Open University of Mauritius

# OBJECT ORIENTED PROGRAMMING

## Table of Contents

# COURSE OVERVIEW

## Introduction

This module consists of 16 units in the field of Object Oriented Programming. OBJECT-ORIENTED PROGRAMMING (OOP) represents an attempt to make programs more closely model the way people think about and deal with the world. In the older styles of programming, a programmer who is faced with some problem must identify a computing task that needs to be performed in order to solve the problem. Programming then consists of finding a sequence of instructions that will accomplish that task. But at the heart of object-oriented programming, instead of tasks, we find objects – entities that have behaviors that hold information and that can interact with one another. Programming consists of designing a set of objects that model the problem at hand. Software objects in the program can represent real or abstract entities in the problem domain. This is supposed to make the design of the program more natural and hence easier to get right and easier to understand.

An object-oriented programming language such as JAVA includes a number of features that make it very different from a standard language. In order to make effective use of those features, you have to "orient" your thinking correctly.

## Course Aims and Objectives

The aim of this module is to provide learners with knowledge of Object Oriented concepts such that the learners can develop on any Object Oriented platform, be it Open Source technology or Proprietary.

## Learning Outcomes

Upon completion of this course, learners should be able to:

- Compile and run Java programs using either Eclipse or NetBeans
- Learn the different data types employed in programming
- Make use of operators and expressions
- Create methods and pass parameters
- Perform Selection and Iteration to solve problems
- Use data structures such as arrays and collections appropriately
- Understand the core concepts of OOP
- Grasp the concepts of objects and classes as used in OOP
- Put in practice the three core concepts of OOP – Encapsulation, Inheritance and Polymorphism
- Understand errors generation and deploy exception handling mechanisms

### Assignment and Assessment

The course is assessed on continuous assessment and a final written examination. The weight of the continuous assessment (assignment + online activities) is 40% and that of the examination is 60%.

## STUDENT SUPPORT AND ASSISTANCE

### Academic Support

Online support via the university LMS is accessible on www.oum.ac.mu . Face-to-Face sessions are held every three weeks whereby tutorials and answers to the units' activities will be provided.

### Study guidelines

Learners are advised to:

- Read the unit topics completely before attempting the activities
- Seek to meet the learning outcomes
- Spend some extra reading time with the recommended textbooks
- Prepare your questions for the face to face sessions
- Collaborate and share with your peers online and in the classroom

# UNIT 1 JAVA Development Tools

## 1.1. Introduction

This unit gives an overview of Java as a programming language and its evolution. This unit will also introduce the students to various development tools that support the development of large-scale Java systems (e.g. Eclipse and NetBeans). The importance of a compiler is also explained.

## 1.2. Unit Objectives

The objectives of this unit are to:

- Learn the History of Java
- Discover the Features of Java
- Understand the roles of the JDK and JRE
- Investigate professional development environments (Eclipse and NetBeans)

## 1.3. Programming Language: Java

Java programming language was originally developed by Sun Microsystems which was initiated by James Gosling and released in 1995 as core component of Sun Microsystems' Java platform (Java 1.0 [J2SE]). As of March 2021, the latest release of the Java Standard Edition is 16 (J2SE). With the advancement of Java and its widespread popularity, multiple configurations were built to suite various types of platforms. Ex: J2EE for Enterprise Applications, J2ME for Mobile Applications. Sun Microsystems has renamed the new J2 versions as Java SE, Java EE and Java ME, respectively. Java is guaranteed to be Write Once, Run Anywhere.

Java is:

• **Object Oriented**: In Java, everything is an Object. Java can be easily extended since it is based on the Object model.

• **Platform independent**: Unlike many other programming languages including C and C++, when Java is compiled, it is not compiled into platform specific machine, rather into platform independent byte code. This byte code is distributed over the web and interpreted by Java Virtual Machine (JVM) on whichever platform it is run.

- **Simple**: Java is designed to be easy to learn. If you understand the basic concept of OOP, Java would be easy to master.

- **Secure**: With Java's secure feature, it enables to develop virus-free, tamper-free systems. Authentication techniques are based on public-key encryption.

- **Architectural-neutral:** Java compiler generates an architecture-neutral object file format, which makes the compiled code to be executable on many processors, with the presence of Java runtime system.

- **Portable**: Being architectural-neutral and having no implementation dependent aspects of the specification makes Java portable. Compiler in Java is written in ANSI C with a clean portability boundary that is a POSIX subset.

- **Robust**: Java makes an effort to eliminate error prone situations by emphasizing mainly on compile time error checking and runtime checking.

- **Multithreaded**: With Java's multithreaded feature, it is possible to write programs that can do many tasks simultaneously. This design feature allows developers to construct smoothly running interactive applications.

- **Interpreted**: Java byte code is translated on the fly to native machine instructions and is not stored anywhere. The development process is more rapid and analytical since the linking is an incremental and lightweight process.

- **High Performance**: With the use of Just-In-Time compilers, Java enables high performance.

- **Distributed**: Java is designed for the distributed environment of the internet with IP, ports, TCP, UDP, etc. addressing.

- **Dynamic**: Java is considered to be more dynamic than C or C++ since it is designed to adapt to an evolving environment. Java programs can carry extensive amount of run-time information that can be used to verify and resolve accesses to objects on run-time.

## 1.4. Terminologies

- **Object** - Objects have states and behaviors. Example: A dog has states-color, name, breed as well as behaviors - wagging, barking and eating. An object is an instance of a class.

- **Class** - A class can be defined as a template/blueprint that describes the behaviors/states that object of its type support.

• **Methods** - A method is basically a behavior. A class can contain many methods. It is in methods where the logics are written, data is manipulated and all the actions are executed.

• **Instance Variables** - Each object has its unique set of instance variables. An object's state is created by the values assigned to these instance variables.

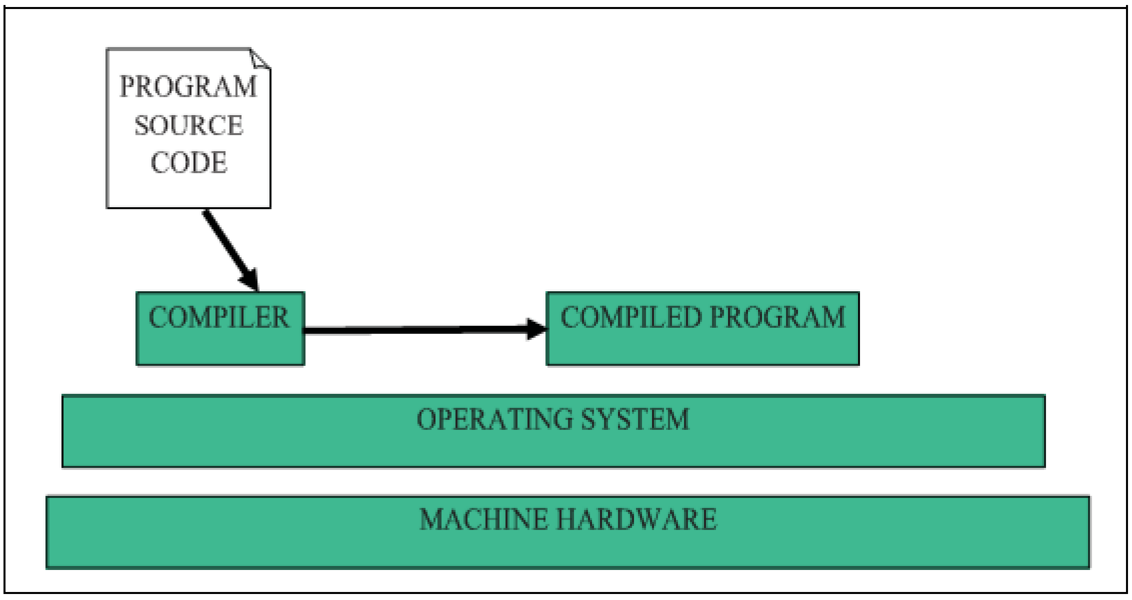
## 1.5. Software Implementation

Before a computer can complete useful tasks for us, (e.g. check the spelling in our documents) software needs to be written and implemented on the machine it will run on. Software implementation involves the writing of program source code and preparation for execution on a particular machine. Of course, before the software is written, it needs to be designed and at some point, it needs to be tested. There are many iterative lifecycles to support the process of design, implementation and testing that involve multiple implementation phases. Of particular concern, here are three long established approaches to getting source code to execute on a particular, namely: compilation into machine-language object code, direct execution of source code by 'interpreter' program and compilation into intermediate object code that is then interpreted by run-time system.

Implementing Java programs involves compiling the source code into intermediate object code that is then interpreted by a run-time system called the JRE. This approach has some advantages and disadvantages and it is worth comparing these three options in order to appreciate the implications for Java developer.

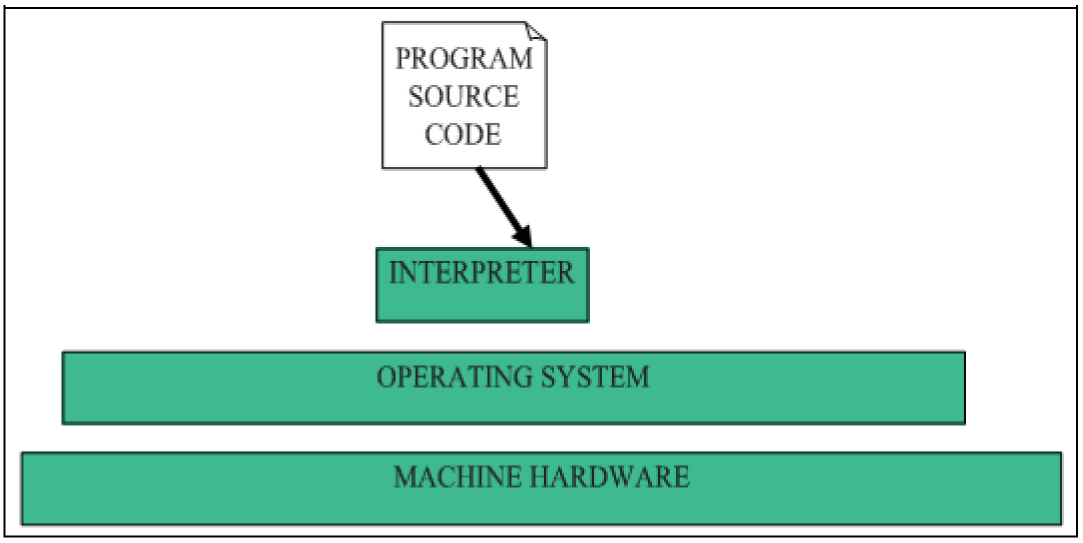
**Compilation**

The compiler translates the source code into machine code for the relevant hardware/OS combination. Strictly speaking, there are two stages: compilation of the program units, followed by linking when the complete executable program is put together including the separate program units and relevant library code etc.



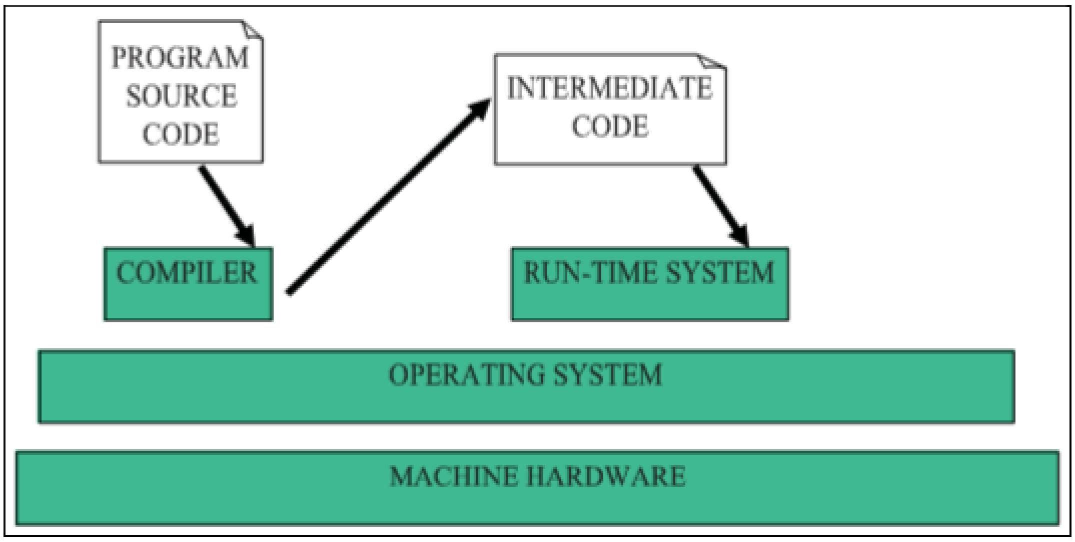
## Interpretation

Here the source code is not translated into machine code. Instead, an interpreter reads the source code and performs the actions it specifies. No re-compilation is required after changing the code, but the interpretation inflicts a significant impact on execution speed.



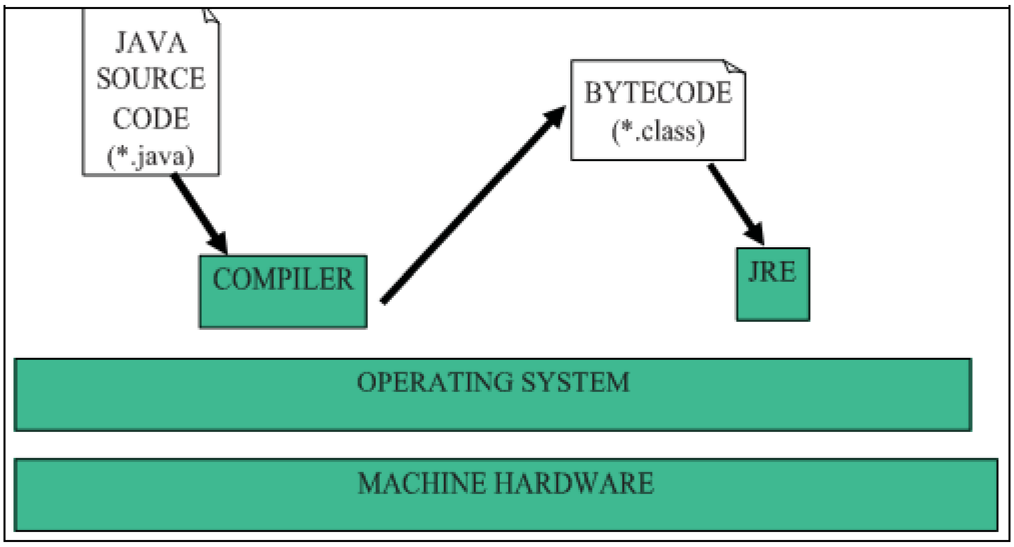
## Intermediate Code

This model is a hybrid between the previous two. Compilation takes place to convert the source code into a more efficient intermediate representation that can be executed by a 'run-time system' more quickly than direct interpretation of the source code. However, the use of an intermediate code that is then executed by run-time system software allows the compilation process to be independent of the OS/hardware platform.



## 1.6. The JRE

To run Java programs, we must first generate intermediate code (called bytecode) using a compiler available as part of the Java Development Kit (JDK).

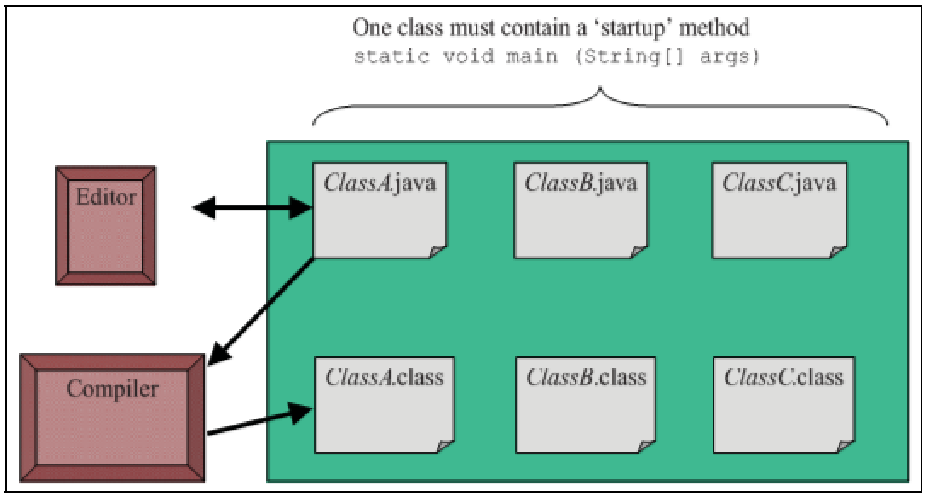


A version of the Java Runtime Environment, which incorporates a Java Virtual Machine, is required to execute the bytecode and the Java library packages. Thus, a JRE must be present on any machine that is to run Java programs.

The Java bytecode is standard and platform independent and as JRE's have been created for most computing devices (including PCs, laptops, mobile devices, mobile phones, internet devices etc.) this makes Java programs highly portable.

## 1.7. Java programs

When writing java programs, each class in a Java program has its own name.java file containing the source code. These are processed by the compiler to produce name.class files containing the corresponding bytecode. To actually run as an application, one of the classes must contain a main() method with the signature shown below.



## 1.8. JDK

To develop Java programs, you must first install the Java Development Kit (JDK). This was developed by Sun and is available freely from the internet via http://java.su.com/ . Prior to version 5.0, this was known as the Java Software Development Kit (SDK).

In the past, we used the JDK directly from the command line to compile and run programs. Nowadays, it is easier to use the additional facilities offered by an IDE. Java IDEs, e.g. Eclipse or NetBeans, sits on top of the JDK and include features like interface development tools, syntax checking, code debugging tools, testing tools and refactoring tools.

## 1.9. Compiling & running programs using Eclipse

Eclipse is a vast extendable set of tools for software development. Here we are interested in Eclipse's Integrated Development Environment (IDE) component for writing Java software. Eclipse is an open source project of Eclipse Foundation; you can find information about Eclipse Project at http://www.eclipse.org/eclipse. Eclipse is available free of charge under the Eclipse Public License.

Eclipse was developed by software professionals for software professionals; it may seem overwhelming to a novice. This document describes the very basics of Eclipse, enough to get started with Java in an educational setting. Eclipse runs on multiple platforms including Windows, Linux, and Mac OS. There may be minor differences between Eclipse versions for different platforms and operating systems, but the core features work the same way. Here we will use examples and screen shots from Windows.

### 1.10. Compiling & running programs using Netbeans

NetBeans is another very powerful IDE, originally developed and distributed by Sun Microsystems, Netbeans was made open source in 2000 and is extensively supported with plug-ins and video tutorials developed by the open source community. NetBeans can be downloaded for free from www.netbeans.org.

NetBeans has many features that have been developed to aid novice programmers and save time. These include:

- Help features with automatic pop-up windows highlighting relevant parts of the Java API
- Formatting and debugging facilities
- Code completion features that will auto insert for loops, try catch blocks and constructors and so on.
- Project management features that monitor changes to a project that allow the project to be reverted to an earlier stage.

### 1.11. Summary

- Understand the roles of the JDK and JRE
- Compile and run a simple Java program using JDK, Eclipse and NetBeans.
- Investigate professional development environments (Eclipse and NetBeans)

### 1.12. Activities

1. Compile and run the following java codes using Eclipse and NetBeans.

```
public class HelloWorld {
   public static void main(String[] args) {
      System.out.println("Hello, World");
   }
}
```

UNIT 2

# Data and Variable Types

## 2.1 Introduction

Variables are nothing but reserved memory locations to store values. This means that when you create a variable you reserve some space in memory.

Based on the data type of a variable, the operating system allocates memory and decides what can be stored in the reserved memory. Therefore, by assigning different data types to variables, you can store integers, decimals, or characters in these variables.

There are two data types available in Java:

- Primitive Data Types
- Reference/Object Data Types

A variable provides us with named storage that our programs can manipulate. Each variable in Java has a specific type, which determines the size and layout of the variable's memory; the range of values that can be stored within that memory; and the set of operations that can be applied to the variable.

This unit will explain various variable types available in Java Language. There are three kinds of variables in Java:

- Local variables
- Instance variables
- Class/static variables

## 2.2 Unit Objectives

The objectives of this unit are to:

- Illustrate the types of variables
- point out the need for different Data Types
- describe Data Types: byte, short, int, long, float, double and char
- highlight the importance of Reference Data Types and Literals.

## 2.3 Primitive Data types

There are eight primitive data types supported by Java. Primitive data types are predefined by the language and named by a keyword. Let us now look into detail about the eight primitive data types.

### byte:

- Byte data type is an 8-bit signed two's complement integer.
- Minimum value is -128 (-2^7)
- Maximum value is 127 (inclusive)(2^7 -1)
- Default value is 0
- Byte data type is used to save space in large arrays, mainly in place of integers, since a byte is four times smaller than an int.
- Example: byte a = 100, byte b = -50

### short:

- Short data type is a 16-bit signed two's complement integer.
- Minimum value is -32,768 (-2^15)
- Maximum value is 32,767(inclusive) (2^15 -1)
- Short data type can also be used to save memory as byte data type. A short is 2 times smaller than an int.
- Default value is 0.
- Example: short s= 10000, short r = -20000

### int:

- int data type is a 32-bit signed two's complement integer.
- Minimum value is - 2,147,483,648.(-2^31)
- Maximum value is 2,147,483,647(inclusive).(2^31 -1)
- int is generally used as the default data type for integral values unless there is a concern about memory.
- The default value is 0.
- Example: int a = 100000, int b = -200000

### long:

- Long data type is a 64-bit signed two's complement integer.
- Minimum value is -9,223,372,036,854,775,808.(-2^63)
- Maximum value is 9,223,372,036,854,775,807 (inclusive). (2^63 -1)
- This type is used when a wider range than int is needed.
- Default value is 0L.
- Example: int a = 100000L, int b = -200000L

### float:

- Float data type is a single-precision 32-bit IEEE 754 floating point.
- Float is mainly used to save memory in large arrays of floating point numbers.
- Default value is 0.0f.
- Float data type is never used for precise values such as currency.
- Example: float f1 = 234.5f

### double:

- double data type is a double-precision 64-bit IEEE 754 floating point.
- This data type is generally used as the default data type for decimal values, generally the default choice.
- Double data type should never be used for precise values such as currency.
- Default value is 0.0d.
- Example: double d1 = 123.4

### boolean:

- boolean data type represents one bit of information.
- There are only two possible values: true and false.
- This data type is used for simple flags that track true/false conditions.
- Default value is false.
- Example: boolean one = true

### char:

- char data type is a single 16-bit Unicode character.
- Minimum value is '\u0000' (or 0).
- Maximum value is '\uffff' (or 65,535 inclusive).
- Char data type is used to store any character.
- Example: char letterA ='A'

**Final and static variable**

When a variable is declared with final keyword, its value can’t be modified, essentially, a constant. This also means that you must initialize a final variable.

The static keyword in Java is used for memory management mainly. The static variable can be used to refer to the common property of all objects (which is not unique for each object), for example, the company name of employees, college name of students, etc.
The static variable gets memory only once in the class area at the time of class loading.

## 2.4 Reference Data types

Reference variables are created using defined constructors of the classes. They are used to access objects. These variables are declared to be of a specific type that cannot be changed. For example, Employee, Puppy, etc.

Class objects and various types of array variables come under reference data type. Default value of any reference variable is null. A reference variable can be used to refer to any object of the declared type or any compatible type. Example:
Animal animal = new Animal("giraffe");

### 2.5 Literals

A literal is a source code representation of a fixed value. They are represented directly in the code without any computation.

Literals can be assigned to any primitive type variable. For example:

```
byte a =68;

char a ='A';
```

byte, int, long, and short can be expressed in decimal(base 0), hexadecimal (base 16) or octal(base 8) number systems as well.

Prefix 0 is used to indicate octal and prefix 0x indicates hexadecimal when using these number systems for literals.

For example:

```
int decimal=100;

int octal =0144;

int hexa =0x64;
```

String literals in Java are specified as they are in most other languages by enclosing a sequence of characters between a pair of double quotes. Examples of string literals are:

```
"Hello World"

"two\nlines"

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

String and char types of literals can contain any Unicode characters. For example:

```
char a ='\u0001';

String a ="\u0001";
```

Java language supports few special escape sequences for String and char literals as well. They are:

| Notation | Character represented |
|---|---|
| \n | Newline (0x0a) |
| \r | Carriage return (0x0d) |
| \f | Formfeed (0x0c) |
| \b | Backspace (0x08) |
| \s | Space (0x20) |
| \t | Tab |
| \" | Double quote |
| \' | Single quote |
| \\ | Backslash |
| \ddd | Octal character (ddd) |

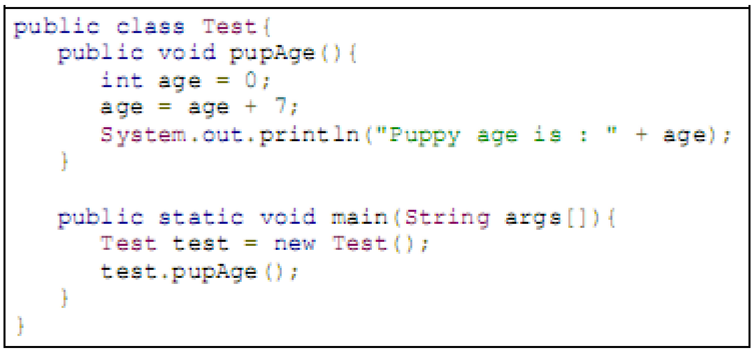
## 2.6 Local Variables

Local variables are declared in methods, constructors, or blocks. Local variables are created when the method, constructor or block is entered and the variable will be destroyed once it exits the method, constructor or block. Access modifiers cannot be used for local variables.

Local variables are visible only within the declared method, constructor or block. Local variables are implemented at stack level internally. There is no default value for local variables so local variables should be declared and an initial value should be assigned before the first use.

**Example:**

Here, age is a local variable. This is defined inside pupAge() method and its scope is limited to this method only.



