INTRODUCTION TO OOP AND JAVA FUNDAMENTALS

1.1 OBJECT-ORIENTED PROGRAMMING

Object-oriented programming (OOP) is a programming paradigm based on the concept of “objects”, which may contain data, in the form of fields, often known as attributes; and code, in the form of procedures, often known as methods.

List of object-oriented programming languages

Ada 95	Fortran 2003	PHP since v4, greatly enhanced in v5
BETA	Graphtalk	Python
C++	IDLscript	Ruby
C#	J#	Scala
COBOL	Java	Simula
Cobra	LISP	Smalltalk
ColdFusion	Objective-C	Tcl
Common Lisp	Perl since v5	

Abstraction

Abstraction is one of the key concepts of object-oriented programming (OOP) languages. Its main goal is to handle complexity by hiding unnecessary details from the user. This enables the user to implement more complex logic on top of the provided abstraction without understanding about all the hidden complexity.

For example, people do not think of a car as a set of tens of thousands of individual parts. They think of it as a well-defined object with its own unique behavior. This abstraction allows people to use a car to drive to the desired location without worrying about the complexity of the parts that form the car. They can ignore the details of how the engine, transmission, and braking systems work. Instead, they are free to utilize the object as a whole.

A powerful way to manage abstraction is through the use of hierarchical classifications. This allows us to layer the semantics of complex systems, breaking them into more manageable pieces.

- Hierarchical abstractions of complex systems can also be applied to computer programs.
- The data from a traditional process-oriented program can be transformed by abstraction into its component objects.
- A sequence of process steps can become a collection of messages between these objects.
- Thus, each of these objects describes its own unique behavior.
- We can treat these objects as concrete entities that respond to messages telling them to do something.

Objects And Classes

Object

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

Class

A class can be defined as a template/blueprint that describes the behavior/state that the object of its type support.

Objects in Java

If we consider the real-world, we can find many objects around us, cars, dogs, humans, etc. All these objects have a state and a behavior.

If we consider a dog, then its state is - name, breed, color, and the behavior is - barking, wagging the tail, running.

If we compare the software object with a real-world object, they have very similar characteristics.

Software objects also have a state and a behavior. A software object’s state is stored in fields and behavior is shown via methods.

So in software development, methods operate on the internal state of an object and the object-to-object communication is done via methods.

Classes in Java

A class is a blueprint from which individual objects are created.

Following is an example of a class.

```
public class Dog {
    String breed;
    int age;
    String color;
    void barking()
    {
    }
}
```

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 initialized when the class is instantiated. 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.

Encapsulation

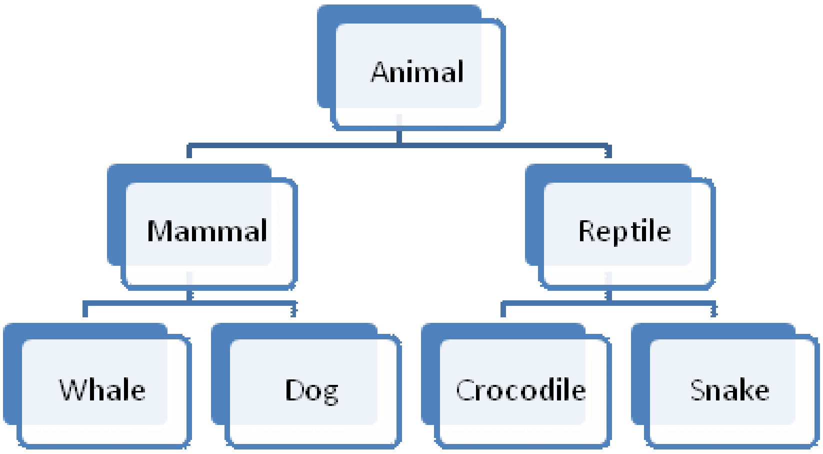
Encapsulation is the mechanism that binds together code and the data it manipulates, and keeps both safe from outside interference and misuse.

- In Java, the basis of encapsulation is the class. There are mechanisms for hiding the complexity of the implementation inside the class.
- Each method or variable in a class may be marked private or public.
- The public interface of a class represents everything that external users of the class need to know, or may know.
- The private methods and data can only be accessed by code that is a member of the class.
- Therefore, any other code that is not a member of the class cannot access a private method or variable.

- Since the private members of a class may only be accessed by other parts of program through the class' public methods, we can ensure that no improper actions take place.

Inheritance

Inheritance is the process by which one object acquires the properties of another object.



For example, a Dog is part of the classification Mammal, which in turn is part of the Animal class. Without the use of hierarchies, each object would need to define all of its characteristics explicitly. However, by use of inheritance, an object need only define those qualities that make it unique within its class. It can inherit its general attributes from its parent. Thus, inheritance makes it possible for one object to be a specific instance of a more general case.

Polymorphism

Polymorphism (from Greek, meaning "many forms") is a feature that allows one interface to be used for a general class of actions. The specific action is determined by the exact nature of the situation.

For eg, a dog's sense of smell is polymorphic. If the dog smells a cat, it will bark and run after it. If the dog smells its food, it will salivate and run to its bowl. The same sense of smell is at work in both situations. The difference is what is being smelled, that is, the type of data being operated upon by the dog's nose.

Consider a stack (which is a last-in, first-out LIFO list). We might have a program that requires three types of stacks. One stack is used for integer values, one for floating-point values, and one for characters. The algorithm that implements each stack is the same, even though the data being stored differs.

1.2 OOP CONCEPTS IN JAVA

OOP concepts in Java are the main ideas behind Java's Object Oriented Programming. They are:

Object

Any entity that has state and behavior is known as an object. It can be either physical or logical.

For example: chair, pen, table, keyboard, bike etc.

Class & Instance

Collection of objects of the same kind is called class. It is a logical entity.

A Class is a 3-Compartment box encapsulating data and operations as shown in figure.

Class Name
Static Attributes
Dynamic Behaviors

Figure: Class Structure

The followings figure shows two classes 'Student' and 'Circle'.

Name (Identifier)	Student	Circle
Variables (Static Attributes)	name, gender, dept, marks	radius, color

Methods
(Dynamic Behaviors)
	getDetails()
	calculateGrade()
		getRadius()
		printArea()

Figure: Examples of classes

A class can be visualized as a three-compartment box, as illustrated:

1. Name (or identity): identifies the class.
2. Variables (or attribute, state, field): contain the static attributes of the class.
3. Methods (or behaviors, function, operation): contain the dynamic behaviors of the class.

An instance is an instantiation of a class. All the instances of a class have similar properties, as described in the class definition. The term "object" usually refers to instance.

For example, we can define a class called "Student" and create three instances of the class "Student" for "John", "Priya" and "Anil".

The following figure shows three instances of the class Student, identified as "John", "Priya" and "Anil".

John : Student	Priya : Student	Anil : Student

name = “John”
gender = “male”
	name = “Priya”
	gender = “female”
		name = “Anil”
		gender = “male”

dept = “CSE”
mark = 88
	gender = “female”
	dept = “IT”
		gender = “male”
		dept = “IT”

getDetails()
calculateGrade()
	getDetails()
	calculateGrade()
		getDetails()
		calculateGrade()

Figure: Instances of a class ‘Student’

Abstraction

Abstraction refers to the quality of dealing with ideas rather than events. It basically deals with hiding the details and showing the essential things to the user.

We all know how to turn the TV on, but we don’t need to know how it works in order to enjoy it.

Abstraction means simple things like objects, classes, and variables represent more complex underlying code and data. It avoids repeating the same work multiple times. In java, we use abstract class and interface to achieve abstraction.

Abstract class:

Abstract class in Java contains the ‘abstract’ keyword. If a class is declared abstract, it cannot be instantiated. So we cannot create an object of an abstract class. Also, an abstract class can contain abstract as well as concrete methods.

To use an abstract class, we have to inherit it from another class where we have to provide implementations for the abstract methods there itself, else it will also become an abstract class.

Interface:

Interface in Java is a collection of abstract methods and static constants. In an interface, each method is public and abstract but it does not contain any constructor. Along with abstraction, interface also helps to achieve multiple inheritance in Java.

So an interface is a group of related methods with empty bodies.

Encapsulation

Binding (or wrapping) code and data together into a single unit is known as encapsulation. It means to hide our data in order to make it safe from any modification.

The best way to understand encapsulation is to look at the example of a medical capsule, where the drug is always safe inside the capsule. Similarly, through encapsulation the methods and variables of a class are well hidden and safe.

