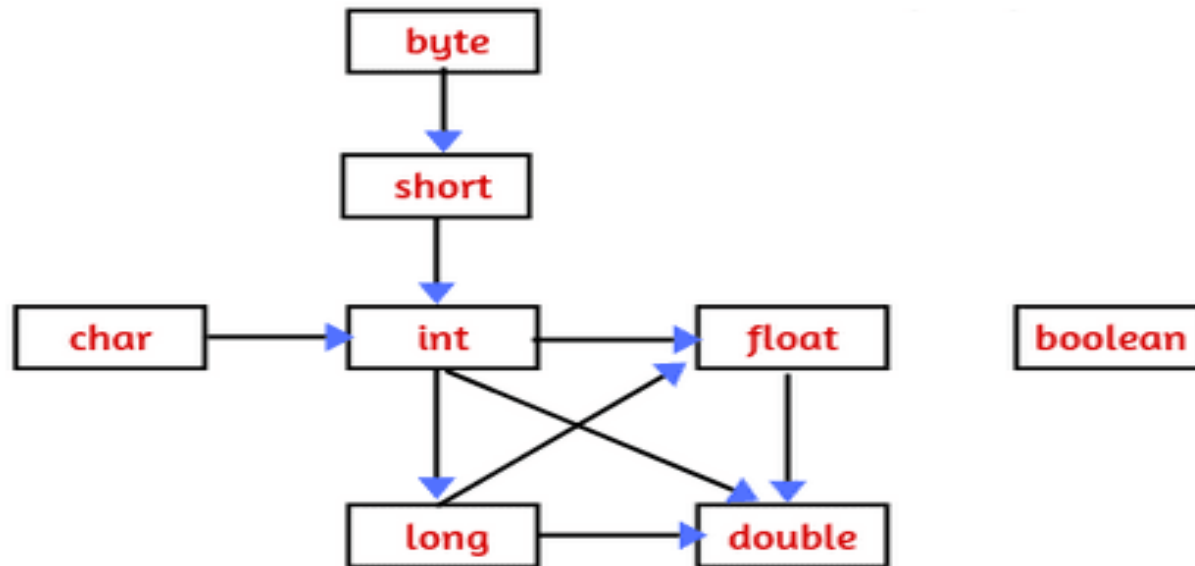
The implicit data type conversion is possible only when:

- The two data types are compatible with each other.
- There is a need to convert a smaller or narrower data type to the larger type size.

In mixed expression, java uses a promotion type conversion rule

- all byte, short, and char values are promoted to int
- If one operand is a long, the whole expression is promoted to long.
- If one operand is a float, the entire expression is promoted to float. If any of the operands are double, the result is double.
- In evaluating assignment expression, if the type of the left variable is narrower in range than the type of

Java Automatic Type Conversion



Example 1

```
float e=2.53;
```

```
int r;
```

```
r = 250 + e*10;    //the type of the evaluated expression?
```

```
type assigned to r
```

Example 2

```
byte b = 50;
```

```
b = b * 2; //will this compile?
```

Java Console Input / Output

The java System class contains three objects **System.out**, **System.err** and **System.in** which can be used for **console output and input**

1. **System.out** is a **PrintStream** that represents the standard output associated by default to screen
2. **System.err** is a **PrintStream** works like System.out except it is normally only used to output error texts.
3. **System.in** is an **InputStream** which is typically connected to keyboard input of console programs

System Out Methods

1. **print**: prints a string. If the argument is not string it will be converted to string

Syntax: ***System.out.print(variable or literal or expression)***

e.g `System.out.print("Java is simple.");`

`System.out.print("sale = " + sale);` // + sign concatenate

2. **println**: prints parameter, and also prints a newline after

Syntax: ***System.out.println(variable or literal or expression)***

3. **printf**: prints formatted text

Syntax: ***System.out.printf(format string, list of parameters)***

e.g `String name="Sara";`

`int age=30,;`

`float height=1.78;`

`System.out.printf("My name is %s, I am %d years`

System.in and Scanner

The java System.in is a byte stream. It reads a byte from the stream. Not very useful

Hence you must wrap a Scanner object around System.in to handle

To use the Scanner class, create an object of the class and use the available methods