```
public class Test{
   public void pupAge(){
      int age = 0;
      age = age + 7;
      System.out.println("Puppy age is : " + age);
   }

   public static void main(String args[]){
      Test test = new Test();
      test.pupAge();
   }
}
```

This would produce the following result:

**Puppy age is: 7**

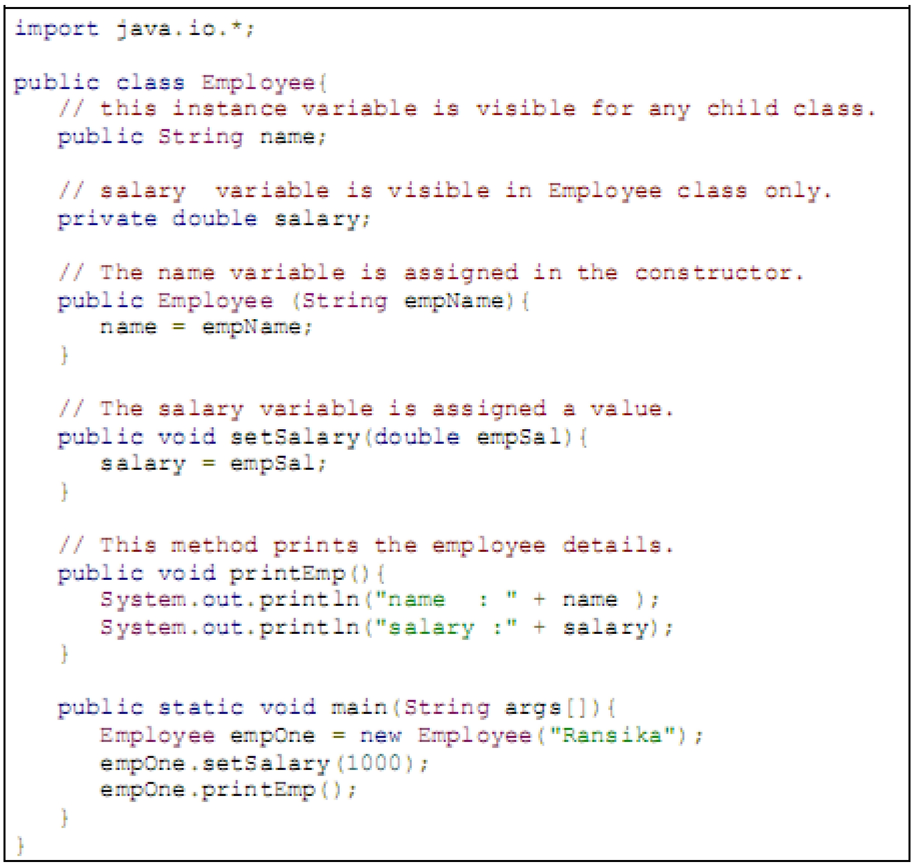
## 2.7 Instance Variables

Instance variables are declared in a class, but outside a method, constructor or any block. When a space is allocated for an object in the heap, a slot for each instance variable value is created. Instance variables are created when an object is created with the use of the keyword 'new' and destroyed when the object is destroyed. They hold values that must be referenced by more than one method, constructor or block, or essential parts of an object's state that must be present throughout the class.

The instance variables are visible for all methods, constructors and block in the class. Normally, it is recommended to make these variables private (access level). However, visibility for subclasses can be given for these variables with the use of access modifiers.

Instance variables have default values. For numbers the default value is 0, for Booleans it is false and for object references it is null. Values can be assigned during the declaration or within the constructor.

**Example:**



```
import java.io.*;

public class Employee{
   // this instance variable is visible for any child class.
   public String name;

   // salary  variable is visible in Employee class only.
   private double salary;

   // The name variable is assigned in the constructor.
   public Employee (String empName){
      name = empName;
   }

   // The salary variable is assigned a value.
   public void setSalary(double empSal){
      salary = empSal;
   }

   // This method prints the employee details.
   public void printEmp(){
      System.out.println("name  : " + name );
      System.out.println("salary :" + salary);
   }

   public static void main(String args[]){
      Employee empOne = new Employee("Ransika");
      empOne.setSalary(1000);
      empOne.printEmp();
   }
}
```

This would produce the following result:

**name : Ransika**

**salary :1000.0**

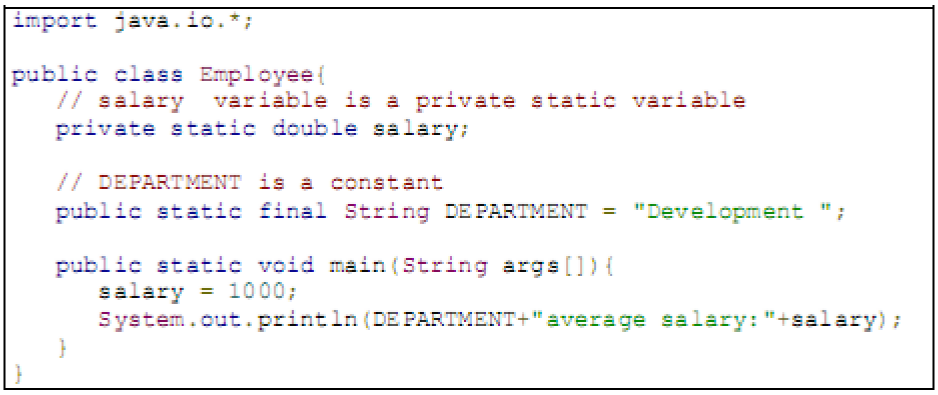
## 2.8 Class/static variables

Class variables also known as static variables are declared with the static keyword in a class, but outside a method, constructor or a block. There would only be one copy of each class variable per class, regardless of how many objects are created from it.

Static variables are rarely used other than being declared as constants. Constants are variables that are declared as public/private, final and static. Constant variables never change from their initial value.

Static variables are stored in static memory. It is rare to use static variables other than declared final and used as either public or private constants. Static variables are created when the program starts and destroyed when the program stops.

**Example:**



```java
import java.io.*;

public class Employee{
   // salary  variable is a private static variable
   private static double salary;

   // DEPARTMENT is a constant
   public static final String DEPARTMENT = "Development ";

   public static void main(String args[]){
      salary = 1000;
      System.out.println(DEPARTMENT+"average salary:"+salary);
   }
}
```

This would produce the following result:

**Development average salary: 1000**

## 2.9 Summary

- There are three kinds of variables in Java:
  - Local variables
  - Instance variables
  - Class/static variables

- The main Data types are byte, short, int, long, float, double and char.
- Reference variables are created using defined constructors of the classes

## 2.10 Activities

1. How do I print a "?

2. OK, so then how do I print a \?

3. Are there any restrictions on the variable names I can use?

4. Suppose that a and b are int values. What does the following sequence of statements do?

```
int t = a;
b = t;
a = b;
```

5. Why does 10/3 give 3 and not 3.33333333?

6. What do each of the following print?

   a. System.out.println(2 + "bc");

b. System.out.println(2 + 3 + "bc");
c. System.out.println((2+3) + "bc");
d. System.out.println("bc" + (2+3));
e. System.out.println("bc" + 2 + 3);

7. Suppose that a variable a is declared as double a = 3.14159. What do each of the following print?
   b) System.out.println(a);
   c) System.out.println(a + 1);
   d) System.out.println(8 / (int) a);
   e) System.out.println(8 / a);
   f) System.out.println((int) (8 / a));

# UNIT 3 Operators and Expressions

## 3.1 Introduction

Java provides a rich set of operators to manipulate variables. We can divide all the Java operators into the following groups:

- Arithmetic Operators
- Relational Operators
- Bitwise Operators
- Logical Operators
- Assignment Operators

## 3.2 Unit Objectives

The objectives of this unit are to:

- Introduce arithmetic, relational, bitwise, logical, and assignment operators.
- Learn to use different operators to create expressions.
- Learn about the precedence of operators to result in the correct expression.
- Discover comparison and logical operators, which make logical expressions possible.

## 3.3 The Arithmetic Operators

Arithmetic operators are used in mathematical expressions in the same way that they are used in algebra. The following table lists the arithmetic operators:

Assume integer variable A holds 10 and variable B holds 20, then:

| Operator | Description | Example |
|---|---|---|
| + | Addition - Adds values on either side of the operator | A + B will give 30 |
| - | Subtraction - Subtracts right hand operand from left hand operand | A - B will give -10 |
| * | Multiplication - Multiplies values on either side of the operator | A * B will give 200 |
| / | Division - Divides left hand operand by right hand operand | B / A will give 2 |
| % | Modulus - Divides left hand operand by right hand operand and returns remainder | B % A will give 0 |
| ++ | Increment - Increases the value of operand by 1 | B++ gives 21 |
| -- | Decrement - Decreases the value of operand by 1 | B-- gives 19 |

**Example**

The following simple example program demonstrates the arithmetic operators. Write the following Java program in Test.java file, compile and run this program:

```
public class Test{

public static void main(String args[]){
int a =10;
int b =20;
int c =25;
int d =25;
System.out.println("a + b = "+(a + b));
System.out.println("a - b = "+(a - b));
System.out.println("a * b = "+(a * b));
System.out.println("b / a = "+(b / a));
System.out.println("b % a = "+(b % a));
System.out.println("c % a = "+(c % a));
System.out.println("a++   = "+(a++));
System.out.println("b--   = "+(a--));
// Check the difference in d++ and ++d
System.out.println("d++   = "+(d++));
System.out.println("++d   = "+(++d));
}
}
```

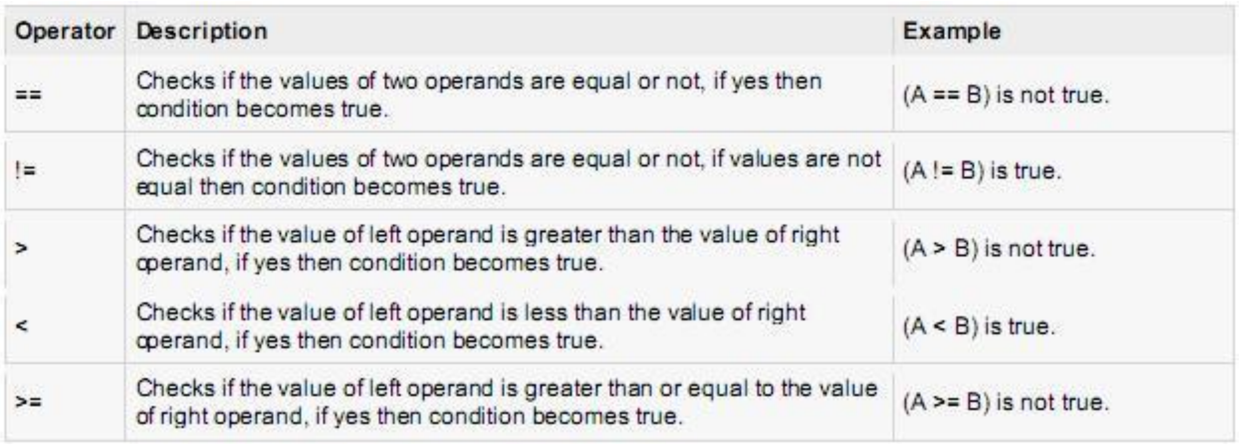
This would produce the following result:

```
a + b =30
a - b =-10
a * b =200
b / a =2
b % a =0
c % a =5
a++=10
b--=11
d++=25
++d   =27
```

## 3.4 The Relational Operators:

There are following relational operators supported by Java language:

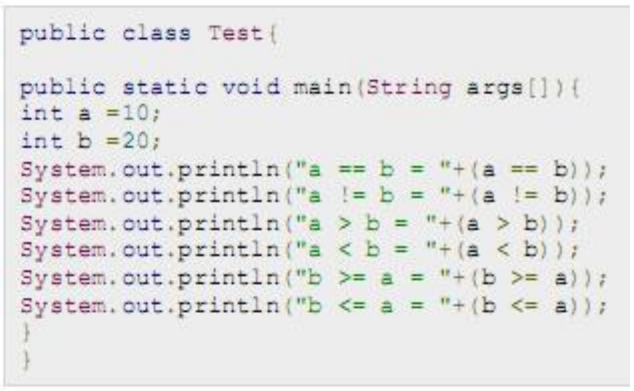
Assume variable A holds 10 and variable B holds 20, then:



| Operator | Description | Example |
|---|---|---|
| == | Checks if the values of two operands are equal or not, if yes then condition becomes true. | (A == B) is not true. |
| != | Checks if the values of two operands are equal or not, if values are not equal then condition becomes true. | (A != B) is true. |
| > | Checks if the value of left operand is greater than the value of right operand, if yes then condition becomes true. | (A > B) is not true. |
| < | Checks if the value of left operand is less than the value of right operand, if yes then condition becomes true. | (A < B) is true. |
| >= | Checks if the value of left operand is greater than or equal to the value of right operand, if yes then condition becomes true. | (A >= B) is not true. |

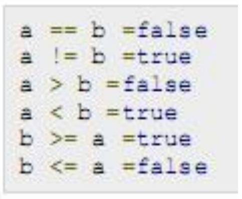
Example

The following simple example program demonstrates the relational operators. Write the following Java program in Test.java file, compile, and run this program:



```
public class Test{

public static void main(String args[]){
int a =10;
int b =20;
System.out.println("a == b = "+(a == b));
System.out.println("a != b = "+(a != b));
System.out.println("a > b = "+(a > b));
System.out.println("a < b = "+(a < b));
System.out.println("b >= a = "+(b >= a));
System.out.println("b <= a = "+(b <= a));
}
}
```

This would produce the following result:



```
a == b =false
a != b =true
a > b =false
a < b =true
b >= a =true
b <= a =false
```

## 3.5 The Bitwise Operators:

Java defines several bitwise operators, which can be applied to the integer Bitwise operator works on bits and performs bit-by-bit operation. Assume if they will be as follows:

a = 0011 1100

b = 0000 1101

a&b = 0000 1100

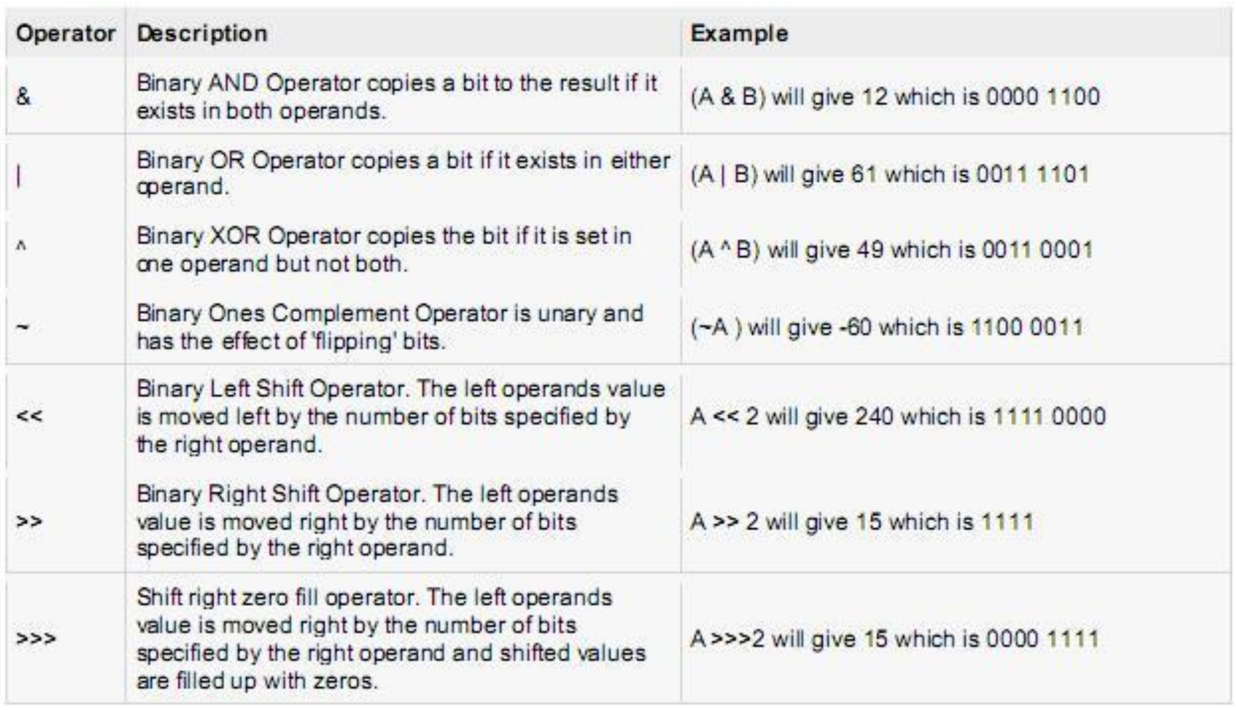
a|b = 0011 1101

a^b = 0011 0001

~a = 1100 0011

The following table lists the bitwise operators:

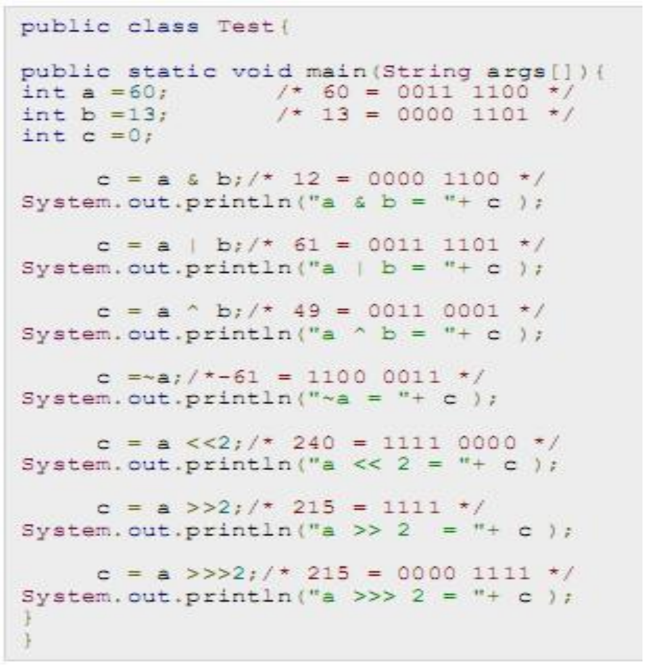
Assume integer variable A holds 60 and variable B holds 13, then:



| Operator | Description | Example |
|---|---|---|
| & | Binary AND Operator copies a bit to the result if it exists in both operands. | (A & B) will give 12 which is 0000 1100 |
| \| | Binary OR Operator copies a bit if it exists in either operand. | (A \| B) will give 61 which is 0011 1101 |
| ^ | Binary XOR Operator copies the bit if it is set in one operand but not both. | (A ^ B) will give 49 which is 0011 0001 |
| ~ | Binary Ones Complement Operator is unary and has the effect of 'flipping' bits. | (~A ) will give -60 which is 1100 0011 |
| << | Binary Left Shift Operator. The left operands value is moved left by the number of bits specified by the right operand. | A << 2 will give 240 which is 1111 0000 |
| >> | Binary Right Shift Operator. The left operands value is moved right by the number of bits specified by the right operand. | A >> 2 will give 15 which is 1111 |
| >>> | Shift right zero fill operator. The left operands value is moved right by the number of bits specified by the right operand and shifted values are filled up with zeros. | A >>>2 will give 15 which is 0000 1111 |

## Example

The following simple example program demonstrates the bitwise operators. Write the following Java program in Test.java file, compile, and run this program:



```
public class Test{

public static void main(String args[]){
int a =60;       /* 60 = 0011 1100 */
int b =13;       /* 13 = 0000 1101 */
int c =0;

      c = a & b;/* 12 = 0000 1100 */
System.out.println("a & b = "+ c );

      c = a | b;/* 61 = 0011 1101 */
System.out.println("a | b = "+ c );

      c = a ^ b;/* 49 = 0011 0001 */
System.out.println("a ^ b = "+ c );

      c =~a;/*-61 = 1100 0011 */
System.out.println("~a = "+ c );

      c = a <<2;/* 240 = 1111 0000 */
System.out.println("a << 2 = "+ c );

      c = a >>2;/* 215 = 1111 */
System.out.println("a >> 2  = "+ c );

      c = a >>>2;/* 215 = 0000 1111 */
System.out.println("a >>> 2 = "+ c );
}
}
```

This would produce the following result:

```
a & b =12
a | b =61
a ^ b =49
~a =-61
a <<2=240
a >>15
a >>>15
```

## 3.6 The Logical Operators:

The following table lists the logical operators:

Assume Boolean variables A holds true and variable B holds false, then:

| Operator | Description | Example |
|---|---|---|
| && | Called Logical AND operator. If both the operands are non-zero, then the condition becomes true. | (A && B) is false. |
| \|\| | Called Logical OR Operator. If any of the two operands are non-zero, then the condition becomes true. | (A \|\| B) is true. |
| ! | Called Logical NOT Operator. Use to reverses the logical state of its operand. If a condition is true then Logical NOT operator will make false. | !(A && B) is true. |

**Example**

The following simple example program demonstrates the logical operators. Write the following Java program in Test.java file, compile, and run this program:

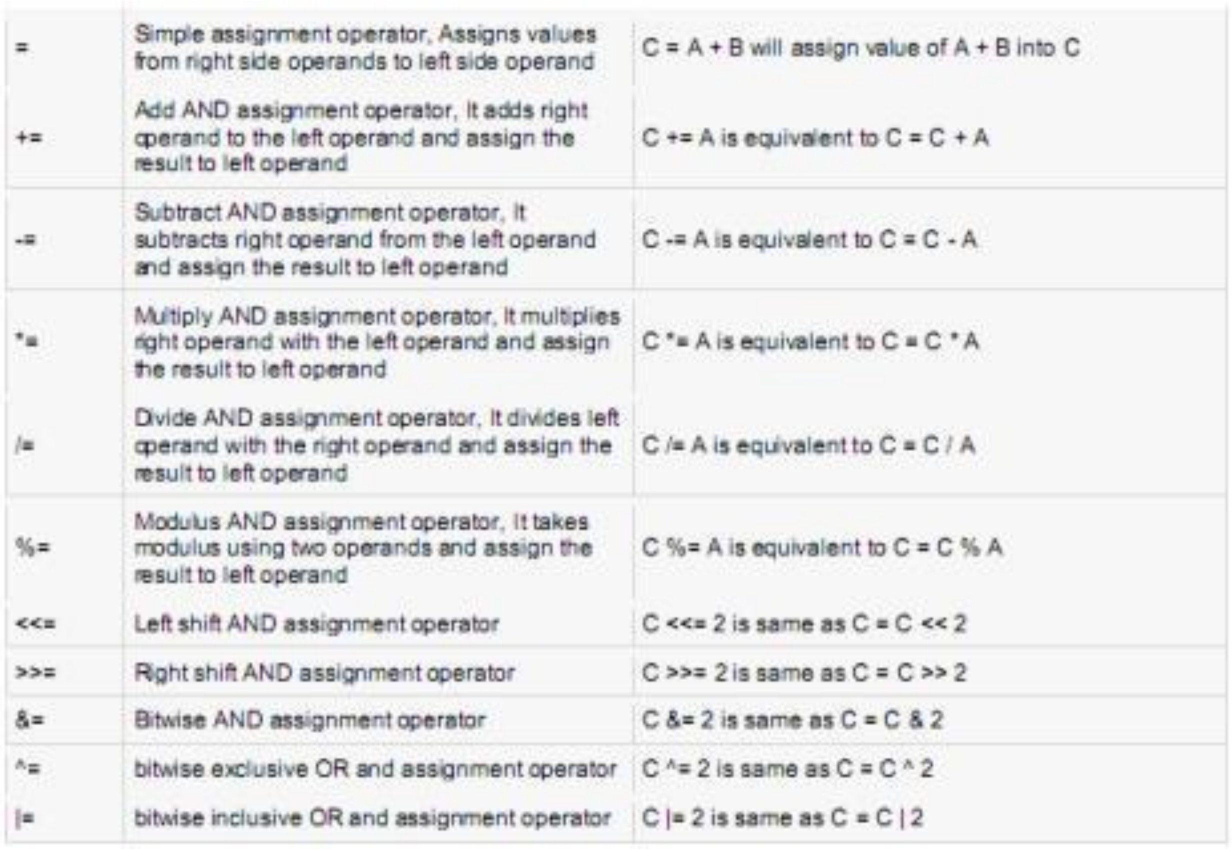
```
public class Test{

public static void main(String args[]){
boolean a =true;
boolean b =false;

System.out.println("a && b = "+(a&&b));

System.out.println("a || b = "+(a||b));

System.out.println("!(a && b) = "+!(a && b));
}
}
```

This would produce the following result:

```
a && b =false
a || b =true
!(a && b)=true
```

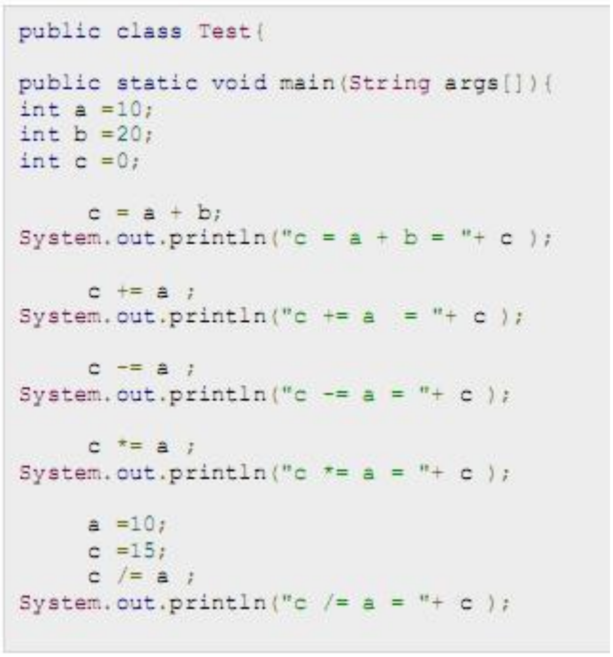
## 3.7 The Assignment Operators:

There are following assignment operators supported by Java language:

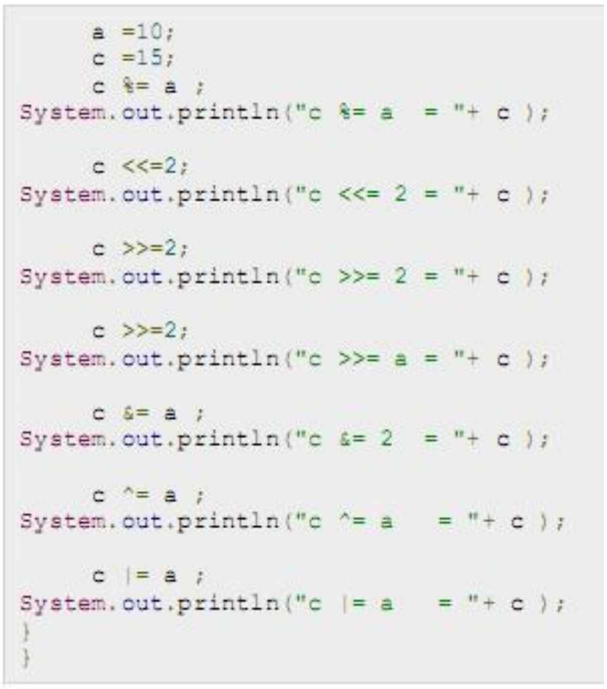


| | | |
|---|---|---|
| = | Simple assignment operator, Assigns values from right side operands to left side operand | C = A + B will assign value of A + B into C |
| += | Add AND assignment operator, It adds right operand to the left operand and assign the result to left operand | C += A is equivalent to C = C + A |
| -= | Subtract AND assignment operator, It subtracts right operand from the left operand and assign the result to left operand | C -= A is equivalent to C = C - A |
| *= | Multiply AND assignment operator, It multiplies right operand with the left operand and assign the result to left operand | C *= A is equivalent to C = C * A |
| /= | Divide AND assignment operator, It divides left operand with the right operand and assign the result to left operand | C /= A is equivalent to C = C / A |
| %= | Modulus AND assignment operator, It takes modulus using two operands and assign the result to left operand | C %= A is equivalent to C = C % A |
| <<= | Left shift AND assignment operator | C <<= 2 is same as C = C << 2 |
| >>= | Right shift AND assignment operator | C >>= 2 is same as C = C >> 2 |
| &= | Bitwise AND assignment operator | C &= 2 is same as C = C & 2 |
| ^= | bitwise exclusive OR and assignment operator | C ^= 2 is same as C = C ^ 2 |
| \|= | bitwise inclusive OR and assignment operator | C \|= 2 is same as C = C \| 2 |

The following simple example program demonstrates the assignment operators. Write the following Java program in Test.java file, compile, and run this program:

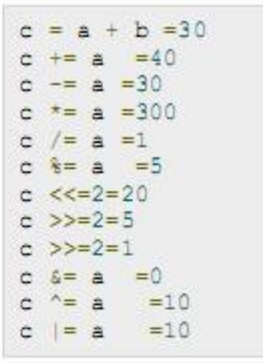