A java class is the example of encapsulation.

Encapsulation can be achieved in Java by:

- Declaring the variables of a class as private.
- Providing public setter and getter methods to modify and view the variables values.

Inheritance

This is a special feature of Object Oriented Programming in Java. It lets programmers create new classes that share some of the attributes of existing classes.

For eg, a child inherits the properties from his father.

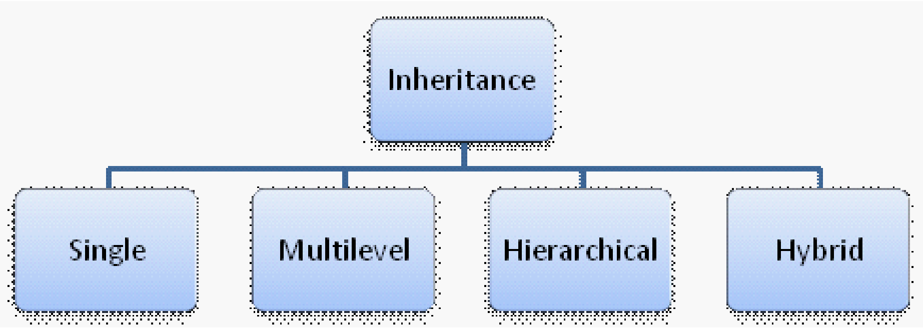
Similarly, in Java, there are two classes:

1. Parent class (Super or Base class)

2. Child class (Subclass or Derived class)

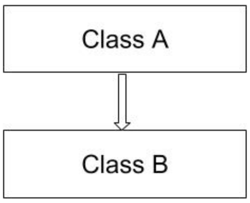
A class which inherits the properties is known as 'Child class' whereas a class whose properties are inherited is known as 'Parent class'.

Inheritance is classified into 4 types:



Single Inheritance

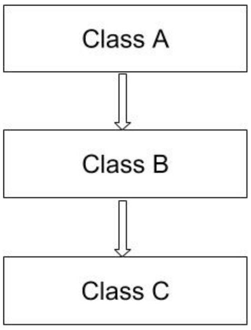
It enables a derived class to inherit the properties and behavior from a single parent class.



Here, Class A is a parent class and Class B is a child class which inherits the properties and behavior of the parent class.

Multilevel Inheritance

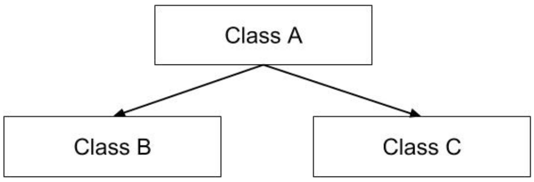
When a class is derived from a class which is also derived from another class, i.e. a class having more than one parent class but at different levels, such type of inheritance is called Multilevel Inheritance.



Here, class B inherits the properties and behavior of class A and class C inherits the properties of class B. Class A is the parent class for B and class B is the parent class for C. So, class C implicitly inherits the properties and methods of class A along with Class B.

Hierarchical Inheritance

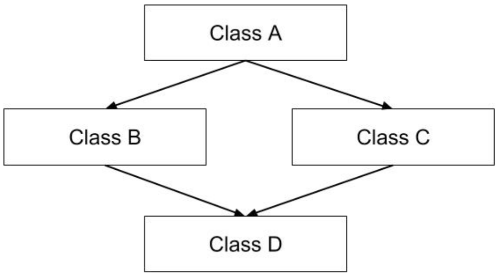
When a class has more than one child class (sub class), then such kind of inheritance is known as hierarchical inheritance.



Here, classes B and C are the child classes which are inheriting from the parent class A.

Hybrid Inheritance

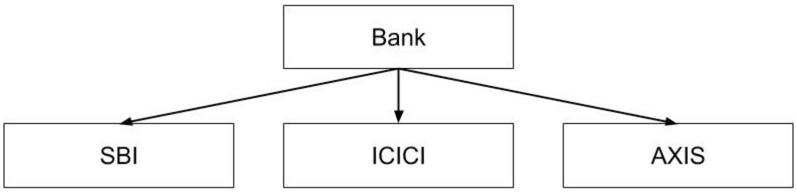
Hybrid inheritance is a combination of *multiple* inheritance and *multilevel* inheritance. Since multiple inheritance is not supported in Java as it leads to ambiguity, this type of inheritance can only be achieved through the use of the interfaces.



Here, class A is a parent class for classes B and C, whereas classes B and C are the parent classes of D which is the only child class of B and C.

Polymorphism

Polymorphism means taking many forms, where 'poly' means many and 'morph' means forms. It is the ability of a variable, function or object to take on multiple forms. In other words, polymorphism allows us to define one interface or method and have multiple implementations.



For eg, Bank is a base class that provides a method rate of interest. But, rate of interest may differ according to banks. For example, SBI, ICICI and AXIS are the child classes that provide different rates of interest.

Polymorphism in Java is of two types:

- Run time polymorphism
- Compile time polymorphism

Run time polymorphism:

In Java, runtime polymorphism refers to a process in which a call to an overridden method is resolved at runtime rather than at compile-time. Method overriding is an example of run time polymorphism.

Compile time polymorphism:

In Java, compile time polymorphism refers to a process in which a call to an overloaded method is resolved at compile time rather than at run time. Method overloading is an example of compile time polymorphism.

1.3 CHARACTERISTICS OF JAVA

Simple :

- Java is Easy to write and more readable.
- Java has a concise, cohesive set of features that makes it easy to learn and use.
- Most of the concepts are drawn from C++, thus making Java learning simpler.

Secure :

- Java program cannot harm other system thus making it secure.
- Java provides a secure means of creating Internet applications.
- Java provides secure way to access web applications.

Portable :

- Java programs can execute in any environment for which there is a Java run-time system.
- Java programs can run on any platform (Linux, Window, Mac)
- Java programs can be transferred over world wide web (e.g applets)

Object-oriented :

- Java programming is object-oriented programming language.
- Like C++, java provides most of the object oriented features.
- Java is pure OOP Language. (while C++ is semi object oriented)

Robust :

- Java encourages error-free programming by being strictly typed and performing run-time checks.

Multithreaded :

- Java provides integrated support for multithreaded programming.

Architecture-neutral :

- Java is not tied to a specific machine or operating system architecture.
- Java is machine independent.

Interpreted :

- Java supports cross-platform code through the use of Java bytecode.
- Bytecode can be interpreted on any platform by JVM (Java Virtual Machine).

High performance :

- Bytecodes are highly optimized.
- JVM can execute bytecodes much faster .

Distributed :

- Java is designed with the distributed environment.
- Java can be transmitted over internet.

Dynamic :

- Java programs carry substantial amounts of run-time type information with them that is used to verify and resolve accesses to objects at run time.

1.4 JAVA RUNTIME ENVIRONMENT (JRE)

The Java Runtime Environment (JRE) is a set of software tools for development of Java applications. It combines the Java Virtual Machine (JVM), platform core classes and supporting libraries.

JRE is part of the Java Development Kit (JDK), but can be downloaded separately. JRE was originally developed by Sun Microsystems Inc., a wholly-owned subsidiary of Oracle Corporation.

JRE consists of the following components:

Name of the component	Elements of the component
Deployment technologies	Deployment Java Web Start Java Plug-in
User interface toolkits	Abstract Window Toolkit (AWT) Swing Java 2D Accessibility Image I/O Print Service Sound Drag and Drop (DnD) Input methods.
Integration libraries	Interface Definition Language (IDL) Java Database Connectivity (JDBC) Java Naming and Directory Interface (JNDI) Remote Method Invocation (RMI) Remote Method Invocation Over Internet Inter-Orb Protocol (RMI-IIOP) Scripting.

base libraries	International support Input/Output (I/O) Eextension mechanism Beans Java Management Extensions (JMX) Java Native Interface (JNI) Math Networking Override Mechanism Security Serialization and Java for XML Processing (XML JAXP).
Lang and util base libraries	lang and util Management Versioning Zip Instrument Reflection Collections Concurrency Java Archive (JAR) Logging Preferences API Ref Objects Regular Expressions.
Java Virtual Machine (JVM)	Java HotSpot Client Server Virtual Machines

1.5 JAVA VIRTUAL MACHINE (JVM)

The JVM is a program that provides the runtime environment necessary for Java programs to execute. Java programs cannot run without JVM for the appropriate hardware and OS platform.