Syntax:

```
Scanner input; //input is a reference of  
type Scanner
```

```
input=new Scanner(System.in);
```

or

```
Scanner input= new Scanner(System.in);
```

Methods of Scanner object

Method	Description
nextBoolean()	Reads a boolean value from the user
nextByte()	Reads a byte value from the user
nextDouble()	Reads a double value from the user
nextFloat()	Reads a float value from the user
nextInt()	Reads a int value from the user
nextLine()	Reads a String value from the user
nextLong()	Reads a long value from the user
nextShort()	Reads a short value from the user

Example

```
Scanner reader=new Scanner(System.in);
int age;
String name;
System.out.println("Enter your full name:");
name=reader.nextLine();
System.out.println("Enter your age:");
age=reader.nextInt();
System.out.println("Your name is %s and your
age is %d.", name,age);
```

Exercise

Write a simple java program that accepts the no of items sold , the unit price of the item, and percentage of discount and display the following output:

Payment Receipt :

Quantity:	10	
Unit Price:	15.55	
Discount:		7.76
Total:	147.23	

Flow Control

Control Structure

*The order in which statements are executed is called **flow control (or control flow)**.*

The flow control statements allow us to control the flow of the execution of our code.

Java provides three types of flow control statements

1. **Selection:** allow your program to choose different paths of execution based upon the outcome of an expression or the state of a variable.
2. **Repetition:** enable program execution to repeat one or more statements
3. **Unconditional jumping:** allow your program to execute in a nonlinear fashion

Selection

Java provides two selection statements

- If statement
- Switch statement

Selection if statement

The general structure for this construct is:

```
If(condition 1)
    single or block statement1;
else if(condition 2)
    single or block statement2;
else if(condition 2)
    single or block statement3;
...
...
else
    single or block statementn;
```

Condition1 is evaluated. If it is true, single or block statement1 is executed otherwise statement1 is skipped and condition2 is evaluated and if it is true, single or block statement2 is executed otherwise statement2 is skipped and so on. If none of the condition is evaluated to true the statement under the “else” is executed. Only the statement

One Way and Two Way Selection

If we leave out the else if and else parts we will get a one way selection

If (condition)

 single or block statement;

If we leave out the else parts we will get a two way selection

If (condition)

 single or block statement;

else

 single or block statement;

Exercise

Using the age classification below write a program that accepts the age of a person and display the corresponding age group name.

Group Name	Age Ranges
Babies	0-2
Children	3-16
Young Adults	17-30
Middle-aged Adults	31-45
Old Adults	Above 45

Exercise

A safety program is aimed at assisting selected people with very low income for 10 years in monthly bases. Each individual receives a certain amount and some percentage out of it will be saved for her/him. The following table shows the amount to be provided and to save based on the different age group. Write a program that accepts the age of the person and display the amount that he/she receives and save

Age Ranges	Amount	%save
0-2	500	10%
3-16	400	15%
17-30	250	15%
31-45	200	20%
Above 45	100	5%

Nested if

An **if-else** statement can contain any simple or block statements, including another if-else statement. Therefore, one or more if-else statements can be included in either part of an if-else statement. Including one or more if statements inside an existing if statement is called a

nested if statement.

if (expression1)

if (expression2)

statement 1

else if (expression3)

statement 2

else

statement 3

else

statement 4

```
if (gender == 'M')
    if (age < 21 )
        policyRate = 0.05;
    else
        policyRate =
0.035;
else if (gender == 'F')
    if (age < 21 )
        policyRate = 0.04;
    else
        policyRate = 0.03;
```

Exercise

Repeat exercise but now the amount to receive varies base on gender as shown in the following table.

Age Ranges	Amount(M)		Amount(F)	
%save				
0-2	500	600	10%	
3-16	400	450	15%	
17-30	250	300	15%	
31-45	200	200	20%	
Above 45	400	450	5%	

Switch Selection Statement

The switch statement is another Java's multiway selection statement. It provides an easy way to branch to different parts of your code based on the value of an expression

switch (expression)

{

case value1:

group of statements 1;

break;

case value2:

group of statements 2;

break;

·

·

case valueN:

group of statements N;

break;

default:

default group of statements

break;

*First expression (called the switch **tag**) is evaluated, and the outcome is compared to each of the constants (called case **labels**), in the order they appear, until a match is found.*

*The statements following the matching case are then executed. Execution continues until either a **break** statement is encountered or all intervening statements until the end of the switch statement are executed. The final default case is optional and is exercised if none of the earlier cases provide a match.*

Unlike the multi-way if, the switch can not be used for range value.

expression must resolve to type

Exercise

Write a program that accepts a month and display the corresponding season:

Month	Season
12, 1, 2	Winter
3,4,5	Spring
6,7,8	Summer
9,10,11	Autumn

Exercise

Repeat the previous exercise but now the user provides not the month number but the month name

Month	Season
December, January, February	Winter
March, April, May	Spring
June, July, August	Summer
September, October, November	Autumn

Exercise

Write a program that takes the type of vehicle ('c' or 'C' for car, 'b' or 'B' for bus and 't' or 'T' for truck) and the hour the vehicle spent in the parking lot and display the parking charge based on the rate below:

Car \$2 per hour

Bus \$3 per hour

Truck \$5 per hour

Repetition

Repetition or loop or iteration is any program construction that repeats a statement or sequence of statements a number of times. Each repetition of the loop body is called an **iteration of the loop**.

Java's iteration statements are:

1. While loop
2. do-while loop.
3. For loop

While loop

It repeats a statement or block while its controlling expression is true. Its general form is:

```
while(condition){  
    statement1;  
    statement2;  
    ...  
    ...  
}
```

The body of the loop will be executed as long as the conditional expression is true. When condition becomes false, control passes to the next line of code immediately following the loop. The curly braces are unnecessary if only a single statement is being repeated.

Exercise

Write a program that display the whole number between 1 and n separated by comma where n is supplied by the user

Exercise

Write a program that display the sum of the whole numbers between 1 and where n is supplied by the user

Do-While loop

It repeats a statement or block while its controlling expression is true. Its general form is:

```
do{  
    statement1;  
    statement2;  
    ...  
    ...  
}while(condition);
```

The do while loop is specially important when the body is to be executed at least once as in case of menu display or user input validation

Each iteration of the do-while loop first executes the body of the loop and then evaluates the conditional expression. If this expression is true, the loop will repeat. Otherwise, the loop terminates.

Exercise

Repeat exercises on Slide 95 and 96 using do-while loop. Validate also the input using the do-while.

For loop

Java has two forms for loop:

1. For Loop
2. For Each Loop

The for Loop

A for statement uses three expressions. The first expression contains any initialization statements, the second contains the loop limit test, and the third contains the updating expression. The general syntax for the for loop is:

```
for (initializations ; loop limit test ;  
    updating){  
    statement 1;  
    statement 2;  
    ...  
}
```

For Loop...

- The initializations are done only once at the first entry to the loop. It initializes the loop variable.
- The loop limit test is executed before every iteration.
- The updating is executed at the end of each iteration.
- Using the comma operator you can initialize and update more than one variable.
- The initializations can be made out of the for structure and the updating can be made inside the body of the loop. But the semicolon must be there.

Exercise

Repeat exercise 1 and 2 on slide 95 and 96 using for loop

For-Each(Enhanced loop)

A for-each style loop is designed to cycle through a collection of objects, such as an array, String in strictly sequential fashion, from start to finish.

The general form of the for-each version of the for is shown here:

```
for(type itr-var : collection){  
    statement 1;  
    statement 2;  
    ...  
    ...  
}
```

If you try to change itr-var inside the block, the change will affect the original element in the Collection if it is reference type but the original element will not be affected if it is primitive types.

Where type is the type of a collection

itr-var is the iteration variable used to refer the current element in the collection

For-Each example

```
int [] arr={12, 20, 50,54, 72, 101};  
for(int x:arr){  
    System.out.println(x);  
}
```

The above code prints all the values in arr.

Exercise

Write a program that computes the check sum value for a string using for each loop

Break statement

The break statement has three uses:

1. Terminates a statement sequence in a switch statement

2. Can be used to exit a loop

When used inside a set of nested loops, the break statement will only break out of the inner most loop.

3. As a form of goto

The general form of the labeled break statement is shown here:

`break label;`

label is the name of a label that identifies a **block of code**

To name a block, put a label at the start of it. A label is any valid Java identifier followed by a colon

Break statement

```
class sum {
    public static void main(String args[]){
        int num, i=0,sum=0;
        do{
            System.out.println("Enter next
number");
            Scanner reader=new
Scanner(System.in);
            num=reader.nextInt();
            if(num<0) break;
            sum += num;
            i++;
        }while(i<100);
        System.out.printf("Sum of the numbers:
%d\n", sum);
    }
}
```

Break statement

```
class Break {  
    public static void main(String args[]){  
        bool t;  
        first:{  
            second:{  
                third:{  
                    System.out.println("Before break");  
                    if(t) break second;  
                    System.out.println("This won't execute.");  
                }  
                System.out.println("This won't execute.");  
            }  
            System.out.println("This is after second block.");  
        }  
    }  
}
```