```
public class Test{

public static void main(String args[]){
int a =10;
int b =20;
int c =0;

     c = a + b;
System.out.println("c = a + b = "+ c );

     c += a ;
System.out.println("c += a  = "+ c );

     c -= a ;
System.out.println("c -= a = "+ c );

     c *= a ;
System.out.println("c *= a = "+ c );

     a =10;
     c =15;
     c /= a ;
System.out.println("c /= a = "+ c );
```



```
     a =10;
     c =15;
     c %= a ;
System.out.println("c %= a  = "+ c );

     c <<=2;
System.out.println("c <<= 2 = "+ c );

     c >>=2;
System.out.println("c >>= 2 = "+ c );

     c >>=2;
System.out.println("c >>= a = "+ c );

     c &= a ;
System.out.println("c &= 2  = "+ c );

     c ^= a ;
System.out.println("c ^= a   = "+ c );

     c |= a ;
System.out.println("c |= a   = "+ c );
}
}
```

This would produce the following result:



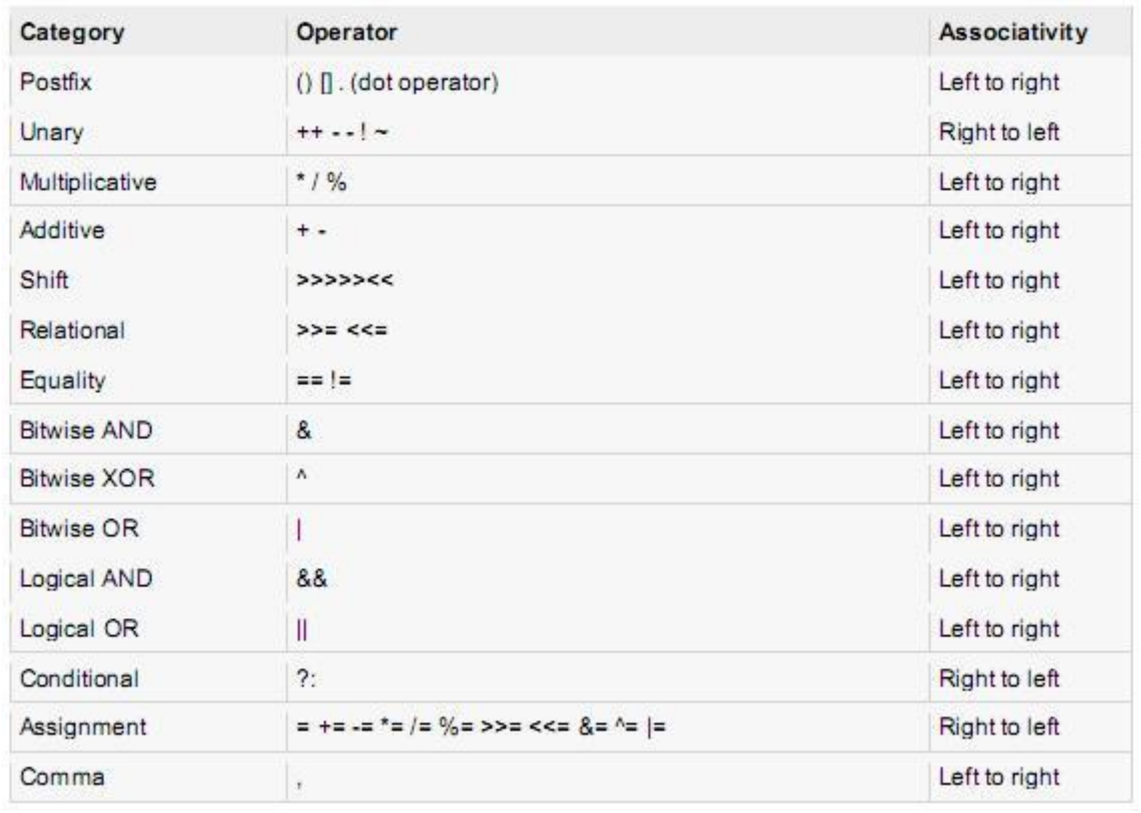
```
c = a + b =30
c += a  =40
c -= a =30
c *= a =300
c /= a =1
c %= a  =5
c <<=2=20
c >>=2=5
c >>=2=1
c &= a  =0
c ^= a   =10
c |= a   =10
```

## 3.8 Precedence of Java Operators:

Operator precedence determines the grouping of terms in an expression. This affects how an expression is evaluated. Certain operators have higher precedence than others do; for example, the multiplication operator has higher precedence than the addition operator:

For example, x = 7 + 3 * 2; here, x is assigned 13, not 20 because operator * has higher precedence than +, so it first is multiplied with 3*2 and then adds into 7.

Here, operators with the highest precedence appear at the top of the table, those with the lowest appear at the bottom. Within an expression, higher precedence operators will be evaluated first.



| Category | Operator | Associativity |
|---|---|---|
| Postfix | () [] . (dot operator) | Left to right |
| Unary | ++ - - ! ~ | Right to left |
| Multiplicative | * / % | Left to right |
| Additive | + - | Left to right |
| Shift | >>>>>><< | Left to right |
| Relational | >>= <<= | Left to right |
| Equality | == != | Left to right |
| Bitwise AND | & | Left to right |
| Bitwise XOR | ^ | Left to right |
| Bitwise OR | \| | Left to right |
| Logical AND | && | Left to right |
| Logical OR | \|\| | Left to right |
| Conditional | ?: | Right to left |
| Assignment | = += -= *= /= %= >>= <<= &= ^= \|= | Right to left |
| Comma | , | Left to right |

## 3.9 Java Type Casting

Type casting is when you assign a value of one primitive data type to another type. In Java, there are two types of casting:

**Widening Casting** (automatically) - converting a smaller type to a larger type size
byte -> short -> char -> int -> long -> float -> double
**Narrowing Casting** (manually) - converting a larger type to a smaller size type
double -> float -> long -> int -> char -> short -> byte

```
class Main {
  public static void main(String[] args) {
    // create double type variable
    double num = 10.99;
    System.out.println("The double value: " + num);

    // convert into int type and multiply by two
    int data = (int)num*2;
    System.out.println("The integer value: " + data);
  }
}
```

## 3.10 Summary

At the end of this unit, the learner/student should be able to:

- Understand the different arithmetic, relational, bitwise, logical and assignment operators.
- Identify of the precedence of operators to result in the correct expression.
- Use comparison and logical operators, which make logical expressions possible.

## 3.11 Activities

1. Find the value of the variable result after executing the following sentences:

```
int a;
int b;
int result;

a  =  4;
b = 12;
result = a + b / 3;

a = 3;
a = a + 3;
b = 2;
result = a – b;

a = 2;
b = a + 1;
a = b + 2;
result = a + b +a;
result = -result;

a = 3;
b = 11;
result = (b % a) + 1;

a = 3; b = a++;
result = 1;
result += a – b;
```

2. Develop a Java program that reads a temperature value in Celsius degrees from the keyboard and transforms it to Fahrenheit degrees. The program must print the two values in the form: X Celsius degrees are Y Fahrenheit degrees.

(Remember: $\frac{F-32}{9} = \frac{C}{5}$)

3. Develop a Java program to exchange the values of two integer variables (e.g.: if x is equal to 10 and y is equal to 5, at the end of the program, x must be equal to 5 and y equals to 10). (Initialize the variables in the code; do not read from the keyboard.)

4. Develop a Java program that, given an integer value representing a number of seconds, transform it to an expression in hours, minutes, and seconds (e.g. 3680 seconds are 1 hour, 1 minute, and 20 seconds).

5. A Physics student gets unexpected results when using the code
F = G * mass1 * mass2 / r * r;
to compute values according to the formula F = G. m1. m2 / $r^2$.
Explain the problem and correct the code.

6. Write a program that takes two int values a and b, and prints a random integer between a and b.

7. Write a Java program that assigns values to three values a, b and c, then calculate the result of a2 + (b - 12) x c.

UNIT 4

# Methods and Parameter Passing

## 4.1 Introduction

A Java method is a collection of statements that are grouped together to perform an operation. When you call the 'System.out.println' method, for example, the system actually executes several statements in order to display a message on the console.

Now you will learn how to create your own methods with or without return values, invoke a method with or without parameters, overload methods using the same names, and apply method abstraction in the program design.

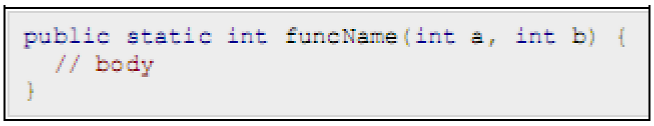
## 4.2 Unit Objectives

The objectives of this unit are to:

- Understand the importance of using method
- Differentiate between function and procedure
- Learn the structure of a method
- Learn to call a method, pass and return values from a method.

## 4.3 Creating Method

Considering the following example to explain the syntax of a method:



```
public static int funcName(int a, int b) {
   // body
}
```

Here,

- public static : modifier.
- int: return type
- funcName: function name
- a, b: formal parameters
- int a, int b: list of parameters

Methods are also known as Procedures or Functions:

- Procedures: They do not return any value.
- Functions: They return a value.

Method definition consists of a method header and a method body. The same is shown below:

```
modifier returnType nameOfMethod (Parameter List) {
 // method body
}
```

The syntax shown above includes:

- modifier: It defines the access type of the method and it is optional to use.
- returnType: Method may return a value.
- nameOfMethod: This is the method name. The method signature consists of the method name and the parameter list.
- Parameter List: The list of parameters, it is the type, order, and number of parameters of a method. These are optional, a method may contain zero parameters.
- method body: The method body defines what the method does with statements.

**Example:**

Here is the source code of the above defined method called max(). This method takes two parameters num1 and num2 and returns the maximum between the two:

```
/** the snippet returns the minimum between two numbers */
public static int minFunction(int n1, int n2) {
   int min;
   if (n1 > n2)
      min = n2;
   else
      min = n1;

   return min;
}
```

## 4.4 Method Calling

For using a method, it should be called. There are two ways in which a method is called i.e. method returns a value or returns nothing (no return value).

The process of method calling is simple. When a program invokes a method, the program control is transferred to the called method. This called method then returns control to the caller in two conditions, when:

- return statement is executed.
- reaches the method ending closing brace.

The methods returning void is considered as call to a statement. Let us consider an example:

```
System.out.println("This is tutorialspoint.com!");
```

The method returning value can be understood by the following example:

```
int result = sum(6, 9);
```

**Example:**

Following is the example to demonstrate how to define a method and how to call it:

```
public class ExampleMinNumber{

   public static void main(String[] args) {
      int a = 11;
      int b = 6;
      int c = minFunction(a, b);
      System.out.println("Minimum Value = " + c);
   }

   /** returns the minimum of two numbers */
   public static int minFunction(int n1, int n2) {
      int min;
      if (n1 > n2)

         min = n2;
      else
         min = n1;

      return min;
   }
}
```

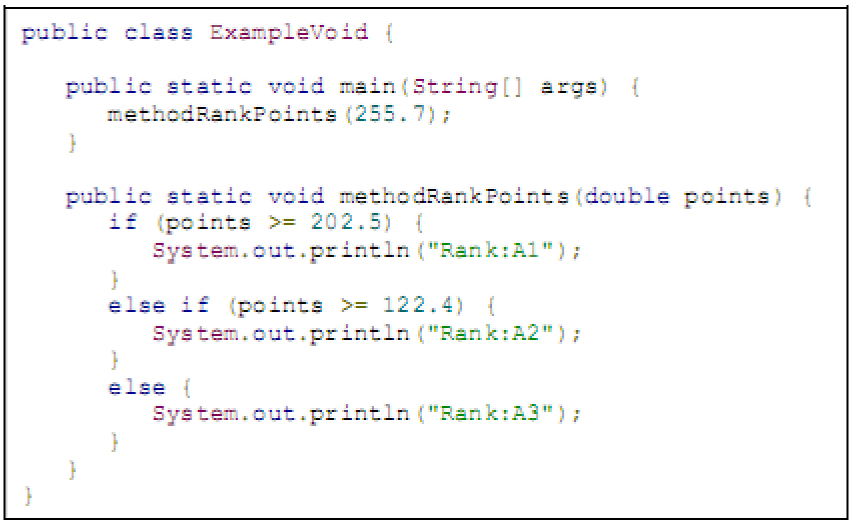
This would produce the following result:

```
Minimum value = 6
```

## 4.5 The void Keyword:

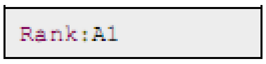
The void keyword allows us to create methods that do not return a value. Here, in the following example, we are considering a void method methodRankPoints. This method is a void method that does not return any value. Call to a void method must be a statement i.e. methodRankPoints(255.7);. It is a Java statement that ends with a semicolon as shown below.

**Example:**



```
public class ExampleVoid {

   public static void main(String[] args) {
      methodRankPoints(255.7);
   }

   public static void methodRankPoints(double points) {
      if (points >= 202.5) {
         System.out.println("Rank:A1");
      }
      else if (points >= 122.4) {
         System.out.println("Rank:A2");
      }
      else {
         System.out.println("Rank:A3");
      }
   }
}
```

This would produce the following result:



```
Rank:A1
```

## 4.6 Passing Parameters by Value:

While working under calling process, arguments are passed. These should be in the same order as their respective parameters in the method specification. Parameters can be passed by value or by reference.

Passing Parameters by Value means calling a method with a parameter. Through this, the argument value is passed to the parameter.

The other way to pass parameters is by reference, but this is not covered here.

**Example:**

The following program shows an example of passing parameter by value. The values of the arguments remain the same even after the method invocation.

```
public class swappingExample {

   public static void main(String[] args) {
      int a = 30;
      int b = 45;

      System.out.println("Before swapping, a = " +
                         a + " and b = " + b);

      // Invoke the swap method
      swapFunction(a, b);
      System.out.println("\n**Now, Before and After swapping values will be
same here**:");
      System.out.println("After swapping, a = " +
                         a + " and b is " + b);
   }

   public static void swapFunction(int a, int b) {

      System.out.println("Before swapping(Inside), a = " + a
                           + " b = " + b);
      // Swap n1 with n2
      int c = a;
      a = b;
      b = c;

      System.out.println("After swapping(Inside), a = " + a
                           + " b = " + b);
   }
}
```

This would produce the following result:

```
Before swapping, a = 30 and b = 45
Before swapping(Inside), a = 30 b = 45
After swapping(Inside), a = 45 b = 30

**Now, Before and After swapping values will be same here**:
After swapping, a = 30 and b is 45
```

## 4.7 Recursion

Recursion is calling a same function in a function until certain a condition is met. It helps in breaking big problem into smaller ones. Recursion also makes code more readable and expressive.

**Example** – a function calling itself.

```
int function(int value) {
   if(value < 1)
      return;
   function(value - 1);

   printf("%d ",value);
}
```

A recursive function can go infinite like a loop. To avoid infinite running of recursive function, there are two properties that a recursive function must have:

- Base criteria − There must be at least one base criteria or condition, such that, when this condition is met the function stops calling itself recursively.
- Progressive approach − The recursive calls should progress in such a way that each time a recursive call is made, it comes closer to the base criteria.

The fibonacci series is a series in which each number is the sum of the previous two numbers. The number at a particular position in the fibonacci series can be obtained using a recursive method.

A program that demonstrates this is given as follows:

```
public class Demo {

  public static long fib(long n) {
    if ((n == 0) || (n == 1))
      return n;
    else
      return fib(n - 1) + fib(n - 2);
  }

  public static void main(String[] args) {
    System.out.println("The 0th fibonacci number is: " + fib(0));
    System.out.println("The 7th fibonacci number is: " + fib(7));
    System.out.println("The 12th fibonacci number is: " + fib(12));
  }

}
```

Output:

```
The 0th fibonacci number is: 0
The 7th fibonacci number is: 13
The 12th fibonacci number is: 144
```

## 4.8 Summary

At the end of this unit, a student should be able to:

- understand the importance of using method
- differentiate between function and procedure
- define the structure of a method
- call a method, pass and return values from a method.
- understand recursion.

## 4.9 Activities

1. Create a program that ask the user to input (use Scanner) values to the length and breadth of a rectangle, and then call a method to calculate and display both the area and perimeter of the rectangle.

2. Create a program that asks the user to input 3 values, and then call a method to display the average of the three numbers. For example, a=10, b=20, c=30. Average is 20.

3. Create a method that takes three values, 'a', 'b' and 'c', which are the coefficients of a quadratic equation and returns the value of the delta, which is given by '$b^2$ - 4ac'.

4. Create an application for conversion between Fahrenheit and Celsius temperatures. Use a method with parameter passing.

First, the user must choose whether to enter the temperature in Celsius or Fahrenheit, chosen after the conversion is performed through a command switch.

If C is the temperature in Celsius and F is Farenheit, the conversion formulas are:

C = 5. (F-32) / 9

F = (9.C/5)+32

# UNIT 5 Selection and Iteration

## 5.1 Introduction

There are two types of decision-making statements in Java. They are 'if' and 'switch' statements.

There may be a situation when we need to execute a block of code several number of times and this is often referred to as a loop/iteration. Java has very flexible three looping mechanisms. You can use one of the following three loops:

- while Loop
- do...while Loop
- for Loop

## 5.2 Unit Objectives

The objectives of this unit are to:

- Understand if and switch statements for decision-making.
- Understand loops for iteration purpose.
- Differentiate between while, do..while and for loops.
- Apply the loop structures in different scenarios.

## 5.3 The if Statement:

An if statement consists of a Boolean expression followed by one or more statements.

Syntax:

```
if(Boolean_expression)
{
//Statements will execute if the Boolean expression is true
}
```

If the Boolean expression evaluates to true, then the block of code inside the if statement will be executed. If not, the first set of code after the end of the if statement (after the closing curly brace) will be executed.

Example:

```
public class Test{

public static void main(String args[]){
int x =10;

if( x <20){
System.out.print("This is if statement");
}
}
}
```

This would produce the following result:

```
Thisisif statement
```

## 5.4 The if...else Statement:

An if statement can be followed by an optional else statement, which executes when the Boolean expression is false.

Syntax:

The syntax of an if...else is:

```
if(Boolean_expression){
//Executes when the Boolean expression is true
}else{
//Executes when the Boolean expression is false
}
```

Example:

```
public class Test{

public static void main(String args[]){
int x =30;

if(x <20){
System.out.print("This is if statement");
}else{
System.out.print("This is else statement");
}
}
}
```

This would produce the following result:

```
Thisiselse statement
```

## 5.5 The if...else if...else Statement:

An if statement can be followed by an optional else if...else statement, which is very useful to test various conditions using single if...else if statement.

When using if, else if, else statements there are few points to keep in mind.

- An if can have zero or one else's and it must come after any else if's.
- An if can have zero to many else if's and they must come before the else.
- Once an else if succeeds, none of the remaining else if's or else's will be tested.

Syntax:

The syntax of an if...else is:

```
if(Boolean_expression1){
//Executes when the Boolean expression 1 is true
}elseif(Boolean_expression2){
//Executes when the Boolean expression 2 is true
}elseif(Boolean_expression3){
//Executes when the Boolean expression 3 is true
}else{
//Executes when the none of the above condition is true.
}
```

Example:

```
public class Test{

public static void main(String args[]){
int x =30;

if( x ==10){
System.out.print("Value of X is 10");
}elseif( x ==20){
System.out.print("Value of X is 20");
}elseif( x ==30){
System.out.print("Value of X is 30");
}else{
System.out.print("This is else statement");
}
}
}
```

This would produce the following result:

```
Value of X is30
```

## 5.6 Nested if...else Statement:

It is always legal to nest if-else statements which means you can use one if or else if statement inside another if or else if statement.

Syntax:

The syntax for a nested if...else is as follows:

```
if(Boolean_expression1){
//Executes when the Boolean expression 1 is true
if(Boolean_expression2){
//Executes when the Boolean expression 2 is true
}
}
```

*You can nest else if...else in the similar way as we have nested if statement.*

Example:

```
public class Test{

public static void main(String args[]){
int x =30;
int y =10;

if( x ==30){
if( y ==10){
System.out.print("X = 30 and Y = 10");
}
}
}
```

This would produce the following result:

```
X =30and Y =10
```

## 5.7 The switch Statement:

A switch statement allows a variable to be tested for equality against a list of values. Each value is called a case, and the variable being switched on is checked for each case.

Syntax:

```
switch(expression){
case value :
//Statements
break;//optional
case value :
//Statements
break;//optional
//You can have any number of case statements.
default://Optional
//Statements
}
```

Example:

```
public class Test{

public static void main(String args[]){
char grade = args[0].charAt(0);

switch(grade)
{
case'A':
System.out.println("Excellent!");
break;
case'B':
case'C':
System.out.println("Well done");
break;
case'D':
System.out.println("You passed");
case'F':
System.out.println("Better try again");
break;
default:
System.out.println("Invalid grade");
}
System.out.println("Your grade is "+ grade);
}
}
```

This would produce the following result:

```
$ java Test a
Invalid grade
Your grade is a a
$ java Test A
Excellent!
Your grade is a A
$ java Test C
Welldone
Your grade is a C
$
```

## 5.8 The while Loop:

A while loop is a control structure that allows you to repeat a task a certain number of times.

Syntax:

The syntax of a while loop is:

```
while(Boolean_expression)
{
//Statements
}
```

When executing, if the boolean_expression result is true, then the actions inside the loop will be executed. This will continue as long as the expression result is true.

Here, key point of the while loop is that the loop might not ever run. When the expression is tested and the result is false, the loop body will be skipped and the first statement after the while loop will be executed.

**Example:**

```
public class Test{

public static void main(String args[]){
int x =10;

while( x <20){
System.out.print("value of x : "+ x );

      x++;
System.out.print("\n");
}
}
}
```

This would produce the following result:

```
value of x :10
value of x :11
value of x :12
value of x :13
value of x :14
value of x :15
value of x :16
value of x :17
value of x :18
value of x :19
```

## 5.9 The do...while Loop:

A do...while loop is similar to a while loop, except that a do...while loop is guaranteed to execute at least one time.

Syntax:

The syntax of a do...while loop is:

```
do
{
//Statements
}while(Boolean_expression);
```

Notice that the Boolean expression appears at the end of the loop, so the statements in the loop execute once before the Boolean is tested.

If the Boolean expression is true, the flow of control jumps back up to do, and the statements in the loop execute again. This process repeats until the Boolean expression is false.

**Example:**

```
public class Test{

public static void main(String args[]){
int x =10;

do{
System.out.print("value of x : "+ x );
      x++;
System.out.print("\n");
}while( x <20);
}
}
```

This would produce the following result:

```
value of x :10
value of x :11
value of x :12
value of x :13
value of x :14
value of x :15
value of x :16
value of x :17
value of x :18
value of x :19
```

### 5.10 The for Loop:

A for loop is a repetition control structure that allows you to efficiently write a loop that needs to execute a specific number of times.

A for loop is useful when you know how many times a task is to be repeated.

Syntax:

The syntax of a for loop is:

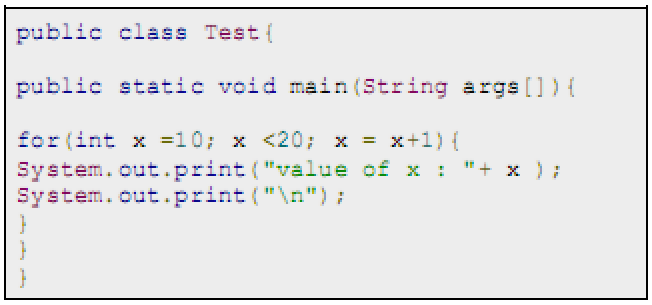
```
for(initialization;Boolean_expression; update)
{
//Statements
}
```

Here is the flow of control in a for loop:

- The initialization step is executed first, and only once. This step allows you to declare and initialize any loop control variables. You are not required to put a statement here, as long as a semicolon appears.

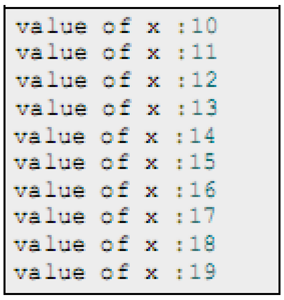
- Next, the Boolean expression is evaluated. If it is true, the body of the loop is executed. If it is false, the body of the loop does not execute and flow of control jumps to the next statement past the for loop.
- After the body of the for loop executes, the flow of control jumps back up to the update statement. This statement allows you to update any loop control variables. This statement can be left blank, as long as a semicolon appears after the Boolean expression.
- The Boolean expression is now evaluated again. If it is true, the loop executes and the process repeats itself (body of loop, then update step, then Boolean expression). After the Boolean expression is false, the for loop terminates.

Example:



```
public class Test{

public static void main(String args[]){

for(int x =10; x <20; x = x+1){
System.out.print("value of x : "+ x );
System.out.print("\n");
}
}
}
```

This would produce the following result:



```
value of x :10
value of x :11
value of x :12
value of x :13
value of x :14
value of x :15
value of x :16
value of x :17
value of x :18
value of x :19
```

## 5.11 Summary

At the end of this unit, a student should be able to:

- understand if and switch statements for decision making.
- understand loops for iteration purpose.
- differentiate between while, do..while and for loops.
- apply the loop structures in different scenarios.

## 5.12 Activities

1. What is the main difference between a while loop and a do..while loop?

2. Write a for loop that will print out all the multiples of 3 from 3 to 36, that is: 3 6 9 12 15 18 21 24 27 30 33 36.

3. Show the exact output that would be produced by the following main() routine:

```
public static void main(String[] args) {
        int N;
        N=1;
        while (N <= 32) {
                N=2*N;
                System.out.println(N);
        }
}
```

4. Show the exact output produced by the following main() routine:

```
public static void main(String[] args) {
        int x,y;
        x = 5;
        y = 1;
        while (x > 0) {
                x = x - 1;
                y = y * x;
                System.out.println(y);
        }
}
```

5. Create a program called Comparison that obtains two integers from the user and uses three if-structures to output if the first is larger than the second, the first is smaller than the second, or both numbers are equal.

6. Create a program called CerealCompare using an if-else structure that obtains the price and number of grams in a box for two boxes of cereal. The program should then output which box costs less per gram.

7. Create a program called DetermineAge that asks for the last two digits of the year in which a Peel student was born (for example, 89 for 1989) and the last two digits of the current year (for example, 06 for 2006). It should then calculate the student’s age. The program must also work for those born after the year 2000.

8. Write a method to calculate the VAT amount for a product. A product may be zero-rated or taxable at 15%. A product may also be entitled to a discount percentage. Then create a main method to test the above method.

9. Write a method to calculate the commission earned by a sales representative. If he sells less than 10 units, no commission is given. If he sells 10 to 50 units, he receives 2% of the sales amount. If he sells over 50 units, he earns 2% on the first 50 units, and 5% on the additional units. The price of an item is constant at 125.00. Then create a main method to test the above method.

10. Mauritian population reaches 1.4 million in 2014 and was growing at the rate of 1% each year. Assuming the same rate of growth, when will the population reach 2.5 million?

# UNIT 6 Arrays and Collections

## 6.1 Introduction

Java provides a data structure, the array, which stores a fixed-size sequential collection of elements of the same type. An array is used to store a collection of data, but it is often more useful to think of an array as a collection of variables of the same type.

Instead of declaring individual variables, such as number0, number1, ..., and number99, you declare one array variable such as numbers and use numbers[0], numbers[1], and ..., numbers[99] to represent individual variables.

This unit introduces how to declare array variables, create arrays, and process arrays using indexed variables.

## 6.2 Unit Objectives

The objectives of this unit are to:

- What are arrays and collections?
- To use arrays to store data in and retrieve data from lists and tables of values.
- To declare arrays, initialize arrays and refer to individual elements of arrays.
- To iterate through arrays with the enhanced for statement.
- To pass arrays to methods.
- To declare and manipulate multidimensional arrays.
- To use variable-length argument lists.

## 6.3 Declaring Array Variables:

To use an array in a program, you must declare a variable to reference the array, and you must specify the type of array the variable can reference. Here is the syntax for declaring an array variable:

```
dataType[] arrayRefVar;// preferred way.

or

dataType arrayRefVar[];//  works but not preferred way.
```

**Example:**

The following code snippets are examples of this syntax:

```
double[] myList;// preferred way.

or

double myList[];//  works but not preferred way.
```

### 6.4 Creating Arrays:

You can create an array by using the new operator with the following syntax:

```
arrayRefVar =new dataType[arraySize];
```

The above statement does two things:

- It creates an array using new dataType[arraySize];
- It assigns the reference of the newly created array to the variable arrayRefVar.

Declaring an array variable, creating an array, and assigning the reference of the array to the variable can be combined in one statement, as shown below:

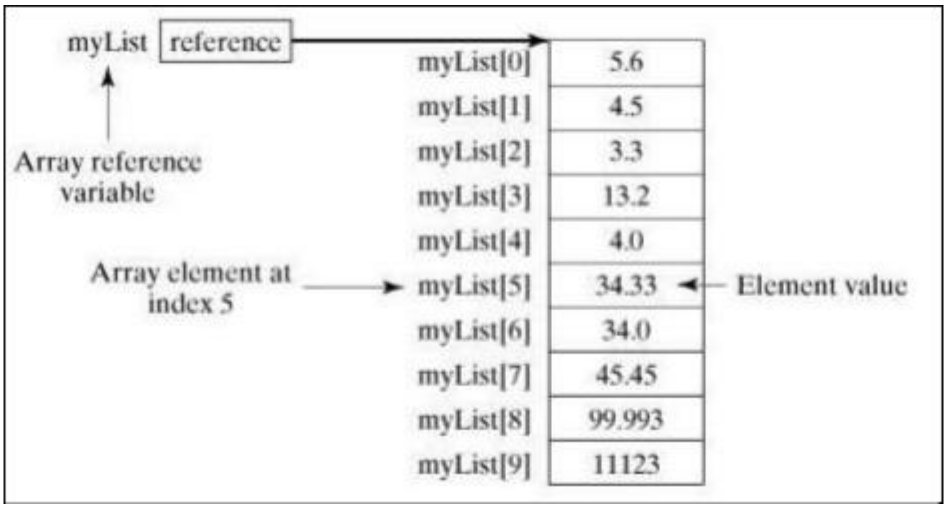
```
dataType[] arrayRefVar =new dataType[arraySize];
```

**Example:**

Following statement declares an array variable, myList, creates an array of 10 elements of double type and assigns its reference to myList:

```
double[] myList =new double[10];
```

Following picture represents array myList. Here, myList holds ten double values and the indices are from 0 to 9.

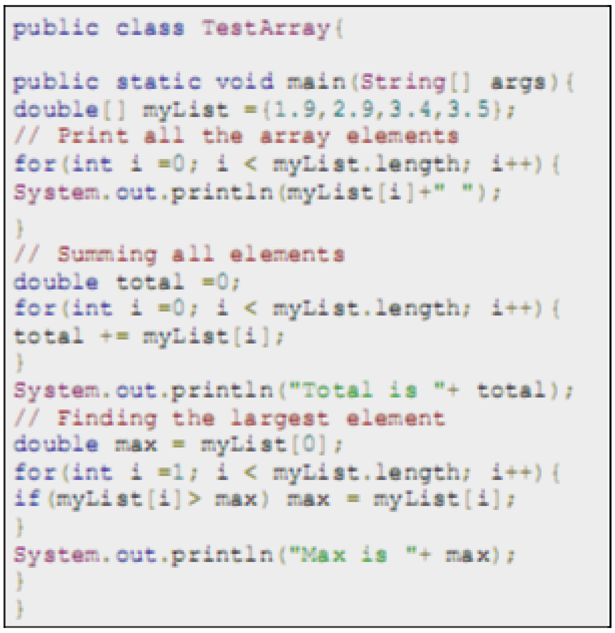


## 6.5 Processing Arrays:

When processing array elements, we often use the for loop because all of the elements in an array are of the same type and the size of the array is known.

**Example:**

Here is a complete example of showing how to create, initialize and process arrays:



```
public class TestArray{

public static void main(String[] args){
double[] myList ={1.9,2.9,3.4,3.5};
// Print all the array elements
for(int i =0; i < myList.length; i++){
System.out.println(myList[i]+" ");
}
// Summing all elements
double total =0;
for(int i =0; i < myList.length; i++){
total += myList[i];
}
System.out.println("Total is "+ total);
// Finding the largest element
double max = myList[0];
for(int i =1; i < myList.length; i++){
if(myList[i]> max) max = myList[i];
}
System.out.println("Max is "+ max);
}
}
```

## 6.6 Collections

Collections are Java classes developed to efficiently store multiple elements of a certain type. Unlike Java arrays, collections can store any number of elements. The simplest collection is ArrayList, which you can treat as a resizable array. The following line of code creates an (initially empty) ArrayList of objects of classPerson:

*ArrayList<Person> friends = new ArrayList<Person>();*

Note that the type of the collection includes the element type in angle brackets. This "tunes" the collection to work with the specific element type, so that, for example, the function get() of friends will return object of type Person. The ArrayList<Person> offers the following API:

- **int size()** – returns the number of elements in the list
- **boolean isEmpty()** – tests if this list has no elements

- **Person get( int index )** – returns the element at the specified position in this list
- **boolean add( Person p )** – appends the specified element to the end of this list
- **Person remove( int index )** – removes the element at the specified position in this list
- **boolean contains( Person p )** – returns true if this list contains a given element
- **void clear()** – removes all of the elements from this list

The following code fragment tests if the friends list contains the person victor and, if, victor is not in there, adds him to the list:

```
if( ! friends.contains( victor ) )
   friends.add( victor );
```

All collection types support iteration over elements. The simplest construct for iteration is a "for" loop. Suppose the class Person has a field income. The loop below prints all friends with income greater than 100000 to the model log:

```
for( Person p : friends ) {
   if( p.income > 100000 )
      traceln( p );
}
```

Another popular collection type is Linked List.

Linked lists are used to model stack or queue structures, i.e. sequential storages where elements are primarily added and removed from one or both ends.

Consider a model of a distributor that maintains a backlog of orders from retailers. Suppose there is a class Order with the field amount. The backlog (essentially a FIFO queue) can be modeled as:

```
LinkedList<Order> backlog = new LinkedList<Order>();
```

LinkedList supports functions common to all collections (like size() or isEmpty()) and also offers a specific API:

- **Order getFirst()** – returns the first element in this list
- **Order getLast()** – returns the last element in this list
- **addFirst( Order o )** – inserts the given element at the beginning of this list
- **addLast( Order o )** – appends the given element to the end of this list
- **Order removeFirst()** – removes and returns the first element from this list

- **Order removeLast()** – removes and returns the last element from this list

When a new order is received by the distributor, it is placed at the end of the backlog:

```
backlog.addLast( order );
```

Each time the inventory is replenished, the distributor tries to ship the orders starting from the head of the backlog. If the amount in an order is bigger than the remaining inventory, the order processing stops. The order processing can look like this:

```
while( ! backlog.isEmpty() ) { //repeat the code below while the backlog
is not empty
   Order order = backlog.getFirst(); //pick the first order in the backlog
   if( order.amount <= inventory ) { //if enough inventory to satisfy this
      order ship( order ); //ship
      inventory -= order.amount; //decrease available inventory
      backlog.removeFirst(); //remove the order from the backlog
   } else { //not enough inventory to ship
      break; //stop order backlog processing }
}
```

## 6.7 Features of ArrayList against LinkedList

### ArrayList

• ArrayList internally uses dynamic array to store the elements. When creating an Array, the size must be specified when it is created, hence an Array is not dynamic.
• Manipulation with ArrayList is slow because it internally uses array. If any element is removed from the array, all the bits are shifted in memory.
• ArrayList class can act as a list only because it implements List only.
• ArrayList is better for storing and accessing data.

### LinkedList

• LinkedList internally uses doubly linked list to store the elements.
• Manipulation with LinkedList is faster than ArrayList because it uses doubly linked list, so no bit shifting is required in memory.
• LinkedList class can act as a list, stack and queue because it implements List and Deque interfaces.
• For a stack, use addLast() and removeLast();

- For a queue, use addLast() and removeFirst();
- LinkedList is better for manipulating data.

## 6.8 Summary

• An array is a group of variables (called elements or components) containing values that all have the same type. Arrays are objects, so they are considered reference types.

• A program refers to any one of an array's elements with an array-access expression that includes the name of the array followed by the index of the particular element in square brackets [].

• The first element in every array has index zero and is sometimes called the zeroth element.

• An index must be a nonnegative integer. A program can use an expression as an index.

• An array object knows its own length and stores this information in a length instance variable.

• To create an array object, specify the array's element type and the number of elements as part of an array-creation expression that uses keyword new.

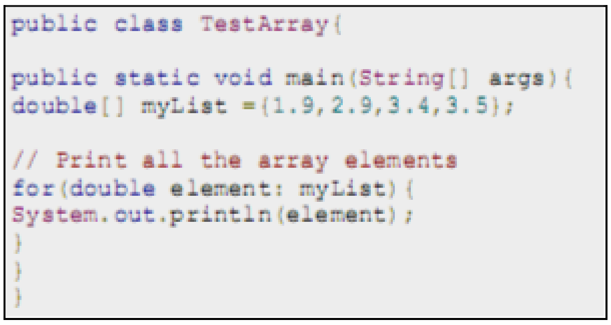
• When an array is created, each element receives a default value—zero for numeric primitive-type elements, false for Boolean elements and null for references.

• In an array declaration, the type and the square brackets can be combined at the beginning of the declaration to indicate that all the identifiers in the declaration are array variables.

• Every element of a primitive-type array contains a variable of the array's declared type. Every element of a reference-type array is a reference to an object of the array's declared type.

## 6.4 Activities

1. Determine the output of the following code:



```
public class TestArray{

public static void main(String[] args){
double[] myList ={1.9,2.9,3.4,3.5};

// Print all the array elements
for(double element: myList){
System.out.println(element);
}
}
}
```

2. Some Java programmers use int a[] instead of int[] a to declare arrays. What is the difference?

3. Why do array indices start at 0 instead of 1?

4. What happens if I use a negative number to index an array?

5. If a[] is an array, why does System.out.println(a) print out a hexadecimal integer, like @f62373, instead of the elements of the array?

6. Write a program that declares and initializes an array a[] of size 1000 and accesses a[1000]. Does your program compile? What happens when you run it?

7. Describe and explain what happens when you try to compile a program HugeArray.java with the following statement:

```
int N = 1000;
int[] a = new int[N*N*N*N];
```

8. Write a code fragment that reverses the order of a one-dimensional array a[] of double values. Do not create another array to hold the result. **Hint**: Use the code in the text for exchanging two elements.

9. What is wrong with the following code fragment?

```
int[] a;
for (int i = 0; i < 10; i++)
   a[i] = i * i;
```

10. What does the following code fragment print?

```
int [] a = new int[10];
for (int i = 0; i < 10; i++)
   a[i] = 9 - i;
for (int i = 0; i < 10; i++)
   a[i] = a[a[i]];
for (int i = 0; i < 10; i++)
   System.out.println(a[i]);
```

11. What values does the following code put in the array a[]?

```
int N = 10;
int[] a = new int[N];
a[0] = 0;
a[1] = 1;
for (int i = 2; i < N; i++)
   a[i] = a[i-1] + a[i-2];
```

12. What does the following code fragment print?

```
int[] a = { 1, 2, 3 };
int[] b = { 1, 2, 3 };
System.out.println(a == b);
```

13. Generate six random numbers between 1 and 40, and keep them in one array. Sum the six numbers of the array.

# UNIT 7 GUI

## 7.1 Introduction

- Java provides many GUI components within its included libraries to create desktop applications.
- The main ones are AWT (Abstract Widowing Toolkit), Swing and FX components.
- These libraries need to be imported whenever their components are being used, using the following syntax:
- Fortunately, modern IDEs like Eclipse & NetBeans make it easy to import these whenever required.

## 7.2 Unit Objectives

The objectives of this unit are to:

- Understand graphical components
- Create a GUI application
- Add events to the components

## 7.3 Basic JFrame Class

The javax.swing.JFrame class is a type of container that inherits the java.awt.Frame class. JFrame works like the main window where components like labels, buttons, textfields are added to create a GUI.

```
import javax.swing.*;
public class TestJFrame
{
  public static void main(String[] args)
  {
    JFrame jf1 = new JFrame("Test Frame");
    jf1.setSize(500,400);
    jf1.setVisible(true);
    System.out.println("GUI Running.");
  }
}
```

## 7.4 Extending JFrame to create MyFrame

```
import javax.swing.*;
public class MyFrame extends JFrame
{
        public MyFrame(String s)
        {
                super(s);
                setSize(500, 500);
                setLocation(200, 200);
                setVisible(true);
        }
        public static void main(String[] args)
        {
                new MyFrame("My Frame 01");
        }
}
```

## 7.5 Adding a Button to JFrame

```
import javax.swing.*;
public class ButtonFrame extends JFrame
{
        private JButton btn1 = new JButton("My Button!");
        public ButtonFrame(String s)
        {
                super(s);
                setSize(400, 300);
                setLocation(100, 200);
                add( btn1 );
                setVisible(true);
        }
        public static void main(String[] args)
        {
                new ButtonFrame("Frame with a Button.");
                System.out.println("GUI Running.");
        }
}
```

## 7.6 Layout Manager

By default, any item(e.g. a JButton) placed on a JFrame, will be set to occupy the maximum possible area.

If two items are placed on the JFrame, both will try to occupy the maximum possible area, such that, the first item will cover the whole JFrame, and the second will cover the first.

In order to position more than one item on a JFrame, we need to make use of the Layout Manager, together with some preset Layouts in Java.

It is possible to set the Layout Manager to null, thus enabling us to position any of our items using a pixel-based coordinate system.

Each item will however need to be positioned manually.

This is however discouraged, because the items are fixed in place, even when the JFrame is resized or maximised. (It is possible to prevent the user from resizing the JFrame).

```
import javax.swing.*;
public class TestNullLayout extends JFrame
{
        private JButton btn1 = new JButton("Button 1");
        private JButton btn2 = new JButton("Button 2");
        public TestNullLayout(String s)
        {
                super(s);
                setSize(400, 300);
                setLocation(100, 200);
                setLayout(null);
                btn1.setBounds(50,50,150,100);
                btn2.setBounds(200,150,150,100);
                add( btn1 );
                add( btn2 );
                setVisible(true);
         }
        public static void main(String[] args)
        {
                new TestNullLayout("2 Buttons Null Layout");
        }
}
```

## 7.7 Event Handling

GUI elements form part of event-programming, i.e. they respond to events.

In Java, an event is implemented as a class.
To use events, we need to use the import statement:

```
        import java.awt.event.*;
```

## 7.8 Action Listener

In Java, events are registered through the use of Listeners.
ActionListener is used for actions, such as button clicks.
Other Listeners also exist, for example Window Listener, Mouse Listener.

Listener is a special type of Class, called an Interface, and is linked to our application by using the implements keyword.

An actionPerformed() method is then needed to run the statements which need to be executed for all events detected

```
private JButton btn = new JButton("Click Me!");
private int num = 0;
public TestInteractiveButton(String s)
{
        super(s);
        setSize(400, 300);
        setLocation(100, 200);
        setLayout(new FlowLayout(FlowLayout.CENTER));
        btn.addActionListener(this);
        add(btn);
        setVisible(true);
}
public void actionPerformed(ActionEvent e)
{
        if(e.getSource().equals(btn))
        {
                num++;
                System.out.println("You clicked me " + num + " times.");
        }
}
```

### 7.9 JOptionPane Class

A JOptionPane object is used whenever a dialog box is required. There are four main types of dialog boxes: Confirm, Input, Message and Option, which are called by using their corresponding method listed below:

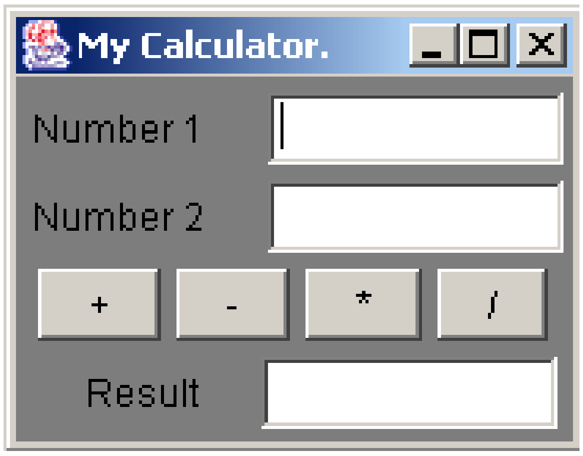
| Method Name | Description |
|---|---|
| showConfirmDialog() | Asks a confirming question, like yes/no/cancel |
| showInputDialog() | Prompt for some input |
| showMessageDialog() | Tell the user about something that has happened |
| showOptionDialog() | The Grand Unification of the above three |

Example:

```
import javax.swing.*;

public class OptionPaneExample {
  JFrame f;
  OptionPaneExample(){
     f=new JFrame();
     JOptionPane.showMessageDialog(f,"Hello.");
  }

  public static void main(String[] args) {
     new OptionPaneExample();
  }
}
```

## 7.10 Activities

1. Create a small calculator like the one below and implements its addition, subtraction, multiplication and division functionality.



# UNIT 8 Database connectivity and Simple I/O

## 8.1 Introduction

There are six steps involved in building a Java/Database application:

- **Import the packages:** Requires that you include the packages containing the JDBC classes needed for database programming. Most often, using *import java.sql.** will suffice.

```
//STEP 1. Import required packages
import java.sql.*;
```

- **Register the JDBC driver**: Requires that you initialize a driver so you can open a communications channel with the database.

```
//STEP 2: Register JDBC driver
Class.forName("com.mysql.jdbc.Driver");
```

- **Open a connection:** Requires using the *DriverManager.getConnection()* method to create a Connection object, which represents a physical connection with the database.

```
//STEP 3: Open a connection
//  Database credentials
static final String USER = "username";
static final String PASS = "password";
System.out.println("Connecting to database...");
conn = DriverManager.getConnection(DB_URL,USER,PASS);
```

- **Execute a query:** Requires using an object of type Statement for building and submitting an SQL statement to the database.

```
//STEP 4: Execute a query
System.out.println("Creating statement...");
stmt = conn.createStatement();
String sql;
sql = "SELECT id, first, last, age FROM Employees";
ResultSet rs = stmt.executeQuery(sql);
```

- **Extract data from result set:** Requires that you use the appropriate *ResultSet.getXXX()* method to retrieve the data from the result set.

```
//STEP 5: Extract data from result set
while(rs.next()){
    //Retrieve by column name
    int id  = rs.getInt("id");
    int age = rs.getInt("age");
    String first = rs.getString("first");
    String last = rs.getString("last");
```

```
    //Display values
    System.out.print("ID: " + id);
    System.out.print(", Age: " + age);
    System.out.print(", First: " + first);
    System.out.println(", Last: " + last);
}
```

- **Clean up the environment:** Requires explicitly closing all database resources versus relying on the JVM's garbage collection.

```
//STEP 6: Clean-up environment
rs.close();
stmt.close();
conn.close();
```

## 8.1 Download requirements

- Install XAMPP to use mySQL database at https://www.apachefriends.org/index.html.
- Download the MySQL Java connector at https://www.mysql.com/products/connector/
- If you are using Eclipse, go to Project - Properties - Java build path - Libraries - Add external jars to add the MySQL Java connector to your project.

## 8.2 Connecting to database

The following codes will connect the Java Application to a database created in MySQL using XAMPP.

```
// sample query
query='Select * from users';

Connection con = null;
Statement st = null;
try{

    DriverManager.registerDriver(new com.mysql.jdbc.Driver ());
    con = DriverManager.getConnection ("jdbc:mysql://localhost/test_db","root","");
    st = con.createStatement();

    st.executeUpdate(query);
    System.out.println("Query Executed");
   }
catch(Exception ex)
{
    System.out.println(ex.getMessage());
}
```

## 8.3 Select records from a table

To be to retrieve data from a table in a database, the SQL SELECT command has to be used. The syntax is as follows:

```
SELECT column_name,column_name
FROM table_name
WHERE criteria;
```

The outcome of the SQL SELECT command will be a result set (virtual table). If several records are in the result set, then the records should be accessed using a loop.

```
//STEP 4: Execute a query
      System.out.println("Creating statement...");
      stmt = conn.createStatement();

      String sql = "SELECT id, first, last, age FROM Account";
      ResultSet rs = stmt.executeQuery(sql); //STEP 5: Extract
      data from result set
      while(rs.next()){
         //Retrieve by column name
         int id  = rs.getInt("id");
         int age = rs.getInt("age");
         String first = rs.getString("first");
         String last = rs.getString("last");

         //Display values
         System.out.print("ID: " + id);
         System.out.print(", Age: " + age);
         System.out.print(", First: " + first);
         System.out.println(", Last: " + last);
      }
      rs.close();
```

## 8.4 Add records to a table

To be able to add records to a table in a database, the SQL INSERT command has to be used. The syntax is as follows:

```
INSERT INTO table_name (column1,column2,column3,...)
VALUES (value1,value2,value3,...);
```

The outcome of the SQL INSERT command will be TRUE if all records added, else it will generate an error message.

```
//STEP 4: Execute a query
      System.out.println("Inserting records into the table...");
      stmt = conn.createStatement();

      String sql = "INSERT INTO Registration " +
```

```
                   "VALUES (100, 'Zara', 'Ali', 18)";
      stmt.executeUpdate(sql);
      sql = "INSERT INTO Registration " +
                   "VALUES (101, 'Mahnaz', 'Fatma', 25)";
      stmt.executeUpdate(sql);
      sql = "INSERT INTO Registration " +
                   "VALUES (102, 'Zaid', 'Khan', 30)";
      stmt.executeUpdate(sql);
      sql = "INSERT INTO Registration " +
                   "VALUES(103, 'Sumit', 'Mittal', 28)";
      stmt.executeUpdate(sql);
      System.out.println("Inserted records into the table...");
```

## 8.5 Delete records from a table

To be able to delete records from a table in a database, the SQL DELETE command has to be used. The syntax is as follows:

```
DELETE FROM table_name
WHERE some_column=some_value;
```

The outcome of the SQL DELETE command will be TRUE if all records deleted, else it will generate an error message.

```
//STEP 4: Execute a query
      System.out.println("Creating statement...");
      stmt = conn.createStatement();
      String sql = "DELETE FROM Registration " +
                   "WHERE id = 101";
      stmt.executeUpdate(sql);

      3. Now you can extract all the records
      4. to see the remaining records
      sql = "SELECT id, first, last, age FROM Registration";
      ResultSet rs = stmt.executeQuery(sql);

      while(rs.next()){
         //Retrieve by column name
         int id  = rs.getInt("id");
         int age = rs.getInt("age");
         String first = rs.getString("first");
         String last = rs.getString("last");

         //Display values
         System.out.print("ID: " + id);
         System.out.print(", Age: " + age);
         System.out.print(", First: " + first);
         System.out.println(", Last: " + last);
      }
      rs.close();
```

## 8.6 Update records from a table

To be able to update records from a table in a database, the SQL UPDATE command has to be used. The Syntax is as follows:

```
UPDATE table_name
SET column1=value1,column2=value2,...
WHERE some_column=some_value;
```

The outcome of the SQL UPDATE command will be TRUE if all records updated, else it will generate an error message.

```
//STEP 4: Execute a query
      System.out.println("Creating statement...");
      stmt = conn.createStatement();
      String sql = "UPDATE Registration " +
                   "SET age = 30 WHERE id in (100, 101)";
      stmt.executeUpdate(sql);

        Now you can extract all the records
        to see the updated records
      sql = "SELECT id, first, last, age FROM Registration";
      ResultSet rs = stmt.executeQuery(sql);

      while(rs.next()){
         //Retrieve by column name
         int id  = rs.getInt("id");
         int age = rs.getInt("age");
         String first = rs.getString("first");
         String last = rs.getString("last");

         //Display values
         System.out.print("ID: " + id);
         System.out.print(", Age: " + age);
         System.out.print(", First: " + first);
         System.out.println(", Last: " + last);
      }
      rs.close();
```

## 8.7 Simple I/O

The java.io package contains nearly every class you might ever need to perform input and output (I/O) in Java. All these streams represent an input source and an output destination. The stream in the java.io package supports many data such as primitives, Object, localized characters, etc.

A stream can be defined as a sequence of data. The InputStream is used to read data from a source and the OutputStream is used for writing data to a destination.

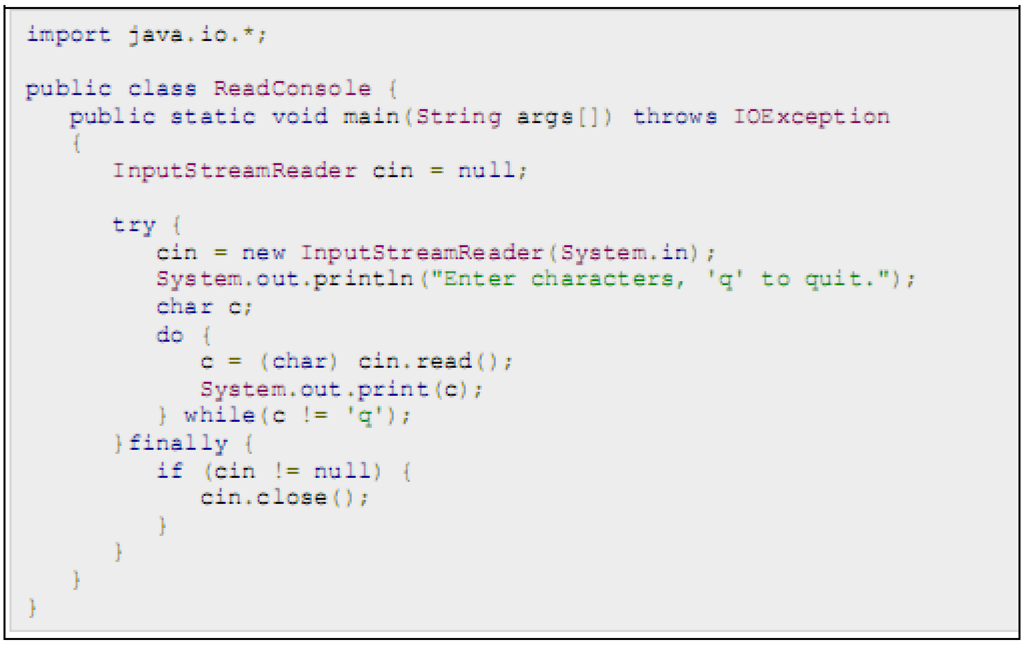
## Standard Streams

All the programming languages provide support for standard I/O where user's program can take input from a keyboard and then produce output on the computer screen. If you are aware if C or C++ programming languages, then you must be aware of three standard devices STDIN, STDOUT and STDERR.

Similar way, Java provides following three standard streams:

- **Standard Input**: This is used to feed the data to user's program and usually a keyboard is used as standard input stream and represented as System.in.
- **Standard Output:** This is used to output the data produced by the user's program and usually a computer screen is used to standard output stream and represented as System.out.
- **Standard Error:** This is used to output the error data produced by the user's program and usually a computer screen is used to standard error stream and represented as System.err.

Following is a simple program which creates InputStreamReader to read standard input stream until the user types a "q":



```
import java.io.*;

public class ReadConsole {
   public static void main(String args[]) throws IOException
   {
      InputStreamReader cin = null;

      try {
         cin = new InputStreamReader(System.in);
         System.out.println("Enter characters, 'q' to quit.");
         char c;
         do {
            c = (char) cin.read();
            System.out.print(c);
         } while(c != 'q');
      }finally {
         if (cin != null) {
            cin.close();
         }
      }
   }
}
```

## 8.8 Reading and Writing Files:

As described earlier, a stream can be defined as a sequence of data. The InputStream is used to read data from a source and the OutputStream is used for writing data to a destination.

Here is a hierarchy of classes to deal with Input and Output streams.

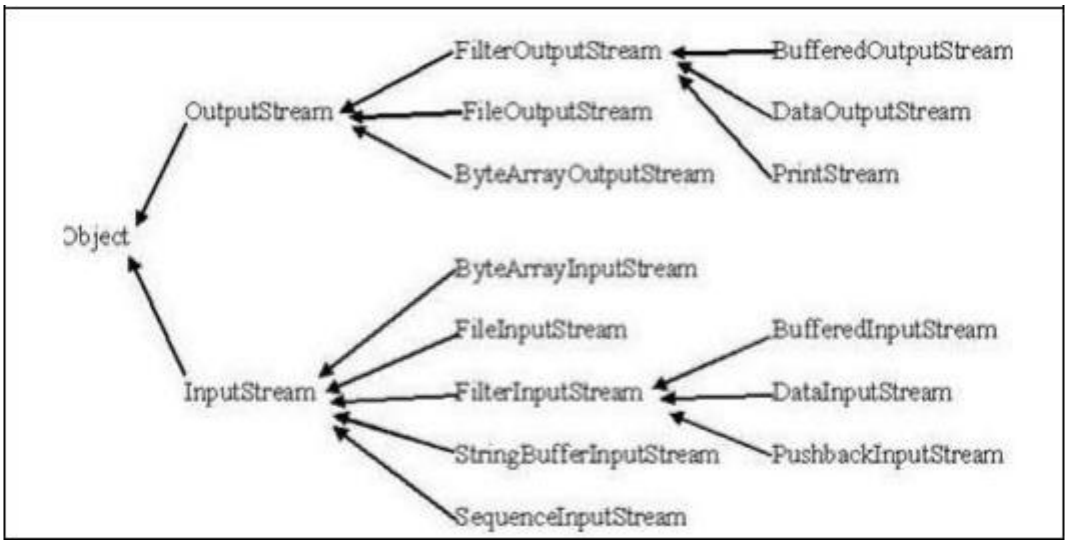


Figure 7.1 – Hierarchy of stream classes

The two important streams are FileInputStream and FileOutputStream, which would be discussed in this tutorial:

## 8.9 FileInputStream:

This stream is used for reading data from the files. Objects can be created using the keyword new and there are several types of constructors available.

Following constructor takes a file name as a string to create an input stream object to read the file:

```
InputStream f = new FileInputStream("C:/java/hello");
```

Following constructor takes a file object to create an input stream object to read the file. First we create a file object using File() method as follows:

```
File f = new File("C:/java/hello");
InputStream f = new FileInputStream(f);
```

FileOutputStream:

FileOutputStream is used to create a file and write data into it. The stream would create a file, if it does not already exist, before opening it for output.

Here are two constructors which can be used to create a FileOutputStream object. Following constructor takes a file name as a string to create an input stream object to write the file:

```
OutputStream f = new FileOutputStream("C:/java/hello")
```

Following constructor takes a file object to create an output stream object to write the file. First, we create a file object using File() method as follows:

```
File f = new File("C:/java/hello");
OutputStream f = new FileOutputStream(f);
```

## 8.10 File Navigation and I/O:

There are several other classes that we would be going through to get to know the basics of File Navigation and I/O.

- File Class
- FileReader Class
- FileWriter Class

## 8.11 File Class

Java File class represents the files and directory pathnames in an abstract manner. This class is used for creation of files and directories, file searching, file deletion etc.

The File object represents the actual file/directory on the disk. Following syntax creates a new File instance by converting the given pathname string into an abstract pathname.