Java programs are started by a command line, such as:

```
java <arguments> <program name>
```

This brings up the JVM as an operating system process that provides the Java runtime environment. Then the program is executed in the context of an empty virtual machine.

When the JVM takes in a Java program for execution, the program is not provided as Java language source code. Instead, the Java language source must have been converted (or compiled) into a form known as Java bytecode. Java bytecode must be supplied to the JVM in a format called class files. These class files always have a .class extension.

The JVM is an interpreter for the bytecode form of the program. It steps through one bytecode instruction at a time. It is an abstract computing machine that enables a computer to run a Java program.

1.6 SETTING UP AN ENVIRONMENT FOR JAVA

Local Environment Setup

Download Java and run the .exe to install Java on the machine.

Setting Up the Path for Windows

Assuming Java is installed in c:\Program Files\java\jdk directory −

- Right-click on ‘My Computer’ and select ‘Properties’.
- Click the ‘Environment variables’ button under the ‘Advanced’ tab.
- Now, alter the ‘Path’ variable so that it also contains the path to the Java executable. Example, if the path is currently set to ‘C:\WINDOWS\SYSTEM32’, then change your path to read ‘C:\WINDOWS\SYSTEM32;c:\Program Files\java\jdk\bin’.

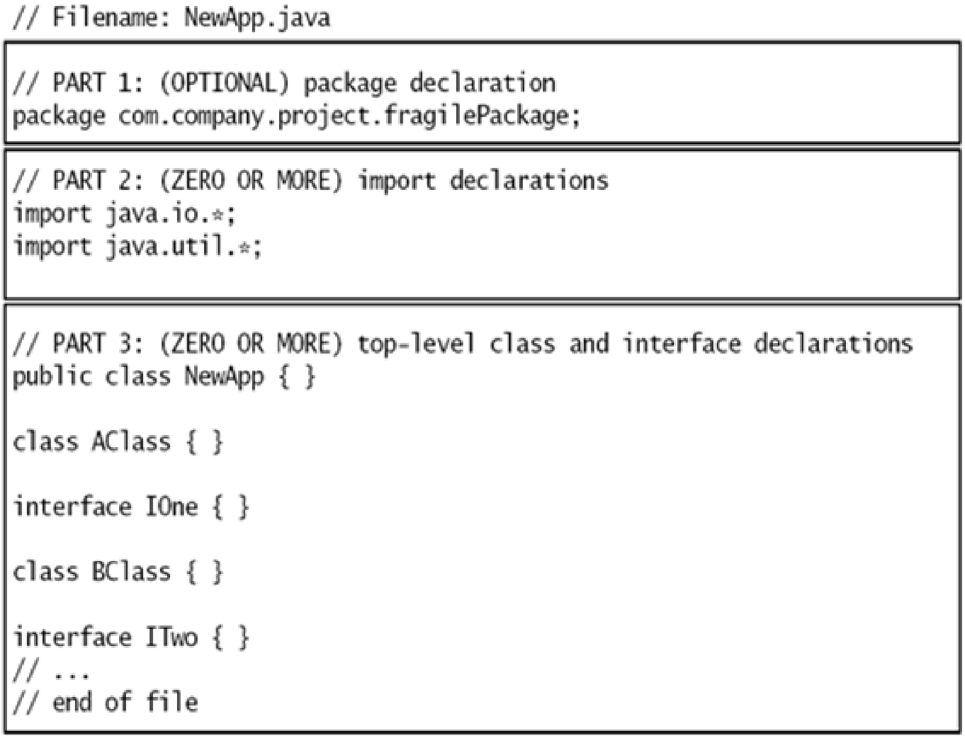
1.7 POPULAR JAVA EDITORS

To write Java programs, we need any of the following:

- **Notepad** − Text editor
- **Netbeans** − A Java IDE that is open-source and free
- **Eclipse** − A Java IDE developed by the eclipse open-source community

1.8 JAVA SOURCE FILE STRUCTURE

When we write a Java source program, it needs to follow a certain structure or template as shown in the following figure:



```
// Filename: NewApp.java

// PART 1: (OPTIONAL) package declaration
package com.company.project.fragilePackage;

// PART 2: (ZERO OR MORE) import declarations
import java.io.*;
import java.util.*;

// PART 3: (ZERO OR MORE) top-level class and interface declarations
public class NewApp { }

class AClass { }

interface IOne { }

class BClass { }

interface ITwo { }
// ...
// end of file
```

Figure: Java Source File Structure

Packages are used in Java in order to prevent naming conflicts, to control access, to make searching/locating and usage of classes, interfaces, enumerations and annotations easier, etc.

A Java source file can have the following elements that must be specified in the following order:

1. An optional package declaration to specify a package name.
2. Zero or more import declarations.
3. Any number of top-level type declarations. Class, enum, and interface declarations are collectively known as type declarations.

Part 1: ***Optional Package Declaration***

A package is a pack (group) of classes, interfaces and other packages. Packages are used in Java in order to prevent naming conflicts, to control access, to make searching / locating and usage of classes, interfaces, enumerations and annotations easier, etc.

Rules:

- The package statement should be the first line in the source file.
- There can be only one package statement in each source file.
- If a package statement is not used, the class, interfaces, enumerations, and annotation types will be placed in the current default package.
- It is a good practice to use names of packages with lower case letters to avoid any conflicts with the names of classes and interfaces.

Following package example contains interface named animals:

```
/* File name : Animal.java */
package animals;
interface Animal
{
   public void eat();
   public void travel();
}
```

Part 2: ***Zero or More import Declarations***

The import statement makes the declarations of external classes available to the current Java source program at the time of compilation. The import statement specifies the path for the compiler to find the specified class.

Syntax of the import statement:

```
import packagename;

or

import packagename.* ;
```

We may import a single class or all the classes belonging to a package.

- To import a single class, we specify the name of the class
- To import all classes, we specify *.

Examples of the import statement:

Statement in Java	Purpose
import mypackage.MyClass;	imports the definition of the MyClass class that is defined in the mypackage package.
import mypackage.reports.accounts.salary. EmpClass;	imports the definition of EmpClass belonging to the mypackage.reports. accounts.salary package.
import java.awt.*;	imports all the classes belonging to the java.awt package.

Part 3: ***Zero or More top-level Declarations***

The Java source file should have one and only one public class. The class name which is defined as public should be the name of Java source file along with .java extension.

Source File Declaration Rules

- 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 which should have .java extension at the end.
- For eg, if 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.

1.9 COMPILATION

In Java, programs are not compiled into executable files. Java source code is compiled into bytecode using javac compiler. The bytecodes are platform-independent instructions for the Java VM. They are saved on the disk with the file extension `.class`. When the program is to be run, the bytecode is converted into the machine code using the just-in-time (JIT) compiler. It is then fed to the memory and executed.

Java code needs to be compiled twice in order to be executed:

1. Java programs need to be compiled to bytecode.
2. When the bytecode is run, it needs to be converted to machine code.

The Java classes / bytecode are compiled to machine code and loaded into memory by the JVM when needed for the first time.

Compiling the Program

The Java compiler is invoked at the command line with the following syntax:

```
javac ExampleProgram.java
```

Interpreting and Running the Program

Once the java program successfully compiles into Java bytecodes, we can interpret and run applications on any Java VM, or interpret and run applets in any Web browser with a Java VM built in such as Netscape or Internet Explorer. Interpreting and running a Java program means invoking the Java VM byte code interpreter, which converts the Java byte codes to platform-dependent machine codes so your computer can understand and run the program.

The Java interpreter is invoked at the command line with the following syntax:

```
java ExampleProgram
```

Quick compilation procedure

To execute the first Java program, follow the steps:

1. Open text editor. For example, Notepad or Notepad++ on Windows; Gedit, Kate or SciTE on Linux; or, XCode on Mac OS, etc.
2. Type the java program in a new text document.
3. Save the file as HelloWorld.java.
4. Next, open any command-line application. For example, Command Prompt on Windows; and, Terminal on Linux and Mac OS.
5. Compile the Java source file using the command: javac HelloWorld.java
6. Once the compiler returns to the prompt, run the application using the following command:

   ```
   java HelloWorld
   ```

1.10 FUNDAMENTAL PROGRAMMING STRUCTURES IN JAVA

Java Comments

The java comments are statements that are not executed by the compiler and interpreter. The comments can be used to provide information or explanation about the variable, method, class or any statement. It can also be used to hide program code for specific time.

Types of Java Comments

There are 3 types of comments in java.