Continue statement

This statement allows to continue running the loop but stop processing the remainder of the code in its body for this particular iteration

Continue statement

```
class sumOfEven{
    public static void main(String args[]){
        int i,sum;
        System.out.println("Enter next number");
        Scanner reader=new
Scanner(System.in);
        num=reader.nextInt();
        for(i=0, sum=0; i<num; i++){
            if(i%2!=0) continue;
            sum +=i;
        }
        System.out.printf("Sum of evens: %d\n",
sum);
    }
}
```

As with the break statement, continue may specify a label to describe which enclosing loop to continue.

Return

The return statement is used to explicitly return from a method. That is, it causes program control to transfer back to the caller of the method.

Java Array

Array

Array is an elementary data structure that exists as built-in in most programming languages.

Array is a **collection structure** composed of a **fixed number of elements** wherein all of the elements have **the same data type**.

The elements are stored in **contiguous computer memory** so that it allows **random access** of its elements, i.e. elements can be accessed using indexing by address calculation.

Arrays are suitable for representing data which consist of many **similar, individual items**. Examples include: a list of names, a table of world cities and their current temperatures, or the monthly

Array Categorization

Array can be categorized into two part

1) One dimensional Array:

collection of elements (of the same type)
arranged in one dimension as a list

2) Multi dimensional :

collection of elements (of the same type)
arranged in multi dimensions

Declaration and Initialization of One Dimensional array

The general form of a one-dimensional array declaration is

**type identifier[]; or
type [] identifier;**

type specifies the type of the array elements

identifier is the common name for the array elements. A particular element is referenced by indexing.

The statement creates **a reference or name** for the array. No array is created by¹¹⁶

Declaration and Initialization ...

To create the array we use the **new** operator

identifier=new type[size];

type specifies the type of the array elements

identifier is the array reference

Size specifies the number of elements in the array

The elements in the array allocated by new will automatically be initialized to zero (for numeric types), false (for boolean), or null (for reference types)

Example **This creates an array for 50 elements of float**
All of them initialized by zero

```
float score[];
```

```
score=new float[50];
```

Declaration and Initialization...

The declaration and the creation of the array can be combined to a single statement

**type identifier[]=new type[size]; or
type [] identifier=new type[size];**

Array can be initialized during declaration

**type identifier[]={value1, value2,
value3,...,valuen};**

The new statement is not needed in here.
The array will automatically be created
large enough to hold the number of
elements you specify in the array
initializer

Accessing an element of array

Individual an element of an array can be accessed by indexing. Indexing is based 0. To refer the element at index i on the array arr

arr[i]

Example

```
int myArray[]={2, 4, 6, 8, 10, 12, 14, 15, 18,20};
```

To display the 3rd element

```
System.out.println("The third element is: " +  
myArray[2]);
```

To fix or change the odd value 15 to 16

```
myArray[7]=16;
```

Attempting to access an array beyond the bounds of the array will result in a runtime exception,

Size of an Array

To determine the size of an array

arr.length;

Where arr is the array identifier

Example

```
String names[]={“Abebe”, “Dawit”,  
“Lulit”, “Beti”};
```

```
for(int i=0; i<names.length; i++){  
    System.out.println(names[i]);  
}
```

Java Arrays Class

Java Arrays class consists exclusively of static methods that operates on array

To use these methods `java.util.Arrays` must be imported

Method	Description	Interface
<code>sort()</code>	Sort the array of type type	<code>void sort(type[] a)</code>
<code>fill()</code>	Fill the array of type type with value val	<code>void fill(type[] a, type val)</code>
<code>equals()</code>	Compares two array in terms of size and value	<code>boolean equals(type[] a1, type[] a2)</code>
<code>binarySearch()</code> Example:	Search a sorted array a for key value and returns an index	<code>int binarySearch(type[] a, type key)</code>

```
Double [] arr={2.3, 1.2, 5.6, 1.2, 0.4 3.2};
```

```
Arrays.sort(arr);
```

```
System.out.println("Position: " +
```

```
Arrays.binarySearch(1.2));
```

Exercise

1. Write a simple program that accepts n names from the user, sort the names and prints the sorted names
 - a. $A + B$
 - b. $A - B$
 - c. $A * B$

Declaration and Initialization of Two Dimensional array

An two dimensional array is **an array of one dimensional array**

The general form of a two-dimensional array declaration is

**type identifier[][]; or
type [][] identifier;**

type specifies the type of the array elements
identifier is the common name for the two dimensional array elements.