```
File(String pathname)
```

Once you have File object in hand then there is a list of helper methods that can be used manipulate the files.

public String getName()

Returns the name of the file or directory denoted by this abstract pathname.

public String getPath()

Converts this abstract pathname into a pathname string.

public boolean isAbsolute()

Tests whether this abstract pathname is absolute. Returns true if this abstract pathname is absolute, false otherwise.

public boolean isDirectory()

Tests whether the file denoted by this abstract pathname is a directory. Returns true if and only if the file denoted by this abstract pathname exists and is a directory; false otherwise.

public boolean isFile()

Tests whether the file denoted by this abstract pathname is a normal file. A file is normal if it is not a directory and, in addition, satisfies other system-dependent criteria. Any non-directory file created by a Java application is guaranteed to be a normal file. Returns true if and only if the file denoted by this abstract pathname exists and is a normal file; false otherwise.

public long lastModified()

Returns the time that the file denoted by this abstract pathname was last modified. Returns a long value representing the time the file was last modified, measured in milliseconds since the epoch (00:00:00 GMT, January 1, 1970), or 0L if the file does not exist or if an I/O error occurs.

public long length()

Returns the length of the file denoted by this abstract pathname. The return value is unspecified if this pathname denotes a directory.

Example:

Following is the example to demonstrate File object:

```
import java.io.File;

public class FileDemo {
  public static void main(String[] args) {
    File f = null;
     String[] strs = {"test1.txt", "test2.txt"};
     try{
        //for each string in string array
        for(String s:strs )
        {
          // create new file
          f= new File(s);

          //true if the file is executable

          boolean bool = f.canExecute();

          // find the absolute path String
          a = f.getAbsolutePath();

          // prints absolute path
          System.out.print(a);

          // prints
          System.out.println(" is executable: "+ bool);
        }
      }catch(Exception e){
         // if any I/O error
        e.printStackTrace();
      }
    }
  }
```

## 8.12 FileReader Class

This class inherits from the InputStreamReader class. FileReader is used for reading streams of characters. This class has several constructors to create required objects.

Following syntax creates a new FileReader, given the File to read from:

FileReader(File file)

Following syntax creates a new FileReader, given the FileDescriptor to read from.

FileReader(FileDescriptor fd)

Following syntax creates a new FileReader, given the name of the file to read from.

FileReader(String fileName)

Once you have FileReader object in hand then there is a list of helper methods which can be used manipulate the files

public int read() throws IOException

Reads a single character. Returns an int, which represents the character read.

public int read(char [] c, int offset, int len)
Reads characters into an array. Returns the number of characters read

**Example:**
Following is the example to demonstrate class:

```
import java.io.*;
public class FileRead{

  public static void main(String args[])throws IOException{

    File file = new File("Hello1.txt");

    //Creates a FileReader Object
    FileReader fr = new FileReader(file);
    char [] a = new char[50];
    fr.read(a); // reads the content to the array
    for(char c : a)
       System.out.print(c); //prints the characters one by one
    fr.close();
  }
}
```

## 8.13 FileWriter Class

This class inherits from the OutputStreamWriter class. The class is used for writing streams of characters. This class has several constructors to create required objects.

Following syntax creates a FileWriter object given a File object:

FileWriter(File file)

Following syntax creates a FileWriter object given a File object.
FileWriter(File file, boolean append)

Once you have FileWriter object in hand, then there is a list of helper methods, which can be used manipulate the files.

public void write(int c) throws IOException
Writes a single character.

public void write(char [] c, int offset, int len)

Writes a portion of an array of characters starting from offset and with a length of len.

public void write(String s, int offset, int len)

Write a portion of a String starting from offset and with a length of len.

Example:

```
import java.io.*;
public class FileRead{

  public static void main(String args[])throws IOException{

    File file = new File("Hello1.txt");
    • creates the file
    file.createNewFile();
    • creates a FileWriter Object
    FileWriter writer = new FileWriter(file);
    • Writes the content to the file
    writer.write("This\n is\n an\n example\n");
    writer.flush();
    writer.close();

    //Creates a FileReader Object
    FileReader fr = new FileReader(file);
    char [] a = new char[50];
    fr.read(a); // reads the content to the array
    for(char c : a)
       System.out.print(c); //prints the characters one by one
    fr.close();
  }
}
```

## 8.14 Summary

At the end of this unit, a student should be able to:

1. connect to a database (drivers and syntax)
2. add, delete and modify records from the application
3. carry out Simple I/O and Read/Write/Modify files.

## 8.15 Activities

1. Write a program that will count the number of lines in each file that is specified by the user. Assume that the files are text files. Note that multiple files can be specified, as in:

```
file1.txt file2.txt file3.txt
```

Write each file name, along with the number of lines in that file, to standard output. If an error occurs while trying to read from one of the files, you should print an error message for that file, but you should still process all the remaining files.

2. Create a database application where a user can input, update, delete and view a list of contacts. Fields to keep are name, address, telephone and email.

# UNIT 9 Classes and Objects

## 9.1 Introduction

In this chapter, we will look into the concepts Classes and Objects.

- **Object** - Objects have states and behaviours. Example: A dog has states-colour, name, breed as well as **behaviours** -wagging, barking, and eating. An object is an instance of a class.
- **Class** - A class can be defined as a template/blue print that describes the behaviours/states that object of its type support.

## 9.2 Unit Objectives

The objectives of this unit are to:

- How to declare a class and use it to create an object.
- How to implement a class's behaviors as methods.
- How to implement a class's attributes as instance variables and properties.
- How to call an object's methods to make them perform their tasks.
- What instance variables of a class and local variables of a method are?
- How to use a constructor to initialize an object's data.
- The differences between primitive and reference types.

## 9.3 Object Orientation as a New Paradigm: The Big Picture

It is claimed that the problem-solving techniques used in object-oriented programming more closely models the way humans solve day-to-day problems.

So let us consider how we solve an everyday problem: Suppose you wanted to send flowers to a friend named Robin who lives in another city. To solve this problem you simply walk to your nearest florist run by, let's say, Fred. You tell Fred the kinds of flowers to send and the address to which they should be delivered. You can be assured that the flowers will be delivered.

Now, let us examine the mechanisms used to solve your problem.

• You first found an appropriate agent (Fred, in this case) and you passed to this agent a message containing a request.

• It is the responsibility of Fred to satisfy the request.

• There is some method (an algorithm or set of operations) used by Fred to do this.

• You do not need to know the particular methods used to satisfy the request such information are hidden from view.

Of course, you do not want to know the details, but on investigation, you may find that Fred delivered a slightly different message to another florist in the city where your friend Robin lives. That florist then passes another message to a subordinate who makes the floral arrangement. The flowers, along with yet another message, is passed onto a delivery person and so on. The florists also has interactions with wholesalers who, in turn, had interactions with flower growers and so on.

This leads to our first conceptual picture of object-oriented programming:

> *An object-oriented program is structured as* ***community*** *of interacting agents called* ***objects****. Each object has a role to play. Each object provides a* ***service*** *or performs an action that is used by other members of the community.*

**Messages and Responsibilities**

Members of an object-oriented community make requests of each other.
The next important principle explains the use of messages to initiate action:

> *Action is initiated in object-oriented programming by the transmission of a* ***message*** *to an agent (an* ***object****) responsible for the actions. The message encodes the request for an action and is accompanied by any additional information (arguments/parameters) needed to carry out the request. The* ***receiver*** *is the object to whom the message is sent. If the receiver accepts the message, it accepts* ***responsibility*** *to carry out the indicated action. In response to a message, the receiver will perform some* ***method*** *to satisfy the request.*

There are some important issues to point out here:

• The client sending the request needs not know the means by which the request is carried out. In this, we see the principle of information hiding.

• Another principle implicit in message passing is the idea of finding someone else to do the work i.e. reusing components that may have been written by someone else.

• The interpretation of the message is determined by the receiver and can vary with different receivers. For example, if you sent the message “deliver flowers” to a friend, she will probably have understood what was required and flowers would still have been delivered but the method she used would have been very different from that used by the florist.

• In object-oriented programming, behavior is described in terms of responsibilities.

• Client’s requests for actions only indicate the desired outcome. The receivers are free to pursue any technique that achieves the desired outcomes.

• Thinking in this way allows greater independence between objects.

• Thus, objects have responsibilities that they are willing to fulfill on request. The collection of responsibilities associated with an object is often called a protocol.

## Classes and Instances

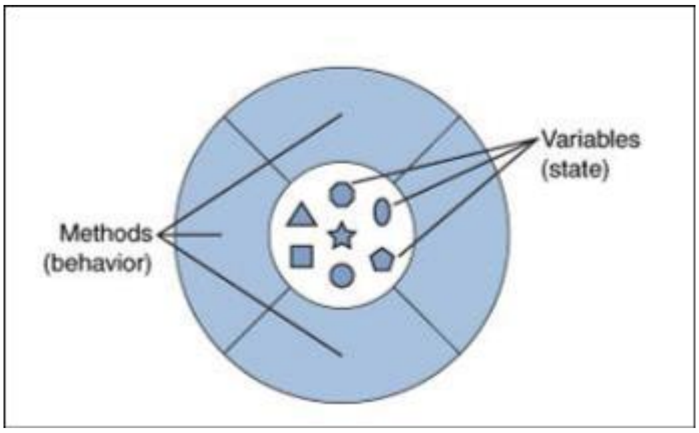
The next important principle of object-oriented programming is:

> *All objects are instances of a class. The method invoked by an object in response to a message is determined by the class of the receiver. All objects of a given class use the same method in response to similar messages.*

Fred is an instance of a category or class of people i.e. Fred is an instance of a class of florists. The term florist represents a class or category of all florists. Fred is an object or instance of a class.

We interact with instances of a class but the class determines the behaviour of instances. We can tell a lot about how Fred will behave by understanding how Florists behave. We know, for example, that Fred, like all florists can arrange and deliver flowers.

In the real world, there is this distinction between classes and objects. Real-world objects share two characteristics: They all have state and behaviour. For example, dogs have state (name, colour, breed, hungry) and behaviour (barking, fetching, wagging tail). Students have state (name, student number, courses they are registered for, gender) and behaviour (take tests, attend courses, write tests, party).



**An Object**

## 9.4 Differences between Structured and OO Programming

| | |
|---|---|
| Structured Programming is designed which focuses on **process**/ logical structure and then data required for that process. | Object Oriented Programming is designed which focuses on **data**. |
| Structured programming follows **top-down approach**. | Object oriented programming follows **bottom-up approach**. |
| Structured Programming is also known as **Modular Programming** and a subset of **procedural programming language**. | Object Oriented Programming supports **inheritance, encapsulation, abstraction, polymorphism**, etc. |
| In Structured Programming, Programs are divided into small self contained **functions**. | In Object Oriented Programming, Programs are divided into small entities called **objects**. |
| Structured Programming is **less** secure as there is no way of **data hiding**. | Object Oriented Programming is more secure as having data hiding feature. |
| Structured Programming can solve **moderately** complex programs. | Object Oriented Programming can solve any **complex** programs. |
| Structured Programming provides **less reusability**, more function dependency. | Object Oriented Programming provides more reusability, less function **dependency**. |
| Less abstraction and less flexibility. | More abstraction and more **flexibility**. |

## 9.5 Benefits of OOP

- We can build the programs from standard working modules that communicate with one another, rather than having to start writing the code from scratch which leads to saving of development time and higher productivity.
- OOP language allows to break the program into the bit-sized problems that can be solved easily (one object at a time).
- The technology promises greater programmer productivity, better quality of software and lesser maintenance cost.
- OOP systems can be easily upgraded from small to large systems.
- It is possible that multiple instances of objects co-exist without any interference,
- It is very easy to partition the work in a project based on objects.
- It is possible to map the objects in problem domain to those in the program.
- The principle of data hiding helps the programmer to build secure programs which cannot be invaded by the code in other parts of the program.
- By using inheritance, we can eliminate redundant code and extend the use of existing classes, hence re-use.
- Message passing techniques is used for communication between objects which makes the interface descriptions with external systems much simpler.
- The data-centered design approach enables us to capture more details of model in an implementable form.

## 9.6 Objects and Classes

### Objects

In object-oriented programming, we create software objects that model real world objects. Software objects are modelled after real-world objects in that they too have state and behavior. A software object maintains its state in one or more variables. A variable is an item of data named by an identifier. A software object implements its behavior with methods. A method is a function associated with an object.

*Definition: An object is a software bundle of variables and related methods.*

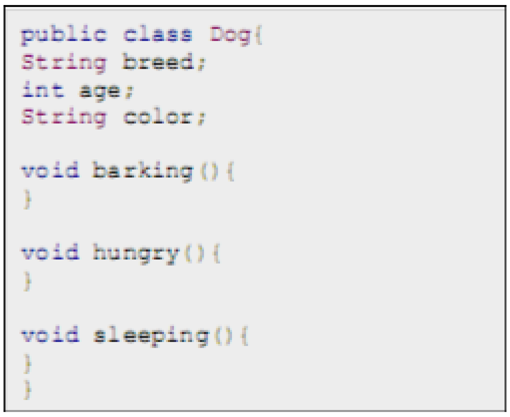
An object is also known as an instance. An instance refers to a particular object. For e.g. Karuna’s bicycle is an instance of a bicycle—it refers to a particular bicycle. Sandile Zuma is an instance of a Student.

### Classes

In object-oriented software, it is possible to have many objects of the same kind that share characteristics: rectangles, employee records, video clips, and so on. A class is a software blueprint for objects. A class is used to manufacture or create objects. The class declares the instance variables necessary to contain the state of every object. The class would also declare and provide implementations for the instance methods necessary to operate on the state of the object.

*Definition: A class is a blueprint that defines the variables and the methods common to all objects of a certain kind.*

A sample of a class is given below:



```
public class Dog{
String breed;
int age;
String color;

void barking(){
}

void hungry(){
}

void sleeping(){
}
}
```

A class can contain any of the following variable types.

***Local variables:*** Variables defined inside methods, constructors or blocks are called local variables. The variable will be declared and initialized within the method and the variable will be destroyed when the method has completed.

***Instance variables:*** Instance variables are variables within a class but outside any method. These variables are instantiated when the class is loaded. Instance variables can be accessed from inside any method, constructor or blocks of that particular class.

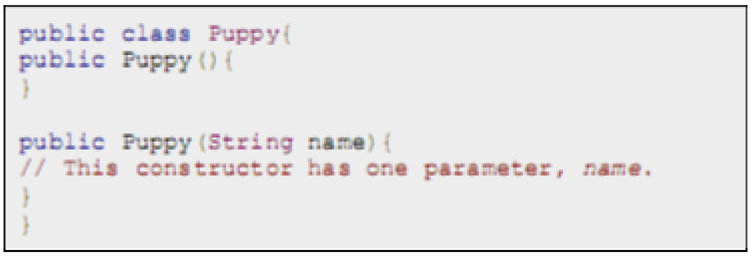
***Class variables:*** Class variables are variables declared within a class, outside any method, with the static keyword.

A class can have any number of methods to access the value of various kinds of methods. In the above example, barking(), hungry() and sleeping() are methods.

## 9.7 Constructors

When discussing about classes, one of the most important subtopic would be constructors. Every class has a constructor. If we do not explicitly write a constructor for a class, the Java compiler builds a default constructor for that class. Each time a new object is created, at least one constructor will be invoked.

The main rule of constructors is that they should have the same name as the class. A class can have more than one constructor (with different number of parameters). Example of a constructor is given below:



```
public class Puppy{
public Puppy(){
}

public Puppy(String name){
// This constructor has one parameter, name.
}
}
```

A no-arg constructor is a constructor with no arguments. A constructor that has parameters is known as parameterized constructor.

**Creating an Object:**

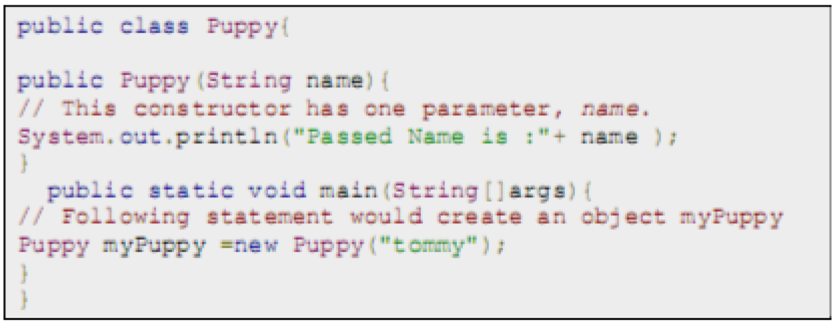
As mentioned previously, a class provides the blueprints for objects. So, basically an object is created from a class.

In Java, the new keyword is used to create new objects. The new keyword calls the constructor where we can initialize variables.

There are three steps when creating an object from a class:

- **Declaration:** A variable declaration with a variable name with an object type.
- **Instantiation:** The 'new' keyword is used to create the object.
- **Initialization:** The 'new' keyword is followed by a call to a constructor. This call initializes the new object.

Example of creating an object is given below:



```
public class Puppy{

public Puppy(String name){
// This constructor has one parameter, name.
System.out.println("Passed Name is :"+ name );
}
  public static void main(String[]args){
// Following statement would create an object myPuppy
Puppy myPuppy =new Puppy("tommy");
}
}
```

If we compile and run the above program, then it would produce the following result:

*Passed Name is: tommy*

### 9.8 Accessing Instance Variables and Methods:

Instance variables and methods are accessed via created objects. To access an instance variable, the fully qualified path should be as follows:

```
/* First create an object */
ObjectReference = new Constructor();

/* Now call a variable as follows */
ObjectReference.variableName;

/* Now you can call a class method as follows */
ObjectReference.MethodName();
```

**Example:**

This example explains how to access instance variables and methods of a class:

```
public class Puppy{

int puppyAge;

public Puppy(String name){
// This constructor has one parameter, name.
System.out.println("Passed Name is :"+ name );
}
public void setAge(int age ){
puppyAge = age;
}

public int getAge(){
System.out.println("Puppy's age is :"+ puppyAge );
return puppyAge;
}
  public static void main(String[]args){
/* Object creation */
Puppy myPuppy =newPuppy("tommy");

/* Call class method to set puppy's age */
myPuppy.setAge(2);

/* Call another class method to get puppy's age */
    myPuppy.getAge();

/* You can access instance variable as follows as well */
System.out.println("Variable Value :"+ myPuppy.puppyAge );
}
}
```

If we compile and run the above program, then it would produce the following result:

***Passed Name is: tommy***

***Puppy's age is: 2***

***Variable Value: 2***

### 9.9 Source file declaration rules:

As the last part of this section, let us now look into the source file declaration rules. These rules are essential when declaring classes, import statements and package statements in a source file.

- There can be only one public class per source file.
- A source file can have multiple non-public classes.
- The public class name should be the name of the source file as well which should be appended by .java at the end. For example: The class name is *.public class Employee{}* Then the source file should be as *Employee.java*.

- If the class is defined inside a package, then the package statement should be the first statement in the source file.
- If import statements are present, then they must be written between the package statement and the class declaration. If there are no package statements, then the import statement should be the first line in the source file.
- Import and package statements will imply to all the classes present in the source file. It is not possible to declare different import and/or package statements to different classes in the source file.

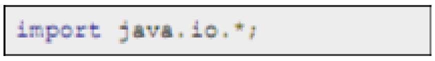
## 9.10 Java Package:

In simple, it is a way of categorizing the classes and interfaces. When developing applications in Java, hundreds of classes and interfaces will be written, therefore categorizing these classes is necessary as well as makes life much easier.

**Import statements:**

In Java if a fully qualified name, which includes the package and the class name, is given, then the compiler can easily locate the source code or classes. Import statement is a way of giving the proper location for the compiler to find that particular class.

For example, the following line would ask compiler to load all the classes available in directory java_installation/java/io.



```
import java.io.*;
```

## 9.11 Summary

• Each class declaration that begins with the access modifier public must be stored in a file that has exactly the same name as the class and ends with the .java file-name extension.

• Every class declaration contains keyword class followed immediately by the class's name.

• A method declaration that begins with keyword public indicates that the method can be called by other classes declared outside the class declaration.

•Keyword void indicates that a method will perform a task but will not return any information.

• By convention, method names begin with a lowercase first letter and all subsequent words in the name begin with a capital first letter.

• Empty parentheses following a method name indicate that the method does not require any parameters to perform its task.

• Every method's body is delimited by left and right braces ({ and }).

• The method's body contains statements that perform the method's task. After the statements execute, the method has completed its task.

• When you attempt to execute a class, Java looks for the class's main method to begin execution.

• Typically, you cannot call a method of another class until you create an object of that class.

• A class instance creation expression begins with keyword new and creates a new object.

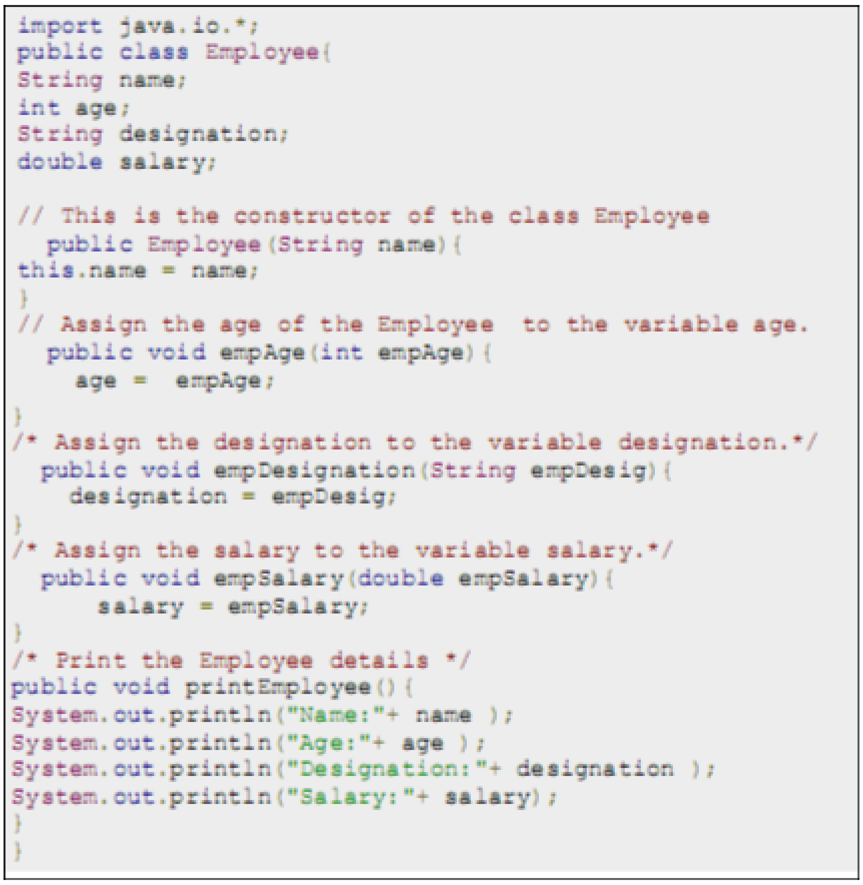
• To call a method of an object, follow the variable name with a dot separator (.) the method name and a set of parentheses containing the method's arguments.

## 9.12 Activity

For our activity, we will be creating two classes. They are Employee and EmployeeTest.

First open Eclipse, create a new java program and add the following code. Remember this is the Employee class and the class is a public class. Now, save this source file with the name Employee.java.

The Employee class has four instance variables name, age, designation and salary. The class has one explicitly defined constructor, which takes a parameter.

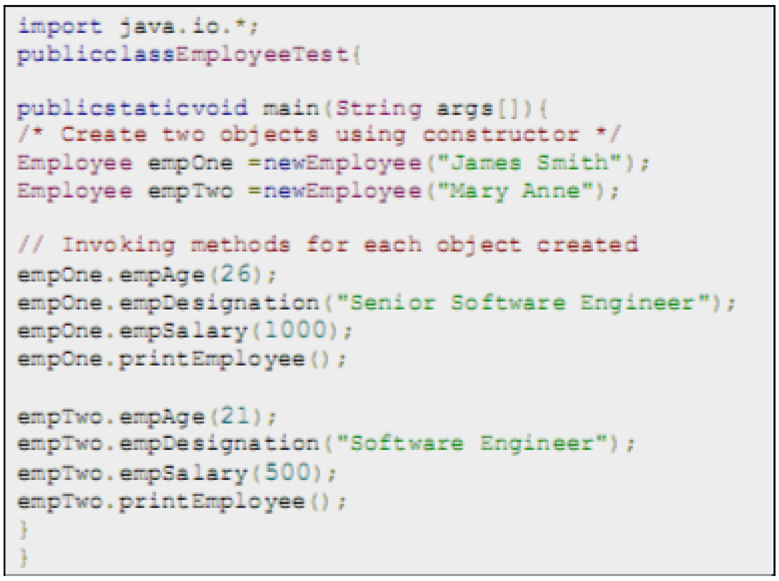


```java
import java.io.*;
public class Employee{
String name;
int age;
String designation;
double salary;

// This is the constructor of the class Employee
  public Employee(String name){
this.name = name;
}
// Assign the age of the Employee  to the variable age.
  public void empAge(int empAge){
    age =  empAge;
}
/* Assign the designation to the variable designation.*/
  public void empDesignation(String empDesig){
    designation = empDesig;
}
/* Assign the salary to the variable salary.*/
  public void empSalary(double empSalary){
      salary = empSalary;
}
/* Print the Employee details */
public void printEmployee(){
System.out.println("Name:"+ name );
System.out.println("Age:"+ age );
System.out.println("Designation:"+ designation );
System.out.println("Salary:"+ salary);
}
}
```

As mentioned previously, processing starts from the main method. Therefore, in-order for us to run this Employee class there should be *main* method and objects should be created. We will be creating a separate class for these tasks.

Given below is the EmployeeTest class, which creates two instances of the class Employee and invokes the methods for each object to assign values for each variable.

Save the following code in EmployeeTest.java file:



```
import java.io.*;
publicclassEmployeeTest{

publicstaticvoid main(String args[]){
/* Create two objects using constructor */
Employee empOne =newEmployee("James Smith");
Employee empTwo =newEmployee("Mary Anne");

// Invoking methods for each object created
empOne.empAge(26);
empOne.empDesignation("Senior Software Engineer");
empOne.empSalary(1000);
empOne.printEmployee();

empTwo.empAge(21);
empTwo.empDesignation("Software Engineer");
empTwo.empSalary(500);
empTwo.printEmployee();
}
}
```

Now run the EmployeeTest.Java and comment on the results.

# UNIT 10 Encapsulation

## 10.1 Introduction

Encapsulation is one of the four fundamental OOP concepts. The other three are inheritance, polymorphism, and abstraction.

Encapsulation is the technique of making the fields in a class private and providing access to the fields via public methods. If a field is declared private, it cannot be accessed by anyone outside the class, thereby hiding the fields within the class. For this reason, encapsulation is also referred to as data hiding.

## 10.2 Unit Objectives

The objectives of this unit are to:

- understand the concept of encapsulation
- reap the benefits of using encapsulation
- familiarize with the foundation of encapsulation through examples

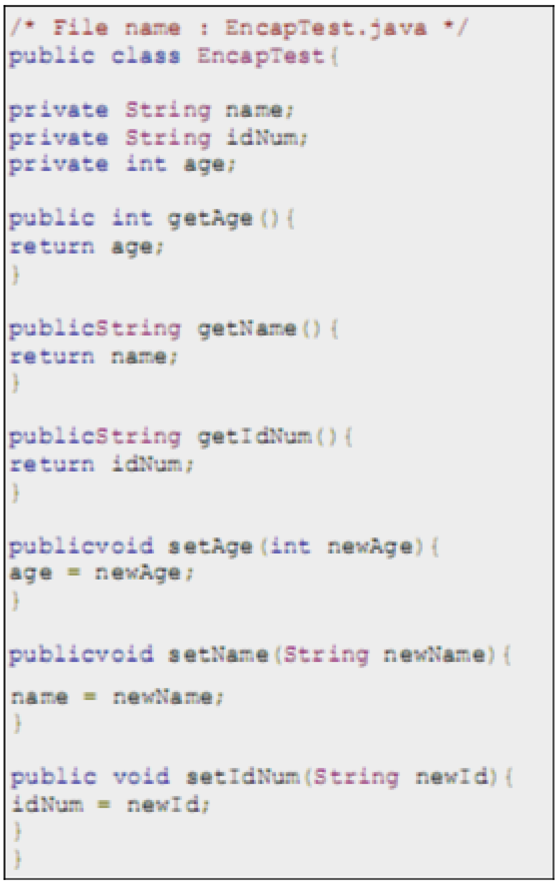
## 10.3 Encapsulation with example

Encapsulation can be described as a protective barrier that prevents the code and data being randomly accessed by other code defined outside the class. Access to the data and code is tightly controlled by an interface.

The main benefit of encapsulation is the ability to modify our implemented code without breaking the code of others who use our code. With this feature, Encapsulation gives maintainability, flexibility and extensibility to our code.

**Example:**

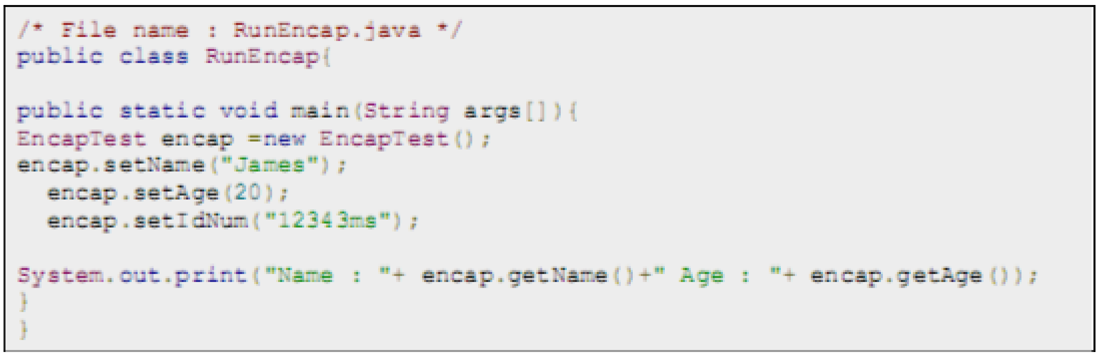
Let us look at an example that depicts encapsulation:



```java
/* File name : EncapTest.java */
public class EncapTest{

private String name;
private String idNum;
private int age;

public int getAge(){
return age;
}

publicString getName(){
return name;
}

publicString getIdNum(){
return idNum;
}

publicvoid setAge(int newAge){
age = newAge;
}

publicvoid setName(String newName){

name = newName;
}

public void setIdNum(String newId){
idNum = newId;
}
}
```

The public methods are the access points to this class fields from the outside java world. Normally, these methods are referred as getters and setters (or accessors and mutators). Therefore, any class that wants to access the variables should access them through these getters and setters. When you use Eclipse or NetBeans, you can generate these setters and getters for all instance variables from the menu.

The variables of the EncapTest class can be accessed as below:



```java
/* File name : RunEncap.java */
public class RunEncap{

public static void main(String args[]){
EncapTest encap =new EncapTest();
encap.setName("James");
  encap.setAge(20);
  encap.setIdNum("12343ms");

System.out.print("Name : "+ encap.getName()+" Age : "+ encap.getAge());
}
}
```

This would produce the following result:

```
Name: James Age: 20
```

## 10.4 Benefits of Encapsulation:

The fields of a class can be made read-only or write-only.

A class can have total control over what is stored in its fields. The users of a class do not need to know how the class stores its data. A class can change the data type of a field and users of the class do not need to change any of their code.

## 10.5 Summary

- Encapsulation is the bringing together of a set of attributes and methods into an object definition and hiding their implementation structure from the object's users.

- Encapsulation is supported by two subordinate concepts: **Bundling and Information Hiding.** Bundling is the act of associating a set of methods with a set of data as the only means of affecting the values of the data. Information Hiding refers to the hiding of internal representation of data and methods from the users of these data and methods.

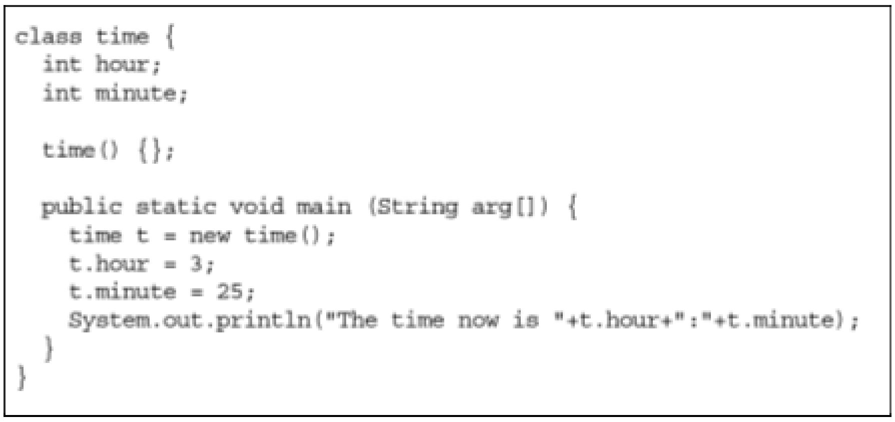
- Encapsulation enhances software maintainability by limiting the ripple effects, resulting from a change in object definition, from affecting other objects.

## 10.6 Activities

1. What is encapsulation? How does encapsulation contribute to software maintainability?

2. How does the code in code listing below measure up to the principle of encapsulation? Comment.

3. How would you enhance the code in the figure beow to achieve the desired effect of encapsulation? What is the trade-off of your enhancement? What are its advantages?



```
class time {
  int hour;
  int minute;

  time() {};

  public static void main (String arg[]) {
    time t = new time();
    t.hour = 3;
    t.minute = 25;
    System.out.println("The time now is "+t.hour+":"+t.minute);
  }
}
```

Code Listing

4. Create a class named Calculate.
    - Add 2 instance variables, Quantity and UnitPrice. UnitPrice can include Rupees and Cents.
    - Add a no-argument constructor.
    - Add getters and setters for the 2 instance variables.
    - Add a user defined method named Total with no arguments. Total will calculate and return Quantity*UnitPrice. If Quantity is greater than 10, give 5% discount.
    - Add a user defined method to display the total amount to the user.
    - Create a Java Main application to create 2 objects based on the Calculate class to test the program.

# UNIT 11 Inheritance

## 11.1 Introduction

Inheritance can be defined as the process where one object acquires the properties (both attributes and methods) of another. With the use of inheritance, the information is made manageable in a hierarchical order.

When we talk about inheritance, the most commonly used keyword would be extends and implements. These words would determine whether one object IS-A type of another. By using these keywords, we can make one object acquire the properties of another object.

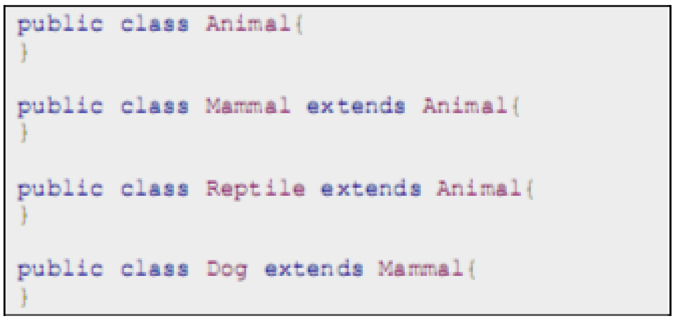
## 11.2 Unit Objectives

The objectives of this unit are to:

- How inheritance promotes software reusability.
- The notions of superclasses, subclasses, and the relationship between them. (also known as parent and child classes).
- To use keyword extends to create a class that inherits attributes and behaviors from another class.
- To use access modifier protected to give subclass methods access to superclass members.
- To access superclass members with super.
- How constructors are used in inheritance hierarchies.

## 11.3 IS-A Relationship

IS-A is a way of saying: This object is a type of that object. Let us see how the extends keyword is used to achieve inheritance.



```
public class Animal{
}

public class Mammal extends Animal{
}

public class Reptile extends Animal{
}

public class Dog extends Mammal{
}
```

Now, based on the above example, In Object Oriented terms the following are true:

- Animal is the superclass of Mammal class.
- Animal is the superclass of Reptile class.
- Mammal and Reptile are subclasses of Animal class.
- Dog is the subclass of both Mammal and Animal classes.

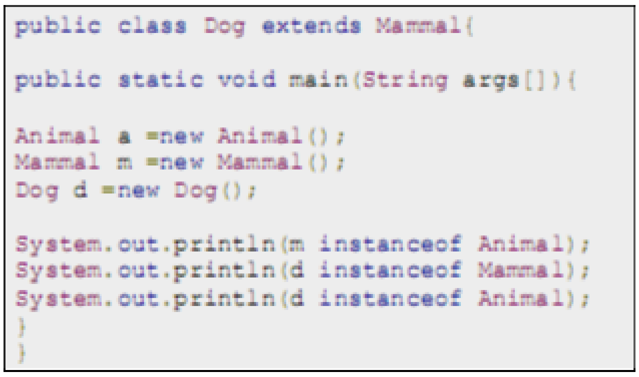
Now, if we consider the IS-A relationship, we can say:

- Mammal IS-A Animal
- Reptile IS-A Animal
- Dog IS-A Mammal
- Hence : Dog IS-A Animal as well

With use of the extends keyword the subclasses will be able to inherit all the properties (attributes and methods) of the superclass except for the private properties of the superclass.

We can assure that Mammal is actually an Animal with the use of the instance operator.

**Example**



```
public class Dog extends Mammal{

public static void main(String args[]){

Animal a =new Animal();
Mammal m =new Mammal();
Dog d =new Dog();

System.out.println(m instanceof Animal);
System.out.println(d instanceof Mammal);
System.out.println(d instanceof Animal);
}
}
```

This would produce the following result:

**true**

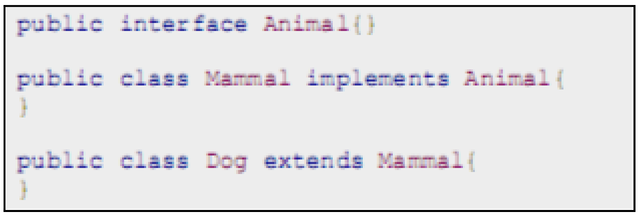
**true**

**true**

Since we have a good understanding of the extends keyword, let us look into how the implements keyword is used to get the IS-A relationship.

The implements keyword is used by classes by inherit from interfaces. Interfaces can never be extended by the classes.

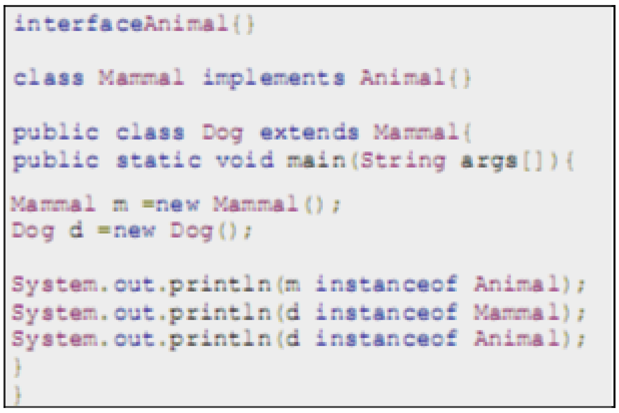
**Example:**



```
public interface Animal{}

public class Mammal implements Animal{
}

public class Dog extends Mammal{
}
```

**The instance of Keyword:**

Let us use the instance of operator to check determine whether Mammal is actually an Animal, and dog is actually an Animal



```
interfaceAnimal{}

class Mammal implements Animal{}

public class Dog extends Mammal{
public static void main(String args[]){

Mammal m =new Mammal();
Dog d =new Dog();

System.out.println(m instanceof Animal);
System.out.println(d instanceof Mammal);
System.out.println(d instanceof Animal);
}
}
```

This would produce the following result:

**true**

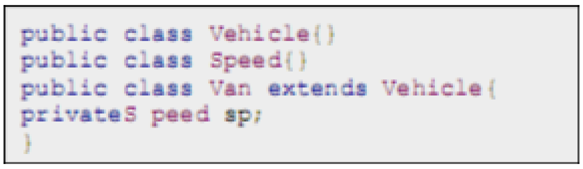
**true**

**true**

### 11.4 HAS-A relationship:

These relationships are mainly based on the usage. This determines whether a certain class HAS-A certain thing. This relationship helps to reduce duplication of code as well as bugs.

Let us look into an example:

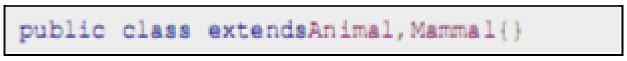


```
public class Vehicle{}
public class Speed{}
public class Van extends Vehicle{
privateS peed sp;
}
```

This shows that class Van HAS-A Speed. By having a separate class for Speed, we do not have to put the entire code that belongs to speed inside the Van class that makes it possible to reuse the Speed class in multiple applications.

In Object-Oriented feature, the users do not need to bother about which object is doing the real work. To achieve this, the Van class hides the implementation details from the users of the Van class. So what happens is the users would ask the Van class to do a certain action and the Van class will either do the work by itself or ask another class to perform the action.

A very important fact to remember is that Java only supports only single inheritance. This means that a class cannot extend more than one class. Therefore the following is illegal:



```
public class extendsAnimal,Mammal{}
```

However, a class can implement one or more interfaces. This has made Java get rid of the impossibility of multiple inheritance.

## 11.5 Summary

- A class can take on the properties of a superclass via the inheritance mechanism.
- Inheritance is the ability of a subclass to take on the general properties of classes higher up in a class hierarchy.
- Properties can only be propagated downward from a superclass to a sub-class.
- Inheritance enables code reuse.
- Inheritance enhances software maintainability.
- Inheritance enables class extension through subclassing.
- A class that takes on properties from only one superclass is said to exhibit single inheritance.

## 11.6 Activities

1. Consider the following two classes:

```
public class ClassA {
    public void methodOne(int i) {
    }
    public void methodTwo(int i) {
    }
    public static void methodThree(int i) {
    }
    public static void methodFour(int i) {
    }
}
public class ClassB extends ClassA { public
    static void methodOne(int i) {
    }
    public void methodTwo(int i) {
    }
    public void methodThree(int i) {
    }
    public static void methodFour(int i) {
    }
}
```

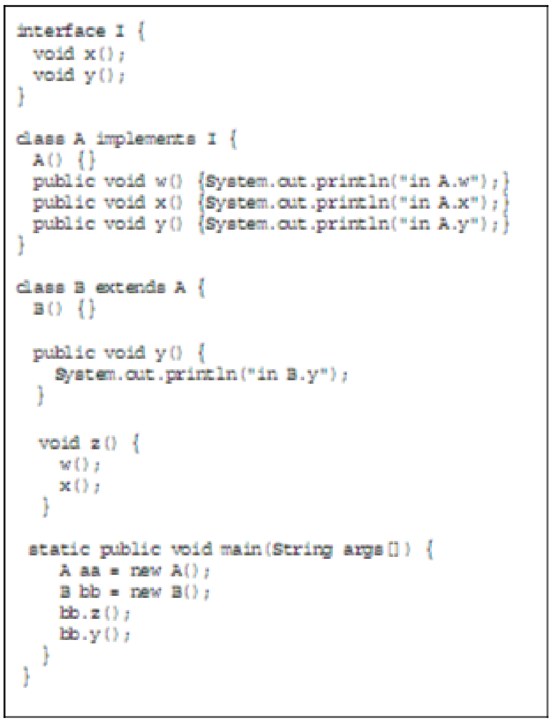
   a. Which method overrides a method in the superclass?

   b. Which method hides a method in the superclass?

   c. What do the other methods do?

2. Give an example of multiple inheritance in a real-life situation.

3. What is the expected output for the code in Figure 6.8.1?



```
interface I {
  void x();
  void y();
}

class A implements I {
  A() {}
  public void w() {System.out.println("in A.w");}
  public void x() {System.out.println("in A.x");}
  public void y() {System.out.println("in A.y");}
}

class B extends A {
  B() {}

  public void y() {
    System.out.println("in B.y");
  }

   void z() {
     w();
     x();
   }

 static public void main(String args[]) {
     A aa = new A();
     B bb = new B();
     bb.z();
     bb.y();
   }
 }
```

4. Write a Java program that will ask a user to input his account balance, account type (1 for debit, 2 for credit) and transaction amount.

   Steps:

- Write a class BankAccount with constructor, getters and setters, a method to credit the account balance, a method to debit the account balance and a method to display the account balance.
- Test the program by inputting any account balance, account type and transaction amount, and check the output whether the account balance is correctly debited or credited by the transaction amount.
- Write another class that inherits from the above class. Add a new method for that sub class called "Adjust" that will add or subtract a value from the balance.
- In that subclass, override the display method of the superclass by anything different.
- Test the program by creating a new object, and calling the "Adjust" method.

# UNIT 12 Polymorphism

## 12.1 Introduction

Polymorphism is the ability of an object to take on many forms. The most common use of polymorphism in OOP occurs when a parent class reference is used to refer to a child class object.

Any Java object that can pass more than one IS-A test is considered to be polymorphic. In Java, all Java objects are polymorphic since any object will pass the IS-A test for their own type and for the class Object.

## 12.2 Unit Objectives

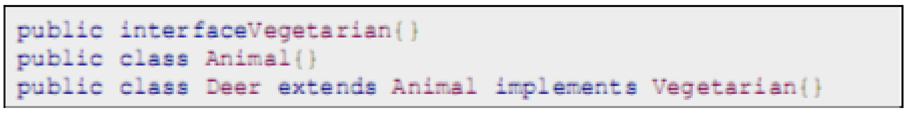
The objectives of this unit are to learn:

- The concept of polymorphism.
- To use overridden methods to effect polymorphism.
- To distinguish between abstract and concrete classes.
- To declare abstract methods to create abstract classes.
- How polymorphism makes systems extensible and maintainable.

## 12.3 Polymorphism

It is important to know that the only possible way to access an object is through a reference variable. A reference variable can be of only one type. Once declared, the type of a reference variable cannot be changed. The reference variable can be reassigned to other objects provided that it is not declared final. The type of the reference variable would determine the methods that it can invoke on the object.

A reference variable can refer to any object of its declared type or any subtype of its declared type. A reference variable can be declared as a class or interface type.

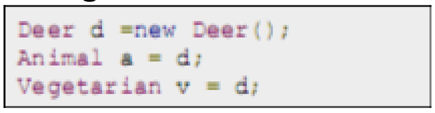


```
public interfaceVegetarian{}
public class Animal{}
public class Deer extends Animal implements Vegetarian{}
```

**Example:**
Now, the Deer class is considered to be polymorphic since this has multiple inheritances. Following are true for the above example:

- A Deer IS-A Animal
- A Deer IS-A Vegetarian
- A Deer IS-A Deer
- A Deer IS-A Object

When we apply the reference variable facts to a Deer object reference, the following declarations are legal:

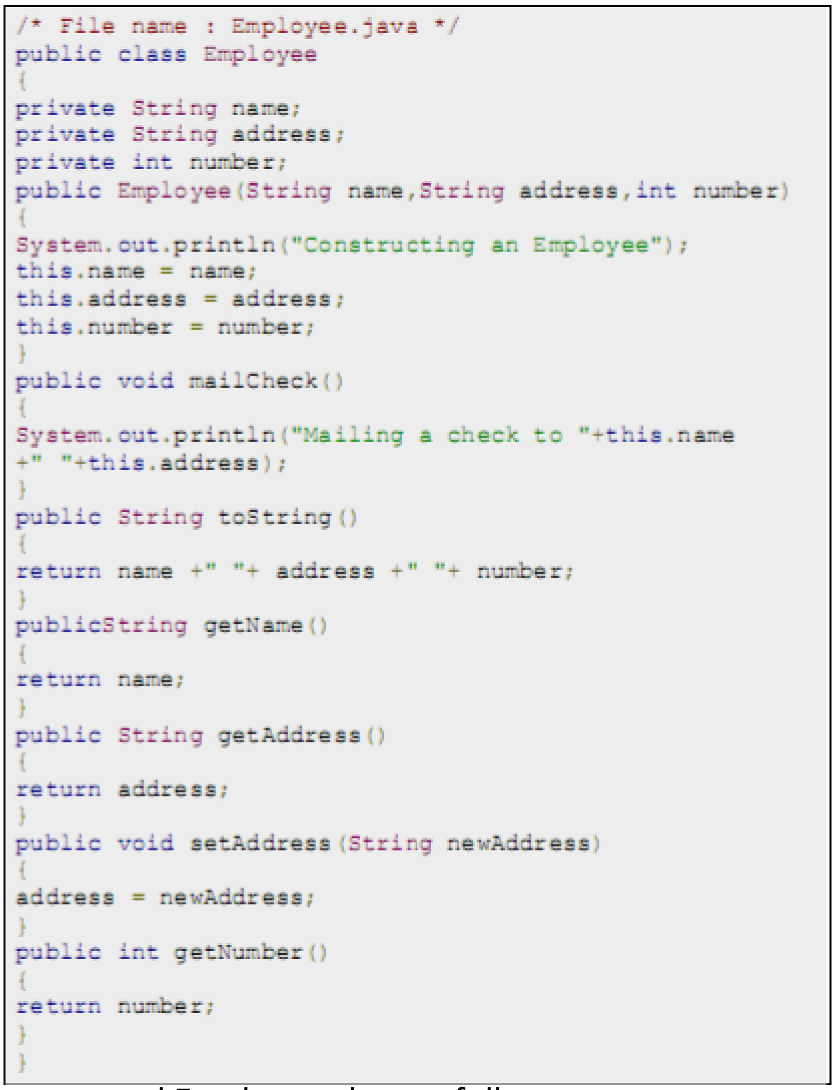


```
Deer d =new Deer();
Animal a = d;
Vegetarian v = d;
```

All the reference variables d, a and v refer to the same Deer object in the heap.

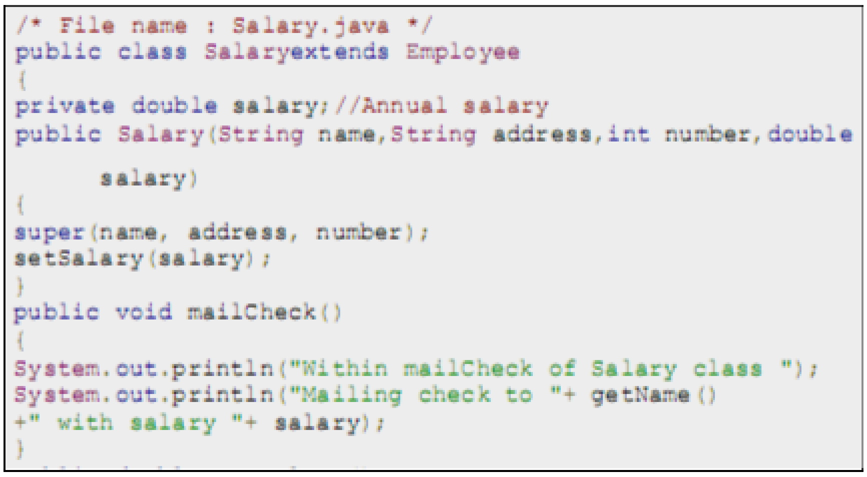
### 12.4 Virtual Methods, Super keyword

This section will show you how the behavior of overridden methods in Java allows you to take advantage of polymorphism when designing your classes. We already have discussed method overriding, where a child class can override a method in its parent. An overridden method is essentially hidden in the parent class, and is not invoked unless the child class uses the super keyword within the overriding method.

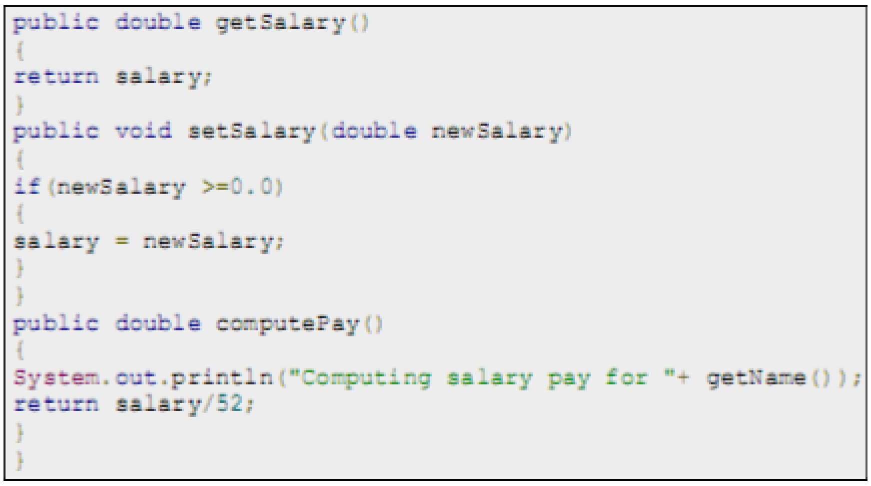


```
/* File name : Employee.java */
public class Employee
{
private String name;
private String address;
private int number;
public Employee(String name,String address,int number)
{
System.out.println("Constructing an Employee");
this.name = name;
this.address = address;
this.number = number;
}
public void mailCheck()
{
System.out.println("Mailing a check to "+this.name
+" "+this.address);
}
public String toString()
{
return name +" "+ address +" "+ number;
}
publicString getName()
{
return name;
}
public String getAddress()
{
return address;
}
public void setAddress(String newAddress)
{
address = newAddress;
}
public int getNumber()
{
return number;
}
}
```

Now suppose we extend Employee class as follows:



```
/* File name : Salary.java */
public class Salaryextends Employee
{
private double salary;//Annual salary
public Salary(String name,String address,int number,double

      salary)
{
super(name, address, number);
setSalary(salary);
}
public void mailCheck()
{
System.out.println("Within mailCheck of Salary class ");
System.out.println("Mailing check to "+ getName()
+" with salary "+ salary);
}
```



```
public double getSalary()
{
return salary;
}
public void setSalary(double newSalary)
{
if(newSalary >=0.0)
{
salary = newSalary;
}
}
public double computePay()
{
System.out.println("Computing salary pay for "+ getName());
return salary/52;
}
}
```

**Method overriding**

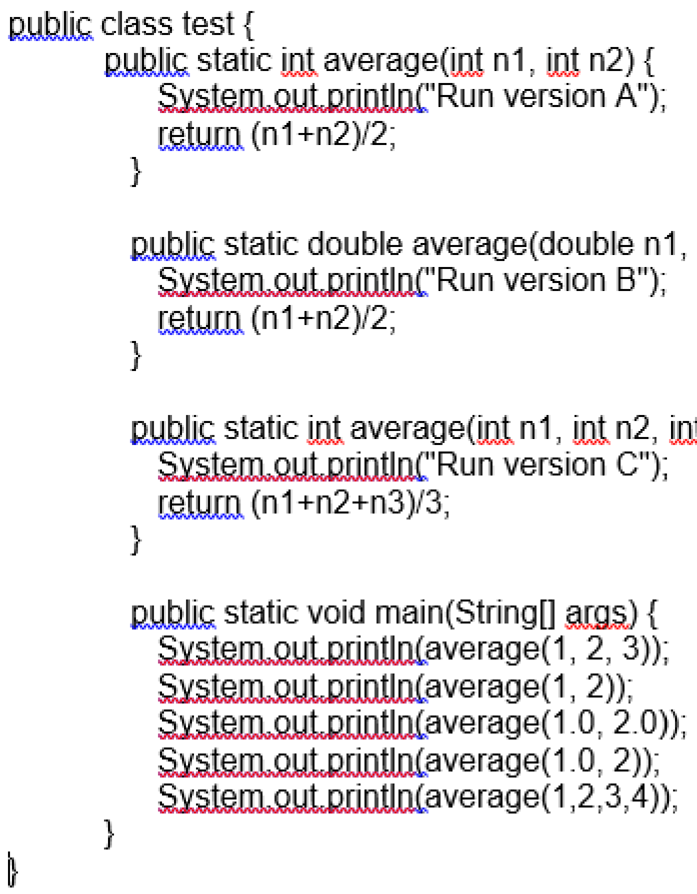
Method mailCheck() in child class (Salary) is overriding the one in parent class (Employee).

Whenever an object is bound with the functionality at run time, this is known as runtime polymorphism. The runtime polymorphism can be achieved by method overriding. Java virtual machine determines the proper method to call at the runtime, not at the compile time. It is also called dynamic or late binding. Method overriding says child class has the same method as declared in the parent class. It means if child class provides the specific implementation of the method that has been provided by one of its parent class, then it is known as method overriding.

**Method overloading**

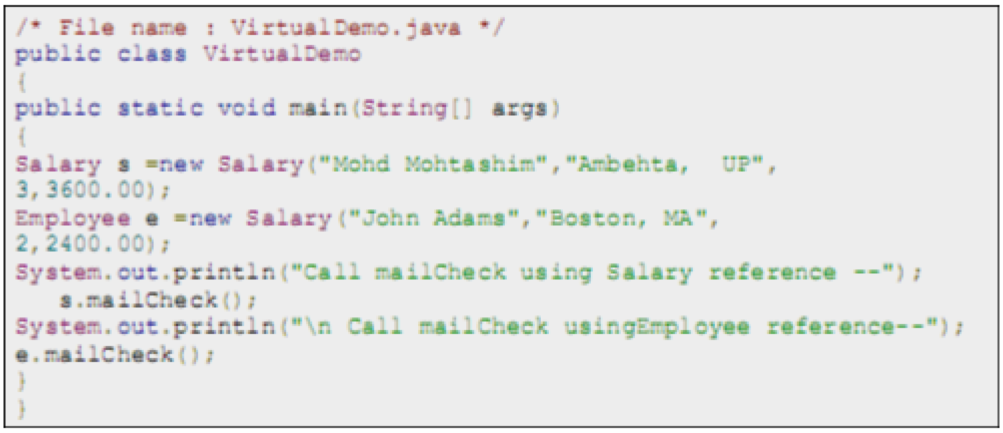
Whenever an object is bound with their functionality at the compile-time, this is known as the compile-time polymorphism. At compile-time, java knows which method to call by checking the method signatures. Therefore, this is called compile-time polymorphism or static or early binding. Compile-time polymorphism is achieved through method overloading.

Method overloading is the same method name but with different number of parameters. Java will match the correct method to use according to the parameters. In case of no match, an error is generated.