1. Single Line Comment
2. Multi Line Comment
3. Documentation Comment

1) Java Single Line Comment

The single line comment is used to comment only one line. A single-line comment begins with a // and ends at the end of the line.

Syntax	Example
//Comment	//This is single line comment

2) Java Multi Line Comment

This type of comment must begin with /* and end with */. Anything between these two comment symbols is ignored by the compiler. A multiline comment may be several lines long.

Syntax	Example
/*Comment starts continues continues . . . Commnent ends*/	/* This is a multi line comment */

3) Java Documentation Comment

This type of comment is used to produce an HTML file that documents our program. The documentation comment begins with a /** and ends with a */.

Syntax	Example
/**Comment start	/**
*	This
*tags are used in order to specify a parameter	is
*or method or heading	documentation
*HTML tags can also be used	comment
*such as <h1>	*/
*	
comment ends/	

1.11 DATA TYPES

Java is a **statically typed and also a strongly typed language. I**n Java, each type of data (such as integer, character, hexadecimal, etc.) is predefined as part of the programming language and all constants or variables defined within a given program must be described with one of the data types.

Data types represent the different values to be stored in the variable. In java, there are two categories of data types:

- Primitive data types
- Non-primitive data types

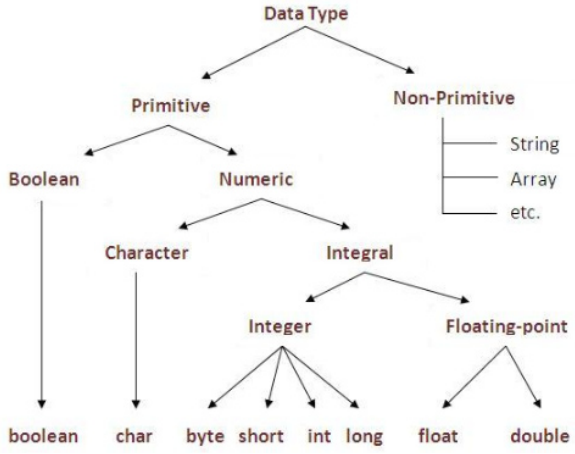


Figure: Data types in java

The Primitive Types

Java defines eight primitive types of data: byte, short, int, long, char, float, double, and boolean. The primitive types are also commonly referred to as simple types and they are grouped into the following four groups:

i) ***Integers*** - This group includes byte, short, int, and long. All of these are signed, positive and negative values. The width and ranges of these integer types vary widely, as shown in the following table:

Name	Width in bits	Range
long	64	–9,223,372,036,854,775,808 to 9,223,372,036,854,775,807
int	32	–2,147,483,648 to 2,147,483,647
short	16	–32,768 to 32,767
byte	8	–128 to 127

Table: Integer Data Types

ii) ***Floating-point numbers*** – They are also known as real numbers. This group includes float and double, which represent single- and double-precision numbers, respectively. The width and ranges of them are shown in the following table:

Table: Floating-point Data Types

Name	Width in bits	Range
double	64	4.9e–324 to 1.8e+308
float	32	1.4e–045 to 3.4e+038

iii) ***Characters*** - This group includes char, which represents symbols in a character set, like letters and numbers. char is a 16-bit type. The range of a char is 0 to 65,536. There are no negative chars.

iv) ***Boolean*** - This group includes boolean. It can have only one of two possible values, true or false.

1.12 VARIABLES

A variable is the holder that can hold the value while the java program is executed. A variable is assigned with a datatype. It is name of *reserved area allocated in memory*. In other words, it is a *name of memory location*. There are three types of variables in java: local, instance and static.

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.

Before using any variable, it must be declared. The following statement expresses the basic form of a variable declaration –

datatype variable [= value][, variable [= value] ...] ;

Here data type is one of Java's data types and variable is the name of the variable. To declare more than one variable of the specified type, use a comma-separated list.

Example

```
int a, b, c;        // Declaration of variables a, b, and c.
int a = 20, b = 30;  // initialization
byte B = 22;        // Declaratrion initializes a byte type variable B.
```

Types of Variable

There are three types of variables in java:

- local variable
- instance variable
- static variable

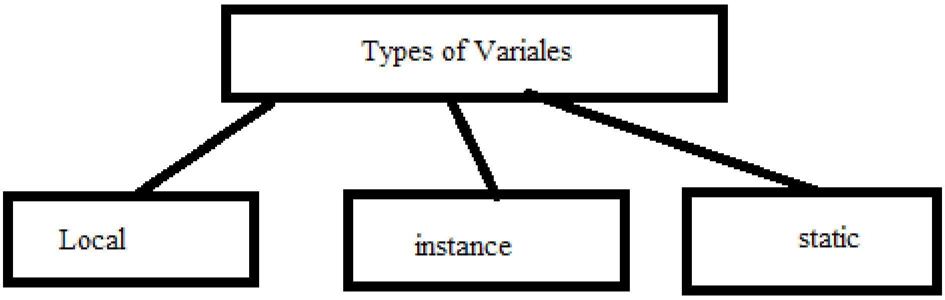


Fig. Types of variables

Local Variable

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

Instance Variable

- A variable declared inside the class but outside the method, is called instance variable. Instance variables are declared in a class, but outside a method, constructor or any block.
- A slot for each instance variable value is created when a space is allocated for an object in the heap.
- Instance variables are created when an object is created with the use of the keyword ‘new’ and destroyed when the object is destroyed.
- Instance variables 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.
- Instance variables can be declared in class level before or after use.
- Access modifiers can be given for instance variables.
- The instance variables are visible for all methods, constructors and block in the class. It is recommended to make these variables as private. However, visibility for subclasses can be given for these variables with the use of access modifiers.
- Instance variables have default values.
    - numbers, the default value is 0,
    - Booleans it is false,
    - Object references it is null.
- Values can be assigned during the declaration or within the constructor.
- Instance variables cannot be declared as static.
- Instance variables can be accessed directly by calling the variable name inside the class. However, within static methods (when instance variables are given accessibility), they should be called using the fully qualified name. ObjectReference.VariableName.

Static variable

- Class variables also known as static variables are declared with the static keyword in a class, but outside a method, constructor or a block.
- Only one copy of each class variable per class is created, 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 the 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.
- Visibility is same as instance variables. However, most static variables are declared public since they must be available for users of the class.
- Default values are same as instance variables.
  - numbers, the default value is 0;
  - Booleans, it is false;
  - Object references, it is null.
- Values can be assigned during the declaration or within the constructor. Additionally, values can be assigned in special static initializer blocks.
- Static variables cannot be local.
- Static variables can be accessed by calling with the class name ClassName. VariableName.
- When declaring class variables as public static final, then variable names (constants) are all in upper case. If the static variables are not public and final, the naming syntax is the same as instance and local variables.

1.13 OPERATORS

Operator in java is a symbol that is used to perform operations. Java provides a rich set of operators to manipulate variables.For example: +, -, *, / etc.

All the Java operators can be divided into the following groups −

- Arithmetic Operators :

  Multiplicative : `* / %`

  Additive : `+ -`

- Relational Operators

  Comparison : `< > <= >= instanceof`

  Equality : `== !=`

- Bitwise Operators

  bitwise AND : `&`

  bitwise exclusive OR : `^`

  bitwise inclusive OR : `|`

  Shift operator: << >> >>>

- Logical Operators

  logical AND : &&

  logical OR : ||

  logical NOT : ~ !
- Assignment Operators: =
- Ternary operator: ? :
- Unary operator

  Postfix : *expr*++ *expr*–

  Prefix : ++*expr* --*expr* +*expr* -*expr*

The Arithmetic Operators

Arithmetic operators are used to perform arithmetic operations in the same way as they are used in algebra. The following table lists the arithmetic operators −

Example:

int A=10,B=20;

Operator	**Description**	**Example**	**Output**
+ (Addition)	Adds values A & B.	A + B	30
- (Subtraction)	Subtracts B from A	A - B	-10
* (Multiplication)	Multiplies values A & B	A * B	200
/ (Division)	Divides B by A	B / A	2
% (Modulus)	Divides left-hand operand by right-hand operand and returns remainder.	B % A	0

// Java program to illustrate arithmetic operators

```
public class Aoperators
{
   public static void main(String[] args)
   {
      int a = 20, b = 10, c = 0, d = 20, e = 40, f = 30;
      String x = “Thank”, y = “You”;
       System.out.println(“a + b = “+(a + b));
      System.out.println(“a - b = “+(a - b));
```

```
            System.out.println("x + y = "+x + y);
            System.out.println("a * b = "+(a * b));
            System.out.println("a / b = "+(a / b));
                System.out.println("a % b = "+(a % b));
        }
    }
```

The Relational Operators

The following relational operators are supported by Java language.