A particular element is referenced by row and column index.

The statement creates **a reference or name** for the array.

No array is created by the statement

Declaration and Initialization ...

To create the array we use the **new** operator

identifier=new type[size1][size2];

Size **size1**, **size2**: the row and column sizes of the dimensions respectively.

Example

float score[][]; **This creates an array for 200**

score=new float[50][4]; **elements of float type. All of**

Two Dimensional Array can be initialized during declaration **them initialized by zero**

type identifier[][]={{value1, value2, value3},{value4, value5, value6},...};

Example

```
int matrix[][]={{1,2,3},{4,5,6},{7,8,9},
{10,11,12}};
```

Accessing An Element of two Dimensional Array

Individual element of a two dimensional array can be accessed by using two index representing the row and column. Indexing is based 0. To refer the element at row index *i* and at column index *j*

`arr[i][j]`

Example

```
int myArray[][]={{2, 4, 6}, {8, 10, 12}, {14, 15, 18}};
```

To display the element at row 2 and column 3
`System.out.println("The third element is: " + myArray[1][2]);`

To fix or change the odd value 15 to 16

```
myArray[2][1] = 16;
```

Size of a 2D Array

To determine the size of a 2D array

```
int matrix[][]={{1,2,3},{4,5,6},{7,8,9},  
               {10,11,12}};
```

```
System.out.println(matrix.length); //prints row size
```

```
System.out.println(matrix[0].length); //prints  
column size
```

Exercise

1. Write a menu driven program that reads two matrix, A and B, and then compute the following based on the user choice.
 - a. $A + B$
 - b. $A - B$
 - c. $A * B$

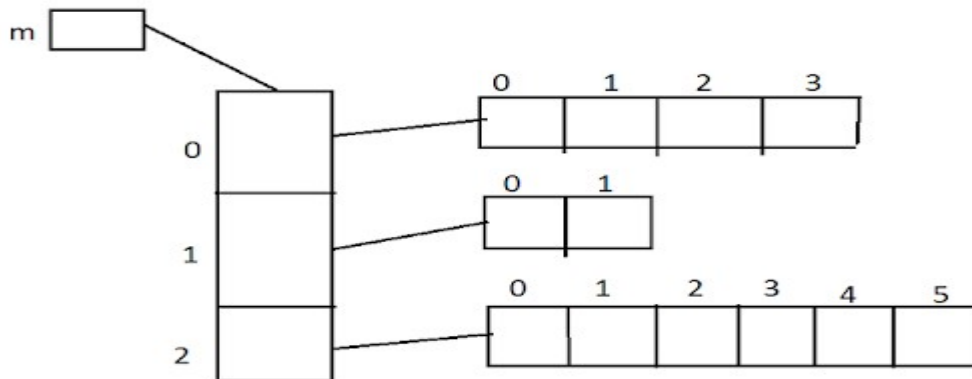
Ragged Two Dimensional Array

To create the array we use the **new** operator

identifier=new type[size1][]; Size1 is fixed.
The second dimension size is unknown.

identifier[0]=new type[size2₁]; //creating
the first row

identifier[1]=new type[size2₂]; //creating
the second row



new type[size2_{size1}];

Ragged...

Example

To create the ragged array in the right side

```
int rarr[][];
```

```
rarr=new int[3][];
```

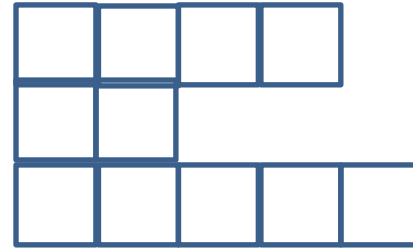
```
rarr[0]=new int[4];
```

```
rarr[1]=new int[2];
```

```
rarr[2]=new int[5];
```

Or

```
int rarr[][]={ {0,0,0,0}, {0,0}, {0,0,0,0,0}};
```



Declaration of Multi-Dimensional Array

The general form of a two-dimensional array declaration is

type identifier[][]...[] = new type[size₁][size₂] ... [size_n]

type specifies the type of the array elements
identifier is the common name for the two dimensional.

size₁, size₂, ..., size_n: is the size of each dimension

A particular element is referenced by n indexes.