```
public class test {
        public static int average(int n1, int n2) {
            System.out.println("Run version A");
            return (n1+n2)/2;
          }

          public static double average(double n1, double n2) {
            System.out.println("Run version B");
            return (n1+n2)/2;
          }

          public static int average(int n1, int n2, int n3) {
            System.out.println("Run version C");
            return (n1+n2+n3)/3;
          }

          public static void main(String[] args) {
            System.out.println(average(1, 2, 3));
            System.out.println(average(1, 2));
            System.out.println(average(1.0, 2.0));
            System.out.println(average(1.0, 2));
            System.out.println(average(1,2,3,4));
        }
}
```

Now, you study the following program carefully and try to determine its output:



```
/* File name : VirtualDemo.java */
public class VirtualDemo
{
public static void main(String[] args)
{
Salary s =new Salary("Mohd Mohtashim","Ambehta,  UP",
3,3600.00);
Employee e =new Salary("John Adams","Boston, MA",
2,2400.00);
System.out.println("Call mailCheck using Salary reference --");
   s.mailCheck();
System.out.println("\n Call mailCheck usingEmployee reference--");
e.mailCheck();
}
}
```

This would produce the following result:

```
Constructing an Employee
Constructing an Employee
Call mailCheck using Salary reference --
Within mailCheck of Salary class
Mailing check to MohdMohtashim with salary 3600.0

Call mailCheck using Employee reference--
Within mailCheck of Salary class
```

**Mailing check to JohnAdams with salary 2400.0**

## 12.5Summary

In this Unit, we discussed:

- Static binding—the binding of variables to operations at compile time;
- Dynamic binding—the binding of variables to operations at run time;
- Operation overloading—the ability to use the same name for two or more methods in a class with difference number of parameters
- Polymorphism—the ability of different objects to perform the appropriate method in response to the same message.

## 12.6Activities

1. Fill in the blanks in each of the following statements:

   a) If a class contains at least one abstract method, it's a(n) ____________class.

   b) Classes from which objects can be instantiated are called ____________ classes.

   c) ___________ involves using a superclass variable to invoke methods on superclass and sub-class objects, enabling you to "program in the general."

   d) Methods that are not interface methods and that do not provide implementations must be declared using keyword ______________.

2. State whether each of the statements that follows is true or false. If false, explain why.

   a) All methods in an abstract class must be declared as abstract methods.

   b) Invoking a subclass-only method through a subclass variable is not allowed.

   c) If a superclass declares an abstract method, a subclass must implement that method.

   d) An object of a class that implements an interface may be thought of as an object of that interface type.

5. How does polymorphism enable you to program "in the general" rather than "in the specific"? Discuss the key advantages of programming "in the general."

6. What are abstract methods? Describe the circumstances in which an abstract method would be appropriate.

7. How does polymorphism promote extensibility?

8. Discuss four ways in which you can assign superclass and subclass references to variables of superclass and subclass types.

# UNIT 13 Interfaces

## 13.1Introduction

An interface is a collection of abstract methods. A class implements an interface, thereby inheriting the abstract methods of the interface.

An interface is not a class. Writing an interface is similar to writing a class, but they are two different concepts. A class describes the attributes and behaviors of an object. An interface contains behaviors that a class implements. Unless the class that implements the interface is abstract, all the methods of the interface need to be defined in the class.

## 13.2 Unit Objectives

The objectives of this unit are to:

- Understand the differences between interfaces and classes
- Learn the declaration and implementation of interfaces
- Extend single or multiple interfaces

## 13.3 Purpose of interfaces

- **Provides communication** − One of the uses of the interface is to provide communication. Through interface, you can specify how you want the methods and fields of a particular type.

- **Multiple inheritance** − Java doesn't support multiple inheritance, using interfaces you can achieve multiple inheritance.

The purpose of interfaces is to allow the programmer to enforce these properties and to know that an object of TYPE T (whatever the interface is) must have functions called X,Y,Z, etc.

Taking an example of button click event, whenever you click a button within your application, a method named actionPerformed is called.

So for you to handle these click events, you need to define this method.

This method is declared in the ActionListener interface and you have to implement this interface to be able to define that method.

JVM already knows that actionPerformed is the method which it will call whenever a button is clicked and now it's your responsibility to define that method.

### 13.4 Interfaces Vs Classes

An interface is similar to a class in the following ways:

- An interface can contain any number of methods.
- An interface is written in a file with a .java extension, with the name of the interface matching the name of the file.
- The bytecode of an interface appears in a .class file.
- Interfaces appear in packages, and their corresponding bytecode file must be in a directory structure that matches the package name.

However, an interface is different from a class in several ways, including:

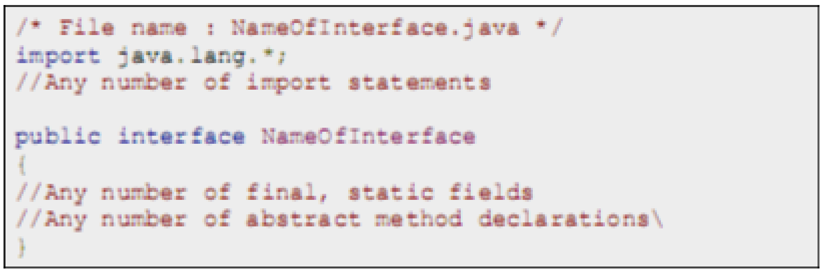
- You cannot instantiate an interface.
- An interface does not contain any constructors.
- All of the methods in an interface are abstract.
- An interface cannot contain instance fields. The only fields that can appear in an interface must be declared both static and final.
- An interface is not extended by a class; it is implemented by a class.
- An interface can extend multiple interfaces.

### 13.5 Declaring Interfaces:

The interface keyword is used to declare an interface. Here is a simple example to declare an interface:

**Example:**

Let us look at an example that depicts encapsulation:

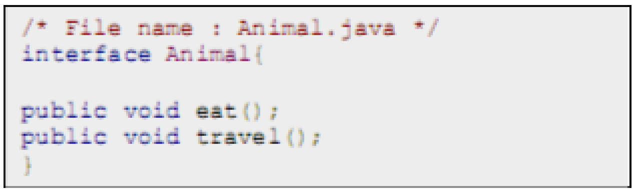


```
/* File name : NameOfInterface.java */
import java.lang.*;
//Any number of import statements

public interface NameOfInterface
{
//Any number of final, static fields
//Any number of abstract method declarations\
}
```

Interfaces have the following properties:

- An interface is implicitly abstract. You do not need to use the abstract keyword when declaring an interface
- Each method in an interface is also implicitly abstract, so the abstract keyword is not needed.
- Methods in an interface are implicitly public.

**Example:**



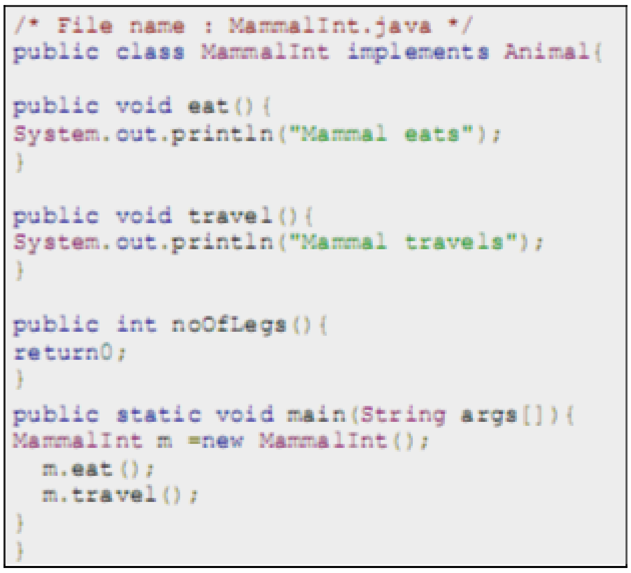
```
/* File name : Animal.java */
interface Animal{

public void eat();
public void travel();
}
```

## 13.6 Implementing Interfaces:

When a class implements an interface, you can think of the class as signing a contract, agreeing to perform the specific behaviours of the interface. If a class does not perform all the behaviours of the interface, the class must declare itself as abstract.

A class that is declared as abstract is known as abstract class. It needs to be extended and its method implemented. It cannot be instantiated.

A class uses the implements keyword to implement an interface. The implements keyword appears in the class declaration following the extends portion of the declaration.



```
/* File name : MammalInt.java */
public class MammalInt implements Animal{

public void eat(){
System.out.println("Mammal eats");
}

public void travel(){
System.out.println("Mammal travels");
}

public int noOfLegs(){
return0;
}
public static void main(String args[]){
MammalInt m =new MammalInt();
  m.eat();
  m.travel();
}
}
```

This would produce the following result:

**Mammal eats**

**Mammal travels**

When overriding methods defined in interfaces, there are several rules to be followed:

- Checked exceptions should not be declared on implementation methods other than the ones declared by the interface method or subclasses of those declared by the interface method.
- The signature of the interface method and the same return type or subtype should be maintained when overriding the methods.
- An implementation class itself can be abstract and if so interface methods need not be implemented.

When implementation interfaces there are several rules:

- A class can implement more than one interface at a time.
- A class can extend only one class, but implement many interfaces.
- An interface can extend another interface, similarly to the way that a class can extend another class.

## 13.7 Extending Interfaces:

An interface can extend another interface, similarly to the way that a class can extend another class. The extends keyword is used to extend an interface, and the child interface inherits the methods of the parent interface.

The following Sports interface is extended by Hockey and Football interfaces:

```
//Filename: Sports.java
public interface Sports
{
public void setHomeTeam(String name);
public void setVisitingTeam(String name);
}

//Filename: Football.java
public interface Football extends Sports
{
public void homeTeamScored(int points);
public void visitingTeamScored(int points);
public void endOfQuarter(int quarter);
}

//Filename: Hockey.java
public interface Hockey extends Sports
{
public void homeGoalScored();
public void visitingGoalScored();

public void endOfPeriod(int period);
public void overtimePeriod(int ot);
}
```

The Hockey interface has four methods, but it inherits two from Sports; thus, a class that implements Hockey needs to implement all six methods. Similarly, a class that implements Football needs to define the three methods from Football and the two methods from Sports.

### 13.8 Extending Multiple Interfaces:

A Java class can only extend one parent class. Multiple inheritance is not allowed. Interfaces are not classes, however, and an interface can extend more than one parent interface.

The extends keyword is used once, and the parent interfaces are declared in a comma-separated list. For example, if the Hockey interface extended both Sports and Event, it would be declared as:

```
public interface Hockey extends Sports,Event
```

### 13.9 Summary

• An interface specifies what operations are allowed but not how they are performed.

• A Java interface describes a set of methods that can be called on an object.

• All interface members must be public, and interfaces may not specify any implementation details, such as concrete method declarations and instance variables.

• All methods declared in an interface are implicitly public abstract methods and all fields are implicitly public, static and final.

• To use an interface, a concrete class must specify that it implements the interface and must declare each interface method with the signature specified in the interface declaration. A class that does not implement all the interface's methods must be declared abstract.

• You can create an interface that describes the desired functionality, then implement the interface in any classes that require that functionality.

•Like public abstract classes, interfaces are typically public types, so they're normally declared in files by themselves with the same name as the interface and the .java file-name extension.

• Java does not allow subclasses to inherit from more than one superclass, but it does allow a class to inherit from a superclass and implement more than one interface.

• All objects of a class that implement multiple interfaces have the is-a relationship with each implemented interface type.

• An interface can declare constants. The constants are implicitly public, static and final.

## 13.10 Activities

1. **Implementing a Queue**

   A queue is an abstract data type for adding and removing elements. The first element added to a queue is the first element that is removed (first-in-first-out, FIFO). Queues can be used, for instance, to manage processes of an operating system: the first process added to the waiting queue is reactivated prior to all other processes (with the same priority). Design an interface Queue, with methods to add and remove elements (integers). Furthermore, a method to check whether the queue is empty or not should exist. Implement the queue with an array.

# UNIT 14 Exception Handling

## 14.1 Introduction

An exception is a problem that arises during the execution of a program. An exception can occur for many different reasons, including the following:

- A user has entered invalid data.
- A file that needs to be opened cannot be found.
- A network connection has been lost in the middle of communications or the JVM has run out of memory.

Some of these exceptions are caused by user error, others by programmer error, and others by physical resources that have failed in some manner.

## 14.2 Unit Objectives

The objectives of this unit are to:

- What exceptions are and how they are handled.
- When to use exception handling.
- To use try blocks to delimit code in which exceptions might occur.
- To throw exceptions to indicate a problem.
- To use catch blocks to specify exception handlers.
- To use the finally block to release resources.
- The exception class hierarchy.
- To create user-defined exceptions.

## 14.3 Types of Exceptions

To understand how exception handling works in Java, you need to understand the three categories of exceptions:

- **Checked exceptions**: A checked exception is an exception that is typically a user error or a problem that cannot be foreseen by the programmer. For example, if a file is to be opened, but the file cannot be found, an exception occurs. These exceptions cannot simply be ignored at the time of compilation.

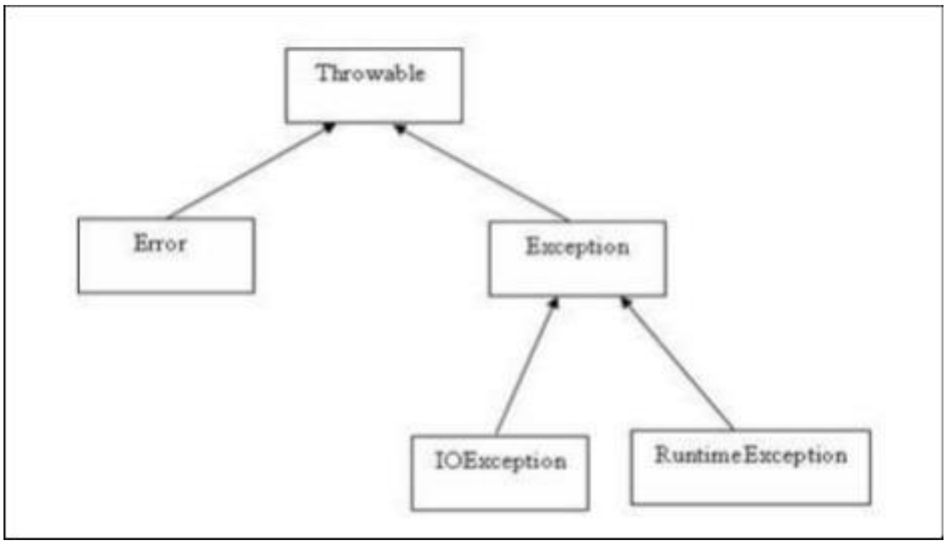
- **Runtime exceptions**: A runtime exception is an exception that occurs that probably could have been avoided by the programmer. As opposed to checked exceptions, runtime exceptions are ignored at the time of compilation.
- **Errors**: These are not exceptions at all, but problems that arise beyond the control of the user or the programmer. Errors are typically ignored in your code because you can rarely do anything about an error. For example, if a stack overflow occurs, an error will arise. They are also ignored at the time of compilation.

## 14.4 Exception Hierarchy:

All exception classes are subtypes of the java.lang.Exception class. The exception class is a subclass of the Throwable class. Other than the exception class, there is another subclass called Error which is derived from the Throwable class.

Errors are not normally trapped form the Java programs. These conditions normally happen in case of severe failures, which are not handled by the java programs. Errors are generated to indicate errors generated by the runtime environment. Example: JVM is out of Memory. Normally programs cannot recover from errors.

The Exception class has two main subclasses: IOException class and RuntimeException Class.



## 14.5 Java's Built-in Exceptions

Java defines several exception classes inside the standard package java.lang.

The most general of these exceptions are subclasses of the standard type RuntimeException. Since java.lang is implicitly imported into all Java programs, most exceptions derived from RuntimeException are automatically available.

Java defines several other types of exceptions that relate to its various class libraries. Following is the list of Java Unchecked RuntimeException.

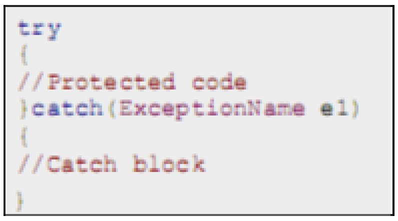
| **Exception** | **Description** |
|---|---|
| ArithmeticException | Arithmetic error, such as divide-by-zero. |
| ArrayIndexOutOfBoundsException | Array index is out-of-bounds. |
| ArrayStoreException | Assignment to an array element of an incompatible type. |
| ClassCastException | Invalid cast. |
| IllegalArgumentException | Illegal argument used to invoke a method. |
| IllegalMonitorStateException | Illegal monitor operation, such as waiting on an unlocked thread. |
| IllegalStateException | Environment or application is in incorrect state. |
| IllegalThreadStateException | Requested operation not compatible with current thread state. |
| IndexOutOfBoundsException | Some type of index is out-of-bounds. |
| NegativeArraySizeException | Array created with a negative size. |
| NullPointerException | Invalid use of a null reference. |
| NumberFormatException | Invalid conversion of a string to a numeric format. |
| SecurityException | Attempt to violate security. |
| StringIndexOutOfBounds | Attempt to index outside the bounds of a string. |
| UnsupportedOperationException | An unsupported operation was encountered. |

Following is the list of Java Checked Exceptions Defined in java.lang.

| **Exception** | **Description** |
|---|---|
| ClassNotFoundException | Class not found. |
| CloneNotSupportedException | Attempt to clone an object that does not implement the Cloneable interface. |
| IllegalAccessException | Access to a class is denied. |
| InstantiationException | Attempt to create an object of an abstract class or interface. |
| InterruptedException | One thread has been interrupted by another thread. |
| NoSuchFieldException | A requested field does not exist. |
| NoSuchMethodException | A requested method does not exist. |

## 14.6 Catching Exceptions:

A method catches an exception using a combination of the try and catch keywords. A try/catch block is placed around the code that might generate an exception. Code within a try/catch block is referred to as protected code, and the syntax for using try/catch looks like the following:

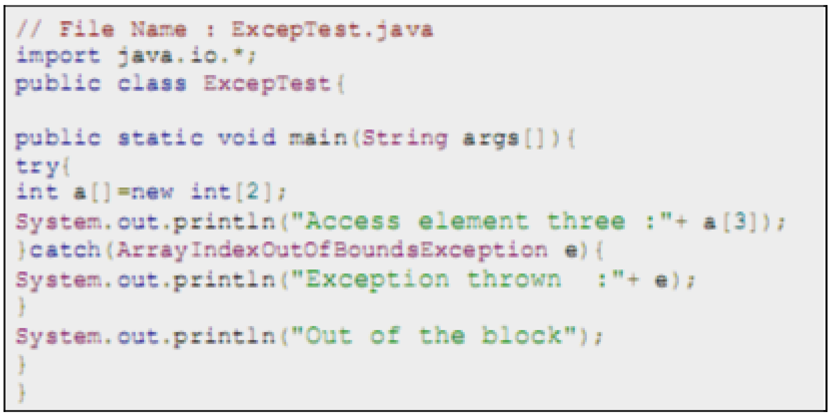


```
try
{
//Protected code
}catch(ExceptionName e1)
{
//Catch block
}
```

A catch statement involves declaring the type of exception you are trying to catch. If an exception occurs in protected code, the catch block (or blocks) that follow the try is checked. If the type of exception that occurred is listed in a catch block, the exception is passed to the catch block much as an argument is passed into a method parameter.

**Example:**

The following is an array is declared with 2 elements. Then, the code tries to access the 3rd element of the array which throws an exception:



```
// File Name : ExcepTest.java
import java.io.*;
public class ExcepTest{

public static void main(String args[]){
try{
int a[]=new int[2];
System.out.println("Access element three :"+ a[3]);
}catch(ArrayIndexOutOfBoundsException e){
System.out.println("Exception thrown  :"+ e);
}
System.out.println("Out of the block");
}
}
```

This would produce the following result:

**Exception thrown: java.lang.ArrayIndexOutOfBoundsException:3**
**Out of the block**

## 14.7 Multiple catch Blocks:

A try block can be followed by multiple catch blocks. The syntax for multiple catch blocks looks like the following:

```
try
{
//Protected code
}catch(ExceptionType1 e1)
{
//Catch block
}catch(ExceptionType2 e2)
{
//Catch block
}catch(ExceptionType3 e3)
{
//Catch block
}
```

The previous statements demonstrate three catch blocks, but you can have any number of them after a single try. If an exception occurs in the protected code, the exception is thrown to the first catch block in the list. If the data type of the exception thrown matches ExceptionType1, it gets caught there. If not, the exception passes down to the second catch statement. This continues until the exception either is caught or falls through all catches, in which case the current method stops execution and the exception is thrown down to the previous method on the call stack.

**Example:**

Here is code segment showing how to use multiple try/catch statements.

```
try
{
   file =newFileInputStream(fileName);
   x =(byte) file.read();
}catch(IOException i)
{
   i.printStackTrace();
return-1;
}catch(FileNotFoundException f)//Not valid!
{
   f.printStackTrace();
return-1;
}
```

## 14.8 The throws/throw Keywords:

If a method does not handle a checked exception, the method must declare it using the throws keyword. The throws keyword appears at the end of a method's signature. You can throw an exception, either a newly instantiated one or an exception that you just caught, by using the throw keyword. Try to understand the different in throws and throw keywords.

The following method declares that it throws a RemoteException:

```
import java.io.*;
public class className
{
public  void deposit(double amount)throws RemoteException
{
// Method implementation
throw new RemoteException();
}
//Remainder of class definition
}
```

A method can declare that it throws more than one exception, in which case the exceptions are declared in a list separated by commas. For example, the following method declares that it throws a RemoteException and an InsufficientFundsException:

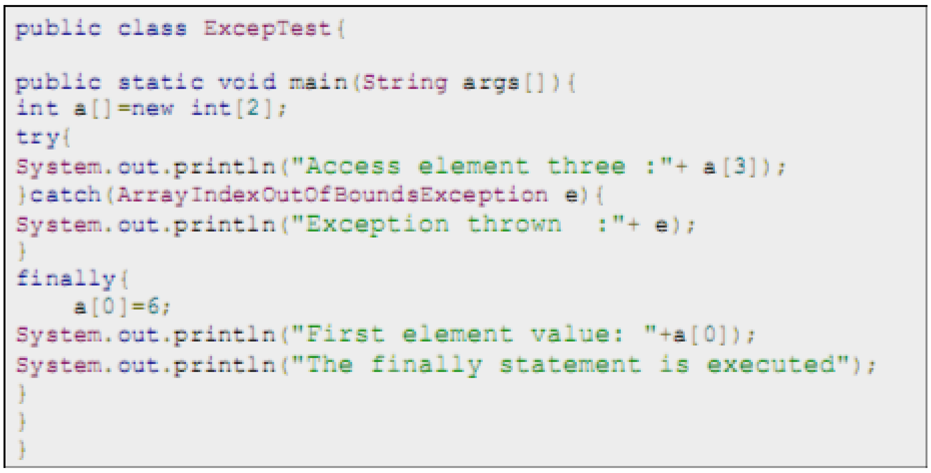
```
import java.io.*;
public class className
{
public void withdraw(double amount)throws RemoteException,
InsufficientFundsException
{
// Method implementation
}
//Remainder of class definition
}
```

### 14.9 The finally Keyword

The finally keyword is used to create a block of code that follows a try block. A finally block of code always executes, whether or not an exception has occurred. Using a finally block allows you to run any cleanup-type statements that you want to execute, no matter what happens in the protected code. A finally block appears at the end of the catch blocks and has the following syntax:

```
try
{
//Protected code
}catch(ExceptionType1 e1)
{
//Catch block
}catch(ExceptionType2 e2)
{
//Catch block
}catch(ExceptionType3 e3)
{
//Catch block
}finally
{
//The finally block always executes.
}
```

**Example:**



```
public class ExcepTest{

public static void main(String args[]){
int a[]=new int[2];
try{
System.out.println("Access element three :"+ a[3]);
}catch(ArrayIndexOutOfBoundsException e){
System.out.println("Exception thrown  :"+ e);
}
finally{
    a[0]=6;
System.out.println("First element value: "+a[0]);
System.out.println("The finally statement is executed");
}
}
}
```

This would produce the following result:

```
Exception thrown :java.lang.ArrayIndexOutOfBoundsException:3
First element value:6
The finally statement is executed
```

## 14.10 Summary

• A try block encloses code that might throw an exception and code that should not execute if that exception occurs.

• Exceptions may surface through explicitly mentioned code in a try block, through calls to other methods or even through deeply nested method calls initiated by code in the try block.

• A catch block begins with keyword catch and an exception parameter followed by a block of code that handles the exception. This code executes when the try block detects the exception.

• An uncaught exception is an exception that occurs for which there are no matching catch blocks.

• An uncaught exception will cause a program to terminate early if that program contains only one thread. Otherwise, only the thread where the exception occurred will terminate. The rest of the program will run but possibly with adverse results.

• At least one catch block or a finally block must immediately follow the try block.

•A catch block specifies in parentheses an exception parameter identifying the exception type to handle. The parameter's name enables the catch block to interact with a caught exception object.

• If an exception occurs in a try block, the try block terminates immediately and program control transfers to the first catch block with a parameter type that matches the thrown exception's type.

• After an exception is handled, program control does not return to the throw point, because the try block has expired. This is known as the termination model of exception handling.

• If there are multiple matching catch blocks when an exception occurs, only the first is executed.

•A throws clause specifies a comma-separated list of exceptions that the method might throw, and appears after the method's parameter list and before the method body.

•The finally block is optional. If it's present, it's placed after the last catch block.

•The finally block will execute whether or not an exception is thrown in the corresponding try block or any of its corresponding catch blocks.

## 14.11 Activities

1. List five common examples of exceptions.
2. Give several reasons why exception-handling techniques should not be used for conventional program control.
3. Why are exceptions particularly appropriate for dealing with errors produced by methods of classes in the Java API?
4. What is a "resource leak"?
5. If no exceptions are thrown in a try block, where does control proceed to when the try block completes execution?
6. Give a key advantage of using catch (Exception exceptionName ).
7. Should a conventional application catch Error objects? Explain.
8. What happens if no catch handler matches the type of a thrown object?
9. What happens if several catch blocks match the type of the thrown object?

10. Why would a programmer specify a superclass type as the type in a catch block?

11. What is the key reason for using finally blocks?

12. What happens when a catch block throws an Exception?

13. What does the statement throw exception Reference do in a catch block?

14. What happens to a local reference in a try block when that block throws an Exception?

15. Write a Java program that divides two user-inputted integers that catches the exception when an integer is divided by zero.

# UNIT 15 Serialisation

## 15.1 Introduction

Java provides a mechanism, called object serialization where an object can be represented as a sequence of bytes that includes the object's data as well as information about the object's type and the types of data stored in the object.

After a serialized object has been written into a file, it can be read from the file and deserialized that is, the type information and bytes that represent the object and its data can be used to recreate the object in memory.

Most impressive is that the entire process is JVM independent, meaning an object can be serialized on one platform and deserialized on an entirely different platform.

## 15.2 Unit Objectives

The objectives of this unit are to:

- learn the purpose of serialization.
- deploy the writeObject and readObject functions for objects.
- serialize and deserialize objects using objectInputStream and objectOutputStream.
- relate errors associated with the serialization process.

## 15.3 ObjectInputStream and ObjectOutputStream

Classes ObjectInputStream and ObjectOutputStream are high-level streams that contain the methods for serializing and deserializing an object.

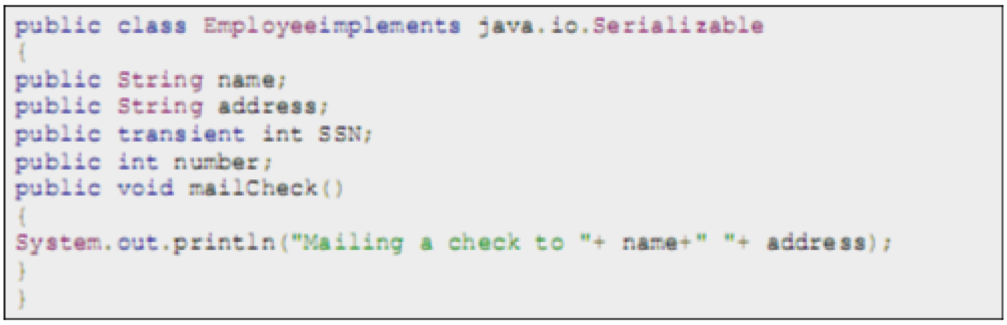
The ObjectOutputStream class contains many write methods for writing various data types, but one method in particular stands out:

*public final void writeObject(Object x)throws IOException*

The above method serializes an Object and sends it to the output stream. Similarly, the ObjectInputStream class contains the following method for deserializing an object:

*public final Object readObject()throws IOException, ClassNotFoundException*

This method retrieves the next Object out of the stream and deserializes it. The return value is Object, so you will need to cast it to its appropriate data type. To demonstrate how serialization works in Java, I am going to use the Employee class that we discussed early on in the book. Suppose that we have the following Employee class, which implements the Serializable interface:



```
public class Employeeimplements java.io.Serializable
{
public String name;
public String address;
public transient int SSN;
public int number;
public void mailCheck()
{
System.out.println("Mailing a check to "+ name+" "+ address);
}
}
```

Notice that for a class to be serialized successfully, two conditions must be met:

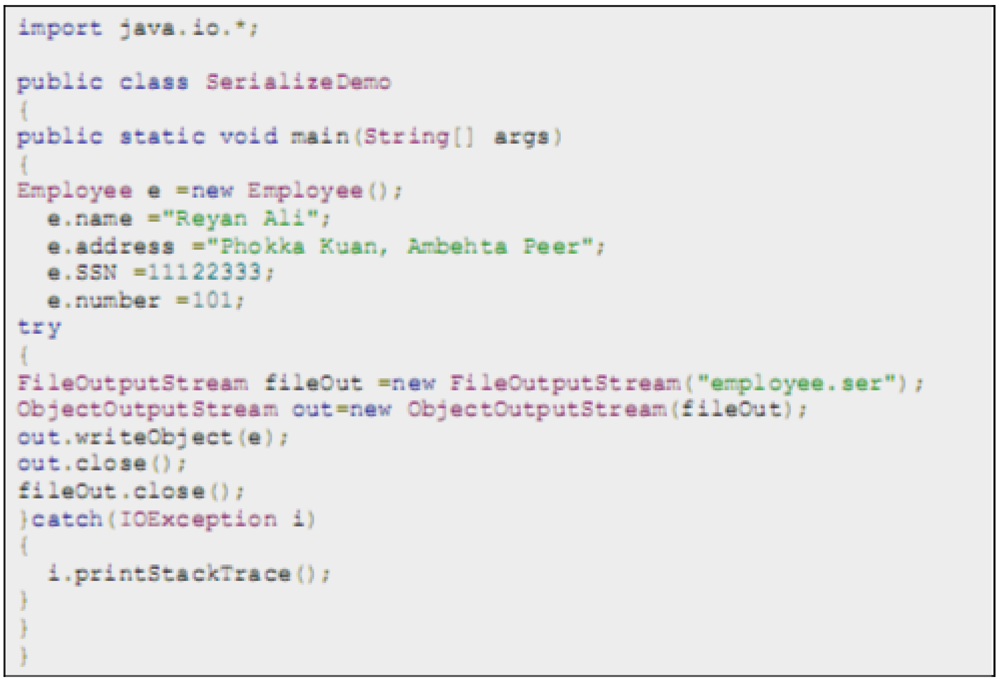
- The class must implement the java.io.Serializable interface.
- All of the fields in the class must be serializable. If a field is not serializable, it must be marked transient.

## 15.4 Serializing an Object:

The ObjectOutputStream class is used to serialize an Object. The following SerializeDemo program instantiates an Employee object and serializes it to a file.

When the program is done executing, a file named employee.ser is created. The program does not generate any output, but study the code and try to determine what the program is doing.

Note: When serializing an object to a file, the standard convention in Java is to give the file a .ser extension.

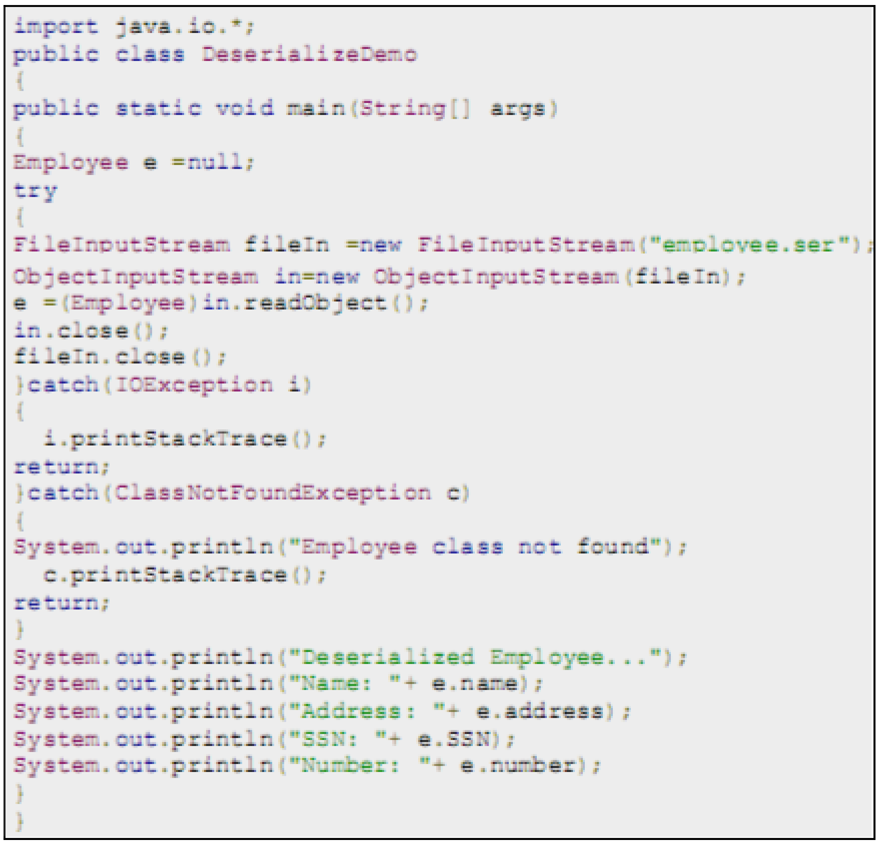


```
import java.io.*;

public class SerializeDemo
{
public static void main(String[] args)
{
Employee e =new Employee();
  e.name ="Reyan Ali";
  e.address ="Phokka Kuan, Ambehta Peer";
  e.SSN =11122333;
  e.number =101;
try
{
FileOutputStream fileOut =new FileOutputStream("employee.ser");
ObjectOutputStream out=new ObjectOutputStream(fileOut);
out.writeObject(e);
out.close();
fileOut.close();
}catch(IOException i)
{
  i.printStackTrace();
}
}
}
```

## 15.5 Deserializing an Object:

The following DeserializeDemo program deserializes the Employee object created in the SerializeDemo program.

Study the program and try to determine its output:



```
import java.io.*;
public class DeserializeDemo
{
public static void main(String[] args)
{
Employee e =null;
try
{
FileInputStream fileIn =new FileInputStream("employee.ser");
ObjectInputStream in=new ObjectInputStream(fileIn);
e =(Employee)in.readObject();
in.close();
fileIn.close();
}catch(IOException i)
{
  i.printStackTrace();
return;
}catch(ClassNotFoundException c)
{
System.out.println("Employee class not found");
  c.printStackTrace();
return;
}
System.out.println("Deserialized Employee...");
System.out.println("Name: "+ e.name);
System.out.println("Address: "+ e.address);
System.out.println("SSN: "+ e.SSN);
System.out.println("Number: "+ e.number);
}
}
```

This would produce the following result:

```
DeserializedEmployee...
Name:ReyanAli
Address:PhokkaKuan,AmbehtaPeer
SSN:0
Number:101
```

Here are following important points to be noted:

- The try/catch block tries to catch a ClassNotFoundException, which is declared by the readObject() method.
- For a JVM to be able to deserialize an object, it must be able to find the bytecode for the class. If the JVM can't find a class during the deserialization of an object, it throws a ClassNotFoundException.
- Notice that the return value of readObject() is cast to an Employee reference.
- The value of the SSN field was 11122333 when the object was serialized, but because the field is transient, this value was not sent to the output stream. The SSN field of the deserialized Employee object is 0.

## 15.6 Summary

• Java provides a mechanism called object serialization that enables entire objects to be written to or read from a stream.

• A serialized object is represented as a sequence of bytes that includes the object's data as well as information about the object's type and the types of data it stores.

• After a serialized object has been written into a file, it can be read from the file and deserialized to recreate the object in memory.

• Classes ObjectInputStream and ObjectOutputStream enable entire objects to be read from or written to a stream (possibly a file).

• Only classes that implement interface Serializable can be serialized and deserialized.

## 15.7 Activities

1. Develop a small class which implements serializable and has (apart from other variables/methods) a String that gets the current time-stamp using the Calendar or Date Object. (Use Java Calendar or Java Date). Serialize it and deserialize it. Is the date stamp from the un-serialized object the same as system date?

2. What is SerialVersionUID in Java?

3. Explain some real-time scenarios where Java Serialization can be used?

4. Is Serializable a Class or Interface?

5. Is it possible to serialize a static variable in the class?

6. What happens when you serialize an object that has references to non-serializable objects?

7. After de-serialization, what will be the value of a transient variable?

8. Objects are de-serialized in the order in which they were serialized – TRUE or FALSE?

9. Can you serialize an object to any other medium other than files?

# UNIT 16 Threads (Multithreading)

## 16.1 Introduction

Java is a multithreaded programming language which means we can develop multithreaded program using Java. A multithreaded program contains two or more parts that can run concurrently and each part can handle different task at the same time making optimal use of the available resources especially when your computer has multiple CPUs.

By definition, multitasking is when multiple processes share common processing resources such as a CPU. Multithreading extends the idea of multitasking into applications where you can subdivide specific operations within a single application into individual threads. Each of the threads can run in parallel. The OS divides processing time not only among different applications, but also among each thread within an application.

Multithreading enables you to write in a way where multiple activities can proceed concurrently in the same program.

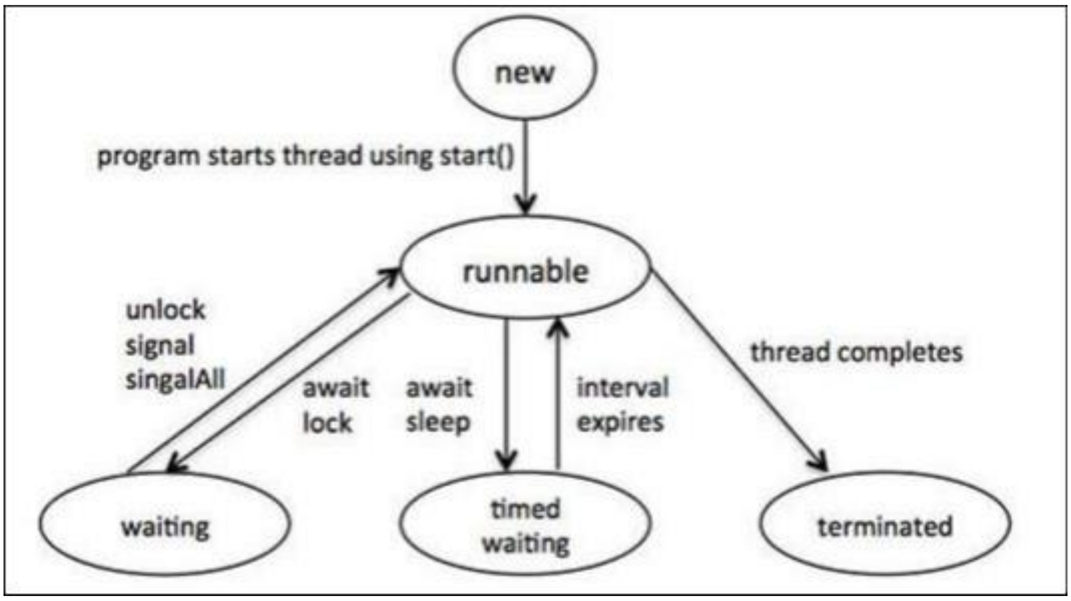
## 16.2 Unit Objectives

The objectives of this unit are to:

- What threads are and why they are useful.
- How threads enable you to manage concurrent activities.
- The life cycle of a thread.
- To create and execute Runnables.
- Thread synchronization.
- To enable multiple threads

## 16.3 Life Cycle of a Thread

A thread goes through various stages in its life cycle. For example, a thread is born, started, runs, and then dies. Following diagram shows complete life cycle of a thread.



Above-mentioned stages are explained here:

- **New**: A new thread begins its life cycle in the new state. It remains in this state until the program starts the thread. It is also referred to as a born thread.
- **Runnable**: After a newly born thread is started, the thread becomes runnable. A thread in this state is considered to be executing its task.
- **Waiting**: Sometimes, a thread transitions to the waiting state while the thread waits for another thread to perform a task. A thread transitions back to the runnable state only when another thread signals the waiting thread to continue executing.
- **Timed waiting**: A runnable thread can enter the timed waiting state for a specified interval of time. A thread in this state transition back to the runnable state when that time interval expires or when the event it is waiting for occurs.
- **Terminated:** A runnable thread enters the terminated state when it completes its task or otherwise terminates.

## 16.4 Thread Priorities:

Every Java thread has a priority that helps the operating system determine the order in which threads are scheduled. Java thread priorities are in the range between MIN_PRIORITY (a constant of 1) and MAX_PRIORITY (a constant of 10). By default, every thread is given priority NORM_PRIORITY (a constant of 5).

Threads with higher priority are more important to a program and should be allocated processor time before lower-priority threads. However, thread priorities cannot guarantee the order in which threads execute and very much platform dependendent.

## 16.5 Create Thread by Implementing Runnable Interface:

If your class is intended to be executed as a thread then you can achieve this by implementingRunnable interface.

You will need to follow three basic steps:

**STEP 1:**

As a first step you need to implement a run() method provided by Runnable interface. This method provides entry point for the thread and you will put you complete business logic inside this method. Following is simple syntax of run() method:

*public void run( )*

**STEP 2:**

At second step you will instantiate a Thread object using the following constructor:

*Thread(Runnable threadObj, String threadName);*

Where, threadObj is an instance of a class that implements the Runnable interface and threadName is the name given to the new thread.

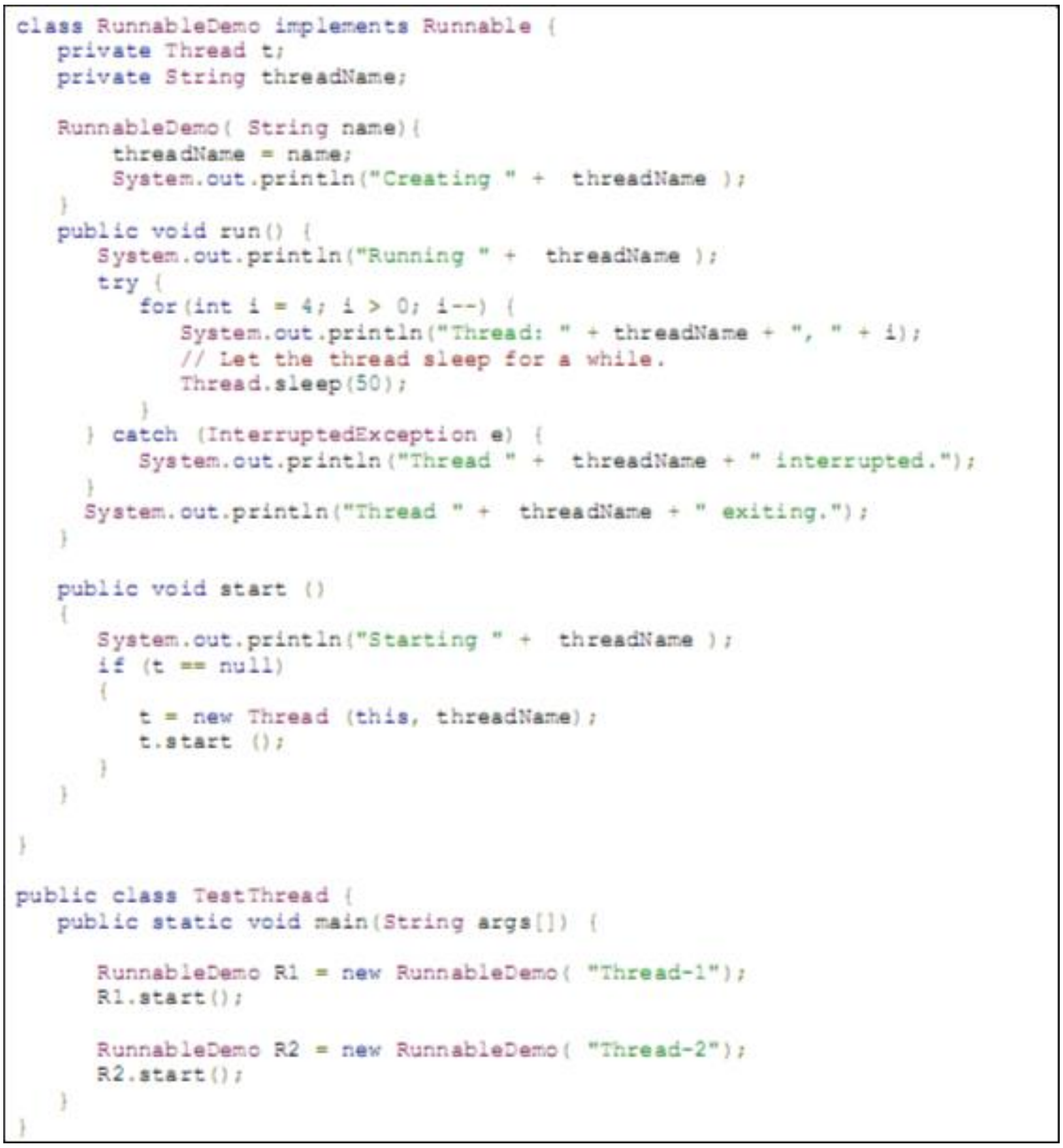
**STEP 3**

Once Thread object is created, you can start it by calling start( ) method, which executes a call to run( ) method. Following is simple syntax of start() method:

*void start( );*

**Example:**

Here is an example that creates a new thread and starts it running:



```
class RunnableDemo implements Runnable {
   private Thread t;
   private String threadName;

   RunnableDemo( String name){
       threadName = name;
       System.out.println("Creating " +  threadName );
   }
   public void run() {
      System.out.println("Running " +  threadName );
      try {
         for(int i = 4; i > 0; i--) {
            System.out.println("Thread: " + threadName + ", " + i);
            // Let the thread sleep for a while.
            Thread.sleep(50);
         }
     } catch (InterruptedException e) {
         System.out.println("Thread " +  threadName + " interrupted.");
     }
     System.out.println("Thread " +  threadName + " exiting.");
   }

   public void start ()
   {
      System.out.println("Starting " +  threadName );
      if (t == null)
      {
         t = new Thread (this, threadName);
         t.start ();
      }
   }

}

public class TestThread {
   public static void main(String args[]) {

      RunnableDemo R1 = new RunnableDemo( "Thread-1");
      R1.start();

      RunnableDemo R2 = new RunnableDemo( "Thread-2");
      R2.start();
   }
}
```

This would produce the following result:

```
Creating Thread-1
Starting Thread-1
Creating Thread-2
Starting Thread-2
Running Thread-1
Thread: Thread-1, 4
Running Thread-2
Thread: Thread-2, 4
Thread: Thread-1, 3
Thread: Thread-2, 3
Thread: Thread-1, 2
Thread: Thread-2, 2
Thread: Thread-1, 1
Thread: Thread-2, 1
Thread Thread-1 exiting.
Thread Thread-2 exiting.
```

## 16.6 Create Thread by Extending Thread Class:

The second way to create a thread is to create a new class that extends Thread class using the following two simple steps. This approach provides more flexibility in handling multiple threads created using available methods in Thread class.

**STEP 1**

You will need to override run( ) method available in Thread class. This method provides entry point for the thread and you will put you complete business logic inside this method. Following is simple syntax of run() method:

*public void run( )*

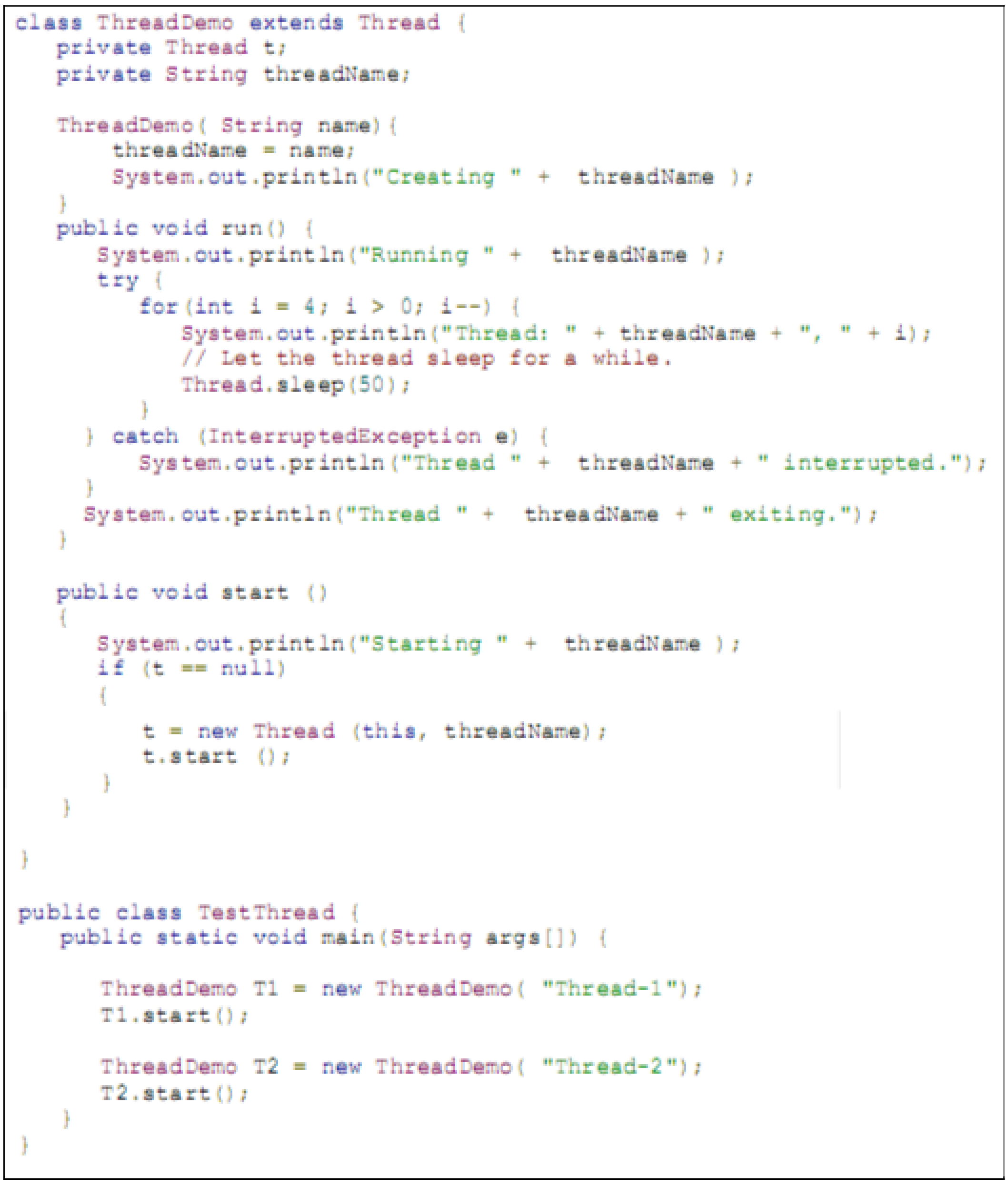
**STEP 2**

Once Thread object is created, you can start it by calling start( ) method, which executes a call to run( ) method. Following is simple syntax of start() method:

*void start( );*

**Example:**

Here is the preceding program rewritten to extend Thread:



```
class ThreadDemo extends Thread {
   private Thread t;
   private String threadName;

   ThreadDemo( String name){
       threadName = name;
       System.out.println("Creating " +  threadName );
   }
   public void run() {
      System.out.println("Running " +  threadName );
      try {
         for(int i = 4; i > 0; i--) {
            System.out.println("Thread: " + threadName + ", " + i);
            // Let the thread sleep for a while.
            Thread.sleep(50);
         }
     } catch (InterruptedException e) {
         System.out.println("Thread " +  threadName + " interrupted.");
     }
     System.out.println("Thread " +  threadName + " exiting.");
   }

   public void start ()
   {
      System.out.println("Starting " +  threadName );
      if (t == null)
      {
         t = new Thread (this, threadName);
         t.start ();
      }
   }

}

public class TestThread {
   public static void main(String args[]) {

      ThreadDemo T1 = new ThreadDemo( "Thread-1");
      T1.start();

      ThreadDemo T2 = new ThreadDemo( "Thread-2");
      T2.start();
   }
}
```

This would produce the following result:

```
Creating Thread-1
Starting Thread-1
Creating Thread-2
Starting Thread-2
Running Thread-1
Thread: Thread-1, 4
```

```
Running Thread-2
Thread: Thread-2, 4
Thread: Thread-1, 3
Thread: Thread-2, 3
Thread: Thread-1, 2
Thread: Thread-2, 2
Thread: Thread-1, 1
Thread: Thread-2, 1
Thread Thread-1 exiting.
Thread Thread-2 exiting.
```

## 16.7 Thread Synchronization

If one thread tries to read the data and other thread tries to update the same date, it leads to inconsistent state. This can be prevented by synchronising access to data.

In Java: “Synchronized” method:

```
syncronised void update()
{
	…
}
```

Example

```
/* Account.java: A program with synchronized methods for the Account class.
*/
package com.javabook.threading;

public class Account {
	private double balance = 0;
	public Account(double balance) {
		this.balance = balance;
	}

	// if ‘synchronized’ is removed, the outcome is unpredictable
	public synchronized void deposit(double amount) {
		if (amount < 0) {
			throw new IllegalArgumentException(“Can’t deposit.”);
		}
		this.balance += amount;
		System.out.println(“Deposit ”+amount+“ in thread” +Thread.currentThread().getId()
		+“, balance is ” +balance);
	}

	// if ‘synchronized’ is removed, the outcome is unpredictable
	public synchronized void withdraw(double amount) {
		if (amount < 0 || amount > this.balance) {
			throw new IllegalArgumentException(“Can’t withdraw.”);
		}
		this.balance -= amount;
		System.out.println(“Withdraw ”+amount+“ in thread ” + Thread.currentThread().getId()
		+ “, balance is ”+balance);
	}
}//end Account class
```

## 16.8 Summary

- A Thread is a running instance of a process. A thread can have five states; namely new, ready, execute, wait or terminated.
- Thread scheduling determines which thread to dispatch based on thread priorities.
- Thread are created in Java by either using a runnable interface or by extending thread class.

## 16.9 Activities

1. (Multithreading Terms) Define each of the following terms.

a) thread

b) multithreading

c) runnable state

d) timed waiting state

e) preemptive scheduling

f) Runnable interface

2. (Blocked State) List the reasons for entering the blocked state. For each of these, describe how the program will normally leave the blocked state and enter the runnable state.

3. Write a Java program that runs two loops from 1 to 1,000,000. Check the time taken for the two loops to run for a normal program and compare it to the time taken for the same program to run using multi-threading.

**END OF MANUAL**