Example:

int A=10,B=20;

Operator	Description	Example	Output
== (equal to)	Checks if the values of two operands are equal or not, if yes then condition becomes true.	(A == B)	true
!= (not equal to)	Checks if the values of two operands are equal or not, if values are not equal then condition becomes true.	(A != B)	true
> (greater than)	Checks if the value of left operand is greater than the value of right operand, if yes then condition becomes true.	(A > B)	true
< (less than)	Checks if the value of left operand is less than the value of right operand, if yes then condition becomes true.	(A < B)	true
>= (greater than or equal to)	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)	true
<= (less than or equal to)	Checks if the value of left operand is less than or equal to the value of right operand, if yes then condition becomes true.	(A <= B)	true
instance of Operator	checks whether the object is of a particular type (class type or interface type) (Object reference variable) instanceof (class/interface type)	boolean result = name instanceof String;	True

// Java program to illustrate relational operators

```
public class operators
{
    public static void main(String[] args)
    {
        int a = 20, b = 10;
        boolean condition = true;
         //various conditional operators
        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(“a >= b :” + (a >= b));
        System.out.println(“a != b :” + (a != b));
        System.out.println(“condition==true :” + (condition == true));
    }
}
```

Bitwise Operators

Java supports several bitwise operators, that can be applied to the integer types, long, int, short, char, and byte. Bitwise operator works on bits and performs bit-by-bit operation.

Example:

int a = 60,b = 13;

binary format of a & b will be as follows −

a = 0011 1100

b = 0000 1101

Bitwise operators follow the truth table:

a	b	a&b	a|b	a^b	~a
0	0	0	0	1	1
0	1	0	1	0	1
1	0	0	1	0	0
1	1	1	1	1	0

a&b = 0000 1100

a|b = 0011 1101

a^b = 0011 0001

~a = 1100 0011

The following table lists the bitwise operators −

int A=60,B=13;

Operator	Description	Example	Output
& (bitwise and)	Binary AND Operator copies a bit to the result if it exists in both operands.	(A & B) will give 12 which is	12 (in binary form:0000 1100)
| (bitwise or)	Binary OR Operator copies a bit if it exists in either operand.	(A | B)	61 (in binary form: 0011 1101)
^ (bitwise XOR)	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	49 (in binary form: 0011 0001)
~ (bitwise compliment)	Binary Ones Complement Operator is unary and has the effect of 'flipping' bits.	(~A) will give -61 which is 1100 0011 in 2's complement form due to a signed binary number.	-61 (in binary form: 1100 0011)
<< (left shift)	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	240 (in binary form: 1111 0000)

>> (right shift)	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	15 (in binary form: 1111)
>>> (zero fill right shift)	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	15 (in binary form: 0000 1111)

// Java program to illustrate bitwise operators

```
public class operators
{
    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 = “ + ~a);
    }
}
```

Logical Operators

The following are the logical operators supported by java.

Example:

A=true;

B=false;

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

Assignment Operators

The following are the assignment operators supported by Java.

Operator	Description	Example
= (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 assigns 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 assigns 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 assigns 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 assigns the result to left operand.	C /= A is equivalent to C = C / A
%= (Modulus AND assignment operator)	It takes modulus using two operands and assigns 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

// Java program to illustrate assignment operators

```
public class operators
{
   public static void main(String[] args)
   {
      int a = 20, b = 10, c, d, e = 10, f = 4, g = 9;
       c = b;
      System.out.println("Value of c = " + c);
      a += 1;
```

```
        b -= 1;
        e *= 2;
        f /= 2;
        System.out.println(“a, b, e, f = “ +
                            a + “,” + b + “,” + e + “,” + f);
      }
    }
```

Ternary Operator

Conditional Operator (? :)

Since the conditional operator has three operands, it is referred as the **ternary operator**. This operator consists of three operands and is used to evaluate Boolean expressions. The goal of the operator is to decide, which value should be assigned to the variable. The operator is written as –

variable x = (expression) ? value if true : value if false

Following is an example −

Example:

```
public class example
{
public static void main(String args[])
{
int a, b;
a = 10;
b = (a == 0) ? 20: 30;
System.out.println( “b : “ +  b );
}
}
```

Unary Operators

Unary operators use only one operand. They are used to increment, decrement or negate a value.

Operator	Description
- Unary minus	negating the values
+ Unary plus	converting a negative value to positive
++ :Increment operator	incrementing the value by 1
— : Decrement operator	decrementing the value by 1
! : Logical not operator	inverting a boolean value

// Java program to illustrate unary operators

```
public class operators
{
public static void main(String[] args)
{
int a = 20, b = 10, c = 0, d = 20, e = 40, f = 30;
boolean condition = true;
c = ++a;
System.out.println("Value of c (++a) = " + c);
c = b++;
System.out.println("Value of c (b++) = " + c);
c = --d;
System.out.println("Value of c (--d) = " + c);
c = --e;
System.out.println("Value of c (--e) = " + c);
System.out.println("Value of !condition =" + !condition);
}
}
```

Precedence of Java Operators

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

For example, the following expression,

x = 10 + 5 * 2;

is evaluated. So, the output is 20, not 30. Because operator * has higher precedence than +.

The following table shows the operators with the highest precedence at the top of the table and those with the lowest 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

1.14 CONTROL FLOW

Java Control statements control the flow of execution in a java program, based on data values and conditional logic used. There are three main categories of control flow statements;

Selection statements: if, if-else and switch.

Loop statements: while, do-while and for.

Transfer statements: break, continue, return, try-catch-finally and assert.

Selection statements

The selection statements checks the condition only once for the program execution.

If Statement:

The if statement executes a block of code only if the specified expression is true. If the value is false, then the if block is skipped and execution continues with the rest of the program.

The simple if statement has the following syntax:

```
        if (<conditional expression>)
        <statement action>
```

The following program explains the if statement.

```
public class programIF{
public static void main(String[] args)
{
int a = 10, b = 20;
if (a > b)
System.out.println(“a > b”);
if (a < b)
System.out.println(“b < a”);
}
}
```

The If-else Statement

The if/else statement is an extension of the if statement. If the condition in the if statement fails, the statements in the else block are executed. The if-else statement has the following syntax:

```
    if (<conditional expression>)
            <statement action>
    else
            <statement action>
```

The following program explains the if-else statement.

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

```
int a = 10, b = 20;
if (a > b)
{
System.out.println(“a > b”);
}
else
{
System.out.println(“b < a”);
}
}
}
```

Switch Case Statement

The switch case statement is also called as multi-way branching statement with several choices. A switch statement is easier to implement than a series of if/else statements. The switch statement begins with a keyword, followed by an expression that equates to a no long integral value.

After the controlling expression, there is a code block that contains zero or more labeled cases. Each label must equate to an integer constant and each must be unique. When the switch statement executes, it compares the value of the controlling expression to the values of each case label.

The program will select the value of the case label that equals the value of the controlling expression and branch down that path to the end of the code block. If none of the case label values match, then none of the codes within the switch statement code block will be executed.

Java includes a default label to use in cases where there are no matches. A nested switch within a case block of an outer switch is also allowed. When executing a switch statement, the flow of the program falls through to the next case. So, after every case, you must insert a break statement.

The syntax of switch case is given as follows:

switch (<non-long integral expression>) {

case label$_1$: <statement$_{1>}$

case label$_2$: <statement$_{2>}$

...

case label$_n$: <statement$_{n>}$

default: <statement>

} // end switch

The following program explains the switch statement.

```
public class ProgramSwitch
{
public static void main(String[] args)
{
int a = 10, b = 20, c = 30;
int status = -1;
if (a > b && a > c)
{
status = 1;
}
else if (b > c)
{
status = 2;
}
else
{
status = 3;
}
switch (status)
{
case 1:System.out.println("a is the greatest");
break;
case 2:System.out.println("b is the greatest");
break;
case 3:System.out.println("c is the greatest");
break;
default:System.out.println("Cannot be determined");
}
}
}
```

Iteration statements

Iteration statements execute a block of code for several numbers of times until the condition is true.

While Statement

The while statement is one of the looping constructs control statement that executes a block of code while a condition is true. The loop will stop the execution if the testing expression evaluates to false. The loop condition must be a boolean expression. The syntax of the while loop is

```
while (<loop condition>)
<statements>
```

The following program explains the while statement.

```
public class ProgramWhile
{
public static void main(String[] args)
{
int count = 1;
System.out.println("Printing Numbers from 1 to 10");
while (count <= 10)
{
System.out.println(count++);}
}
}
}
```

Do-while Loop Statement

The do-while loop is similar to the while loop, except that the test condition is performed at the end of the loop instead of at the beginning. The do—while loop executes atleast once without checking the condition.

It begins with the keyword do, followed by the statements that making up the body of the loop. Finally, the keyword while and the test expression completes the do-while loop. When the loop condition becomes false, the loop is terminated and execution continues with the statement immediately following the loop.

The syntax of the do-while loop is

```
do
<loop body>
while (<loop condition>);
```

The following program explains the do--while statement.

```
public class DoWhileLoopDemo {
public static void main(String[] args) {
int count = 1;
System.out.println("Printing Numbers from 1 to 10");
do {
System.out.println(count++);
} while (count <= 10);
}
}
```

For Loop

The for loop is a looping construct which can execute a set of instructions for a specified number of times. It's a counter controlled loop.

The syntax of the loop is as follows:

```
for (<initialization>; <loop condition>; <increment expression>)
<loop body>
```

- initialization statement executes once before the loop begins. The <initialization> section can also be a comma-separated list of expression statements.
- test expression. As long as the expression is true, the loop will continue. If this expression is evaluated as false the first time, the loop will never be executed.

- Increment(Update) expression that automatically executes after each repetition of the loop body.
- All the sections in the for-header are optional. Any one of them can be left empty, but the two semicolons are mandatory.

The following program explains the for statement.

```
public class ProgramFor {
public static void main(String[] args) {
System.out.println("Printing Numbers from 1 to 10");
for (int count = 1; count <= 10; count++) {
System.out.println(count);
}
}
}
```

Transfer statements

Transfer statements are used to transfer the flow of execution from one statement to another.

Continue Statement

A continue statement stops the current iteration of a loop (while, do or for) and causes execution to resume at the top of the nearest enclosing loop. The continue statement can be used when you do not want to execute the remaining statements in the loop, but you do not want to exit the loop itself.

The syntax of the continue statement is

```
continue; // the unlabeled form
continue <label>; // the labeled form
```

It is possible to use a loop with a label and then use the label in the continue statement. The label name is optional, and is usually only used when you wish to return to the outermost loop in a series of nested loops.

The following program explains the continue statement.

```
public class ProgramContinue
{
public static void main(String[] args) {
System.out.println("Odd Numbers");
for (int i = 1; i <= 10; ++i) {
if (i % 2 == 0)
continue;
System.out.println(i + “\t”);
}
}
}
```

Break Statement

The break statement terminates the enclosing loop (for, while, do or switch statement). Break statement can be used when we want to jump immediately to the statement following the enclosing control structure. As continue statement, can also provide a loop with a label, and then use the label in break statement. The label name is optional, and is usually only used when you wish to terminate the outermost loop in a series of nested loops.

The Syntax for break statement is as shown below;

```
break; // the unlabeled form
break <label>; // the labeled form
```

The following program explains the break statement.

```
public class ProgramBreak {
public static void main(String[] args) {
System.out.println(“Numbers 1 - 10”);
for (int i = 1;; ++i) {
if (i == 11)
break;
// Rest of loop body skipped when i is even
System.out.println(i + “\t”);
}
}
}
```

The transferred statements such as try-catch-finally, throw will be explained in the later chapters.

1.15 DEFINING CLASSES IN JAVA

A class is an entity that determines how an object will behave and what the object will contain. A class *is* the basic building block of an object-oriented language such as Java. It is acting as a template that describes the data and behavior associated with instances of that class.

When you instantiate a class means creating an object. The class contains set of variables and methods.

The data associated with a class or object is stored in variables; the behavior associated with a class or object is implemented with methods. A class is a blueprint from which individual objects are created.

```
class MyClass {
   // field,
   //constructor, and
   // method declarations
}
```

Example:

```
class Myclass{
   public static void main(String[] args)
   {
      System.out.println("Hello World!"); //Display the string.
   }
}
```

The keyword class begins the class definition for a class named name. The variables and methods of the class are embraced by the curly brackets that begin and end the class definition block. The "Hello World" application has no variables and has a single method named main.

In Java, the simplest form of a class definition is

```
class name {
   . . .
}
```

In general, class declarations can include these components, in order:

1. ***Modifiers :*** A class can be public or has default access.
2. ***Class name:*** The name should begin with a initial letter.

3. ***Superclass(if any):*** The name of the class's parent (superclass), if any, preceded by the keyword extends. A class can only extend (subclass) one parent.
4. ***Interfaces(if any):*** A comma-separated list of interfaces implemented by the class, if any, preceded by the keyword implements. A class can implement more than one interface.
5. ***Body:*** The class body surrounded by braces, { }.

1.16 CONSTRUCTORS

Every class has a constructor. If the constructor is not defined in the class, the Java compiler builds a default constructor for that class. While 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.

Constructors are used for initializing new objects. Fields are variables that provide the state of the class and its objects, and methods are used to implement the behavior of the class and its objects.

Rules for writing Constructor

- Constructor(s) of a class must have same name as the class name in which it resides.
- A constructor in Java cannot be abstract, final, static and synchronized.
- Access modifiers can be used in constructor declaration to control its access i.e which other class can call the constructor.

Following is an example of a constructor −

Example

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

Types of Constructors

There are two type of constructor in Java:

1. No-argument constructor:

A constructor that has no parameter is known as default constructor.

If the constructor is not defined in a class, then compiler creates default constructor (with no arguments) for the class. If we write a constructor with arguments or no-argument then compiler does not create default constructor. Default constructor provides the default values to the object like 0, null etc. depending on the type.

// Java Program to illustrate calling a no-argument constructor

```
import java.io.*;
class myclass
{
    int num;
String name;
     // this would be invoked while object of that class created.
    myclass()
    {
        System.out.println(“Constructor called”);
    }
}
 class myclassmain
{
    public static void main (String[] args)
    {
        // this would invoke default constructor.
        myclass m1 = new myclass();
 // Default constructor provides the default values to the object like 0, null
        System.out.println(m1.num);
        System.out.println(m1.name);
    }
}
```

2. Parameterized Constructor

A constructor that has parameters is known as parameterized constructor. If we want to initialize fields of the class with your own values, then use parameterized constructor.

// Java Program to illustrate calling of parameterized constructor.

```
import java.io.*;
 class myclass
{
   // data members of the class.
   String name;
   int num;
    // contructor with arguments.
   myclass(String name, int n)
   {
      this.name = name;
      this.num = n;
  }
}
 class myclassmain{
   public static void main (String[] args)
   {
      // this would invoke parameterized constructor.
      myclass m1 = new myclass("Java", 2017);
      System.out.println("Name :" + m1.name + " num :" + m1.num);
   }
   }
```

There are no "return value" statements in constructor, but constructor returns current class instance. We can write 'return' inside a constructor.

1.17 CONSTRUCTOR OVERLOADING

Like methods, we can overload constructors for creating objects in different ways. Compiler differentiates constructors on the basis of numbers of parameters, types of the parameters and order of the parameters.

// Java Program to illustrate constructor overloading

```
    import java.io.*;
    class myclass
    {
        // constructor with one argument
        myclass (String name)
        {
            System.out.println(“Constructor with one “ + “argument - String : “ + name);
        }
       // constructor with two arguments
        myclass (String name, int id)
        {
       System.out.print(“Constructor with two arguments : “ +” String and Integer : “ + name
+ “ “+ id);
        }
         // Constructor with one argument but with different type than previous.
        myclass (long num)
        {
            System.out.println(“Constructor with one argument : “ +”Long : “ + num);
        }
    }
     class  myclassmain
    {
        public static void main(String[] args)
        {
           myclass m1 = new myclass (“JAVA”);
           myclass m2 = new myclass (“Python”, 2017);
           myclass m3 = new myclass(3261567);
        }
    }
```

Constructors are different from methods in Java

- Constructor(s) must have the same name as the class within which it defined while it is not necessary for the method in java.
- Constructor(s) do not any return type while method(s) have the return type or **void** if does not return any value.
- Constructor is called only once at the time of Object creation while method(s) can be called any numbers of time.

Creating an Object

The class provides the blueprints for objects. The objects are the instances of the class. In Java, the new keyword is used to create new objects.

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.

1.18 METHODS IN JAVA

A method is a collection of statement that performs specific task. In Java, each method is a part of a class and they define the behavior of that class. In Java, method is a jargon used for method.

Advantages of methods

- Program development and debugging are easier
- Increases code sharing and code reusability
- Increases program readability
- It makes program modular and easy to understanding
- It shortens the program length by reducing code redundancy

Types of methods

There are two types of methods in Java programming:

- Standard library methods (built-in methods or predefined methods)
- User defined methods

Standard library methods

The standard library methods are built-in methods in Java programming to handle tasks such as mathematical computations, I/O processing, graphics, string handling etc. These methods are already defined and come along with Java class libraries, organized in packages. In order to use built-in methods, we must import the corresponding packages. Some of library methods are listed below.

Packages	Library Methods	Descriptions
java.lang.Math	acos()	Computes arc cosine of the argument
All maths related methods are defined in this class	exp()	Computes the e raised to given power
	abs()	Computes absolute value of argument
	log()	Computes natural logarithm
	sqrt()	Computes square root of the argument
	pow()	Computes the number raised to given power
java.lang.String	charAt()	Returns the char value at the specified index.
All string related methods are defined in this class	concat()	Concatenates two string
	compareTo()	Compares two string
	indexOf()	Returns the index of the first occurrence of the given character
	toUpperCase()	converts all of the characters in the String to upper case
java.awt	add()	inserts a component
contains classes for graphics	setSize()	set the size of the component
	setLayout()	defines the layout manager
	setVisible()	changes the visibility of the component

Example:

Program to compute square root of a given number using built-in method.

```
public class MathEx {
   public static void main(String[] args) {
      System.out.print("Square root of 14 is: " + Math.sqrt(14));
   }
}
```

Sample Output:

```
Square root of 14 is: 3.7416573867739413
```

User-defined methods

The methods created by user are called user defined methods.

Every method has the following.

- Method declaration (also called as method signature or method prototype)
- Method definition (body of the method)
- Method call (invoke/activate the method)

Method Declaration

The syntax of method declaration is:

Syntax:

```
return_type method_name(parameter_list);
```

Here, the return_type specifies the data type of the value returned by method. It will be void if the method returns nothing. method_name indicates the unique name assigned to the method. parameter_list specifies the list of values accepted by the method.

Method Definition

Method definition provides the actual body of the method. The instructions to complete a specific task are written in method definition. The syntax of method is as follows:

Syntax:

```
modifier return_type method_name(parameter_list){
// body of the method
}
```

Here,

Modifier – Defines the access type of the method i.e accessibility region of method in the application

return_type – Data type of the value returned by the method or void if method returns nothing

method_name – Unique name to identify the method. The name must follow the rules of identifier

parameter_list – List of input parameters separated by comma. It must be like

datatype parameter1,datatype parameter2,……

List will be empty () in case of no input parameters.

method body – block of code enclosed within { and } braces to perform specific task

The first line of the method definition must match exactly with the method prototype. A method cannot be defined inside another method.

Method Call

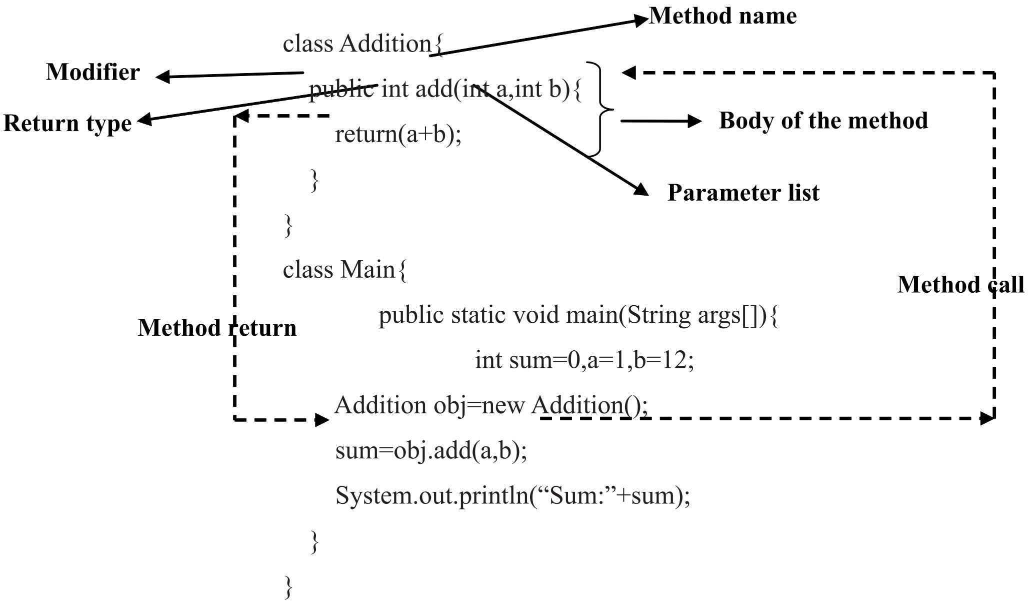
A method gets executed only when it is called. The syntax for method call is.

Syntax:

method_name(parameters);

When a method is called, the program control transfers to the method definition where the actual code gets executed and returns back to the calling point. The number and type of parameters passed in method call should match exactly with the parameter list mentioned in method prototype.

Example:



```
class Addition{
   public int add(int a,int b){
      return(a+b);
   }
}
class Main{
      public static void main(String args[]){
            int sum=0,a=1,b=12;
      Addition obj=new Addition();
      sum=obj.add(a,b);
      System.out.println("Sum:"+sum);
   }
}
```

Sample Output:

```
Sum:13
```

Memory allocation for methods calls

Method calls are implemented using stack. When a method is called, the parameters passed in the call, local variables defined inside method, and return value of the method are stored in stack frame. The allocated stack frame gets deleted automatically at the end of method execution.

Types of User-defined methods

The methods in C are classified based on data flow between calling method and called method. They are:

- Method with no arguments and no return value
- Method with no arguments and a return value
- Method with arguments and no return value
- Method with arguments and a return value.

Method with no arguments and no return value

In this type of method, no value is passed in between calling method and called method. Here, when the method is called program control transfers to the called method, executes the method, and return back to the calling method.

Example:

Program to compute addition of two numbers (no argument and no return value)

```
public class Main{
    public void add(){          //  method definition with no arguments and no return value
        int a=10,b=20;
        System.out.println("Sum:"+(a+b));
    }
    public static void main(String[] args) {
        Main obj=new Main();
        obj.add();               // method call with no arguments
    }
}
```

Sample Output:

```
Sum:30
```

Method with no arguments and a return value

In this type of method, no value is passed from calling method to called method but a value is returned from called method to calling method.

Example:

Program to compute addition of two numbers (no argument and with return value)

```
public class Main {
   public int add(){   // method definition with no arguments and with return value
      int a=10,b=20;
      return(a+b);
   }
   public static void main(String[] args) {
      int sum=0;
      Main obj=new Main();
                                  /* method call with no arguments. The value returned
     sum=obj.add(); ──────────→   from the method is assigned to variable sum */
      System.out.println(“Sum:”+sum);
   }
}
```

Sample Output:

```
Sum:30
```

Method with arguments and no return value

In this type of method, parameters are passed from calling method to called method but no value is returned from called method to calling method.

Example:

Program to compute addition of two numbers (with argument and without return value)

```
public class Main {
   public void add(int x,int y){   // method definition with arguments and no return value
      System.out.println(“Sum:”+(x+y));
   }
   public static void main(String[] args) {
      int a=10,b=20;
      Main obj=new Main();
      obj.add(a,b);   // method call with arguments
   }
}
```

Sample Output:

```
Sum:30
```

Method with arguments and a return value.

In this type of method, there is data transfer in between calling method and called method. Here, when the method is called program control transfers to the called method with arguments, executes the method, and return the value back to the calling method.

Example:

Program to compute addition of two numbers (with argument and return value)

```
public class Main {
   public int add(int x,int y){      // function definition with arguments and return value
      return(x+y);    //return value
   }
   public static void main(String[] args) {
     int a=10,b=20;
                                                /* method call with arguments. The value returned from
     Main obj=new Main();          →            the method is displayed within main() */
     System.out.println("Sum:"+obj.add(a,b));
   }
}
```

Sample Output:

```
Sum:30
```

1.19 PARAMETER PASSING IN JAVA

The commonly available parameter passing methods are:

- Pass by value
- Pass by reference

Pass by Value

In pass by value, the value passed to the method is copied into the local parameter and any change made inside the method only affects the local copy has no effect on the original copy. In Java, parameters are always passed by value. All the scalar variables (of type int, long, short, float, double, byte, char, Boolean) are always passed to the methods by value. Only the non-scalar variables like Object, Array, String are passed by reference.

Note:

Scalar variables are singular data with one value; Non scalar variables are data with multiple values.

Example:

```
Pass by value
class Swapper{
int a;
int b;
Swapper(int x, int y)  // constructor to initialize variables
{
a = x;
b = y;
}
void swap(int x, int y) // method to interchange values
{
            int temp;               /* only the local copy x, y gets swapped. The original object
            temp = x;               value a, b remains unchanged*/
            x=y;
            y=temp;
    }
}
class Main{
    public static void main(String[] args){
            Swapper obj = new Swapper(10, 20);  // create object
            System.out.println("Before swapping: a="+obj.a+" b="+obj.b);
            obj.swap(obj.a,obj.b); // call the method by passing class object as parameter
            System.out.println("Before swapping: a="+obj.a+" b="+obj.b);
    }
}
```

Sample Output:

```
Before swapping: a=10 b=20
After swapping: a=10 b=20
```

Here, to call method swap() first create an object for class Swapper. Then the method is called by passing object values *a* and *b* as input parameters. As these values are scalar, the parameters are passed using pass by value technique. So the changes carried out inside the method are not reflected in original value of *a* and *b*.

Pass by Reference

In pass-by-reference, reference (address) of the actual parameters is passed to the local parameters in the method definition. So, the changes performed on local parameters are reflected on the actual parameters.

Example:

```
class Swapper{
    int a;
    int b;
    Swapper(int x, int y)  // constructor to initialize variables
    {
        a = x;
        b = y;
    }
    void swap(Swapper ref) // method to interchange values
    {
        int temp;          /* Object is passed by reference. So the original object value
        temp = ref.a;      a, b gets changed*/
        ref.a = ref.b;
        ref.b = temp;
    }
}
class PassByRef{
    public static void main(String[] args){
        Swapper obj = new Swapper(10, 20);  // create object
        System.out.println(“Before swapping: a=”+obj.a+” b=”+obj.b);
```

```
            obj.swap(obj); // call the method by passing class object as parameter
            System.out.println(“After swapping: a=”+obj.a+” b=”+obj.b);
      }
   }
```

Sample Output:

```
   Before swapping: a=10 b=20
   After swapping: a=20 b=10
```

In this example, the class object is passed as parameter using pass by reference technique. So the method refers the original value of *a* and *b*.

Method using object as parameter and returning objects

A method can have object as input parameter (*see* pass by reference) and can return a class type object.

Example:

```
   class Addition{
      int no;
      Addition(){}
      Addition(int x){
         no=x;
      }
      public Addition display(Addition oa){
         Addition tmp=new Addition();      /*method with same class object as input parameter &
         tmp.no=no+oa.no;                  return value*/
         return(tmp);
      }
   }
   class Main{
      public static void main(String args[]){
         Addition a1=new Addition(10);
         Addition a2=new Addition(10);
         Addition a3;
        a3=a1.display(a2);  // method is invoked using the object a1 with input parameter a2
```

```
    System.out.println("a1.no="+a1.no+" a2.no="+a2.no+" a3.no="+a3.no);
  }
}
```

Sample Output:

```
a1.no=10 a2.no=10 a3.no=20
```

Here, display() accepts class Addition object a2 as input parameter. It also return same class object as output. This method adds the value of invoking object a1 and input parameter a2. The summation result is stored in temporary object tmp inside the method. The value returned by the method is received using object a3 inside main().

1.20 METHOD OVERLOADING

Method overloading is the process of having multiple methods with same name that differs in parameter list. The number and the data type of parameters must differ in overloaded methods. It is one of the ways to implement polymorphism in Java. When a method is called, the overloaded method whose parameters match with the arguments in the call gets invoked.

Note: Overloaded methods are differentiable only based on parameter list and not on their return type.

Example:

Program for addition using Method Overloading

```
class MethodOverload{
  void add(){
    System.out.println("No parameters");
  }
  void add(int a,int b){            // overloaded add() for two integer parameter
    System.out.println("Sum:"+(a+b));
  }
  void add(int a,int b,int c){      // overloaded add() for three integer parameter
    System.out.println("Sum:"+(a+b+c));
  }
  void add(double a,double b){      // overloaded add() for two double parameter
    System.out.println("Sum:"+(a+b));
    }
}
public class Main {
    public static void main(String[] args) {
        MethodOverload obj=new MethodOverload();
        obj.add();                    // call all versions of add()
        obj.add(1,2);
        obj.add(1,2,3);
        obj.add(12.3,23.4);
    }
}
```

Sample Output:

```
No parameters
Sum:3
Sum:6
Sum:35.7
```

Here, *add()* is overloaded four times. The first version takes no parameters, second takes two integers, third takes three integers and fourth accepts two double parameter.

1.21 ACCESS SPECIFIERS

Access specifiers or access modifiers in java specifies accessibility (scope) of a data member, method, constructor or class. It determines whether a data or method in a class can be used or invoked by other class or subclass.

Types of Access Specifiers

There are 4 types of java access specifiers:

1. Private
2. Default (no speciifer)
3. Protected
4. Public

The details about accessibility level for access specifiers are shown in following table.

Access Modifiers	Default	Private	Protected	Public
Accessible inside the class	Yes	Yes	Yes	Yes
Accessible within the subclass inside the same package	Yes	No	Yes	Yes
Accessible outside the package	No	No	No	Yes
Accessible within the subclass outside the package	No	No	Yes	Yes

Private access modifier

Private data fields and methods are accessible only inside the class where it is declared i.e accessible only by same class members. It provides low level of accessibility. Encapsulation and data hiding can be achieved using private specifier.

Example:

Role of private specifier

```
class PrivateEx{
   private int x;                      // private data
   public int y;                       // public data
   private PrivateEx(){}               // private constructor
   public PrivateEx(int a,int b){      // public constructor
      x=a;
      y=b;
   }
}
public class Main {
   public static void main(String[] args) {
      PrivateEx obj1=new PrivateEx();      // Error: private constructor cannot be applied
      PrivateEx obj2=new PrivateEx(10,20);  // public constructor can be applied to obj2
      System.out.println(obj2.y);           // public data y is accessible by a non-member
      System.out.println(obj2.x);             //Error: x has private access in PrivateEx
   }
}
```

In this example, we have created two classes PrivateEx and Main. A class contains private data member, private constructor and public method. We are accessing these private members from outside the class, so there is compile time error.

Default access modifier

If the specifier is mentioned, then it is treated as default. There is no default specifier keyword. Using default specifier we can access class, method, or field which belongs to same package, but not from outside this package.

Example:

Role of default specifier

```
class DefaultEx{
    int y=10; // default data
}
public class Main {
    public static void main(String[] args) {
        DefaultEx obj=new DefaultEx();
        System.out.println(obj.y);       // default data y is accessible outside the class
    }
}
```

Sample Output:

```
10
```

In the above example, the scope of class DefaultEx and its data *y* is default. So it can be accessible within the same package and cannot be accessed from outside the package.

Protected access modifier

Protected methods and fields are accessible within same class, subclass inside same package and subclass in other package (through inheritance). It cannot be applicable to class and interfaces.

Example:

Role of protected specifier

```
class Base{
   protected void show(){
      System.out.println(“In Base”);
   }
}
public class Main extends Base{
   public static void main(String[] args) {
      Main obj=new Main();
      obj.show();
   }
}
```

Sample Output:

```
In Base
```

In this example, *show()* of class Base is declared as protected, so it can be accessed from outside the class only through inheritance. Chapter 2 explains the concept of inheritance in detail.

Public access modifier

The public access specifier has highest level of accessibility. Methods, class, and fields declared as public are accessible by any class in the same package or in other package.

Example:

Role of public specifier

```
class PublicEx{
   public int no=10;
}
```

```
public class Main{
    public static void main(String[] args) {
        PublicEx obj=new PublicEx();
        System.out.println(obj.no);
    }
}
```

Sample Output:

```
10
```

In this example, public data *no* is accessible both by member and non-member of the class.

1.22 STATIC KEYWORD

The static keyword indicates that the member belongs to the class instead of a specific instance. It is used to create class variable and mainly used for memory management. The static keyword can be used with:

- Variable (static variable or class variable)
- Method (static method or class method)
- Block (static block)
- Nested class (static class)
- import (static import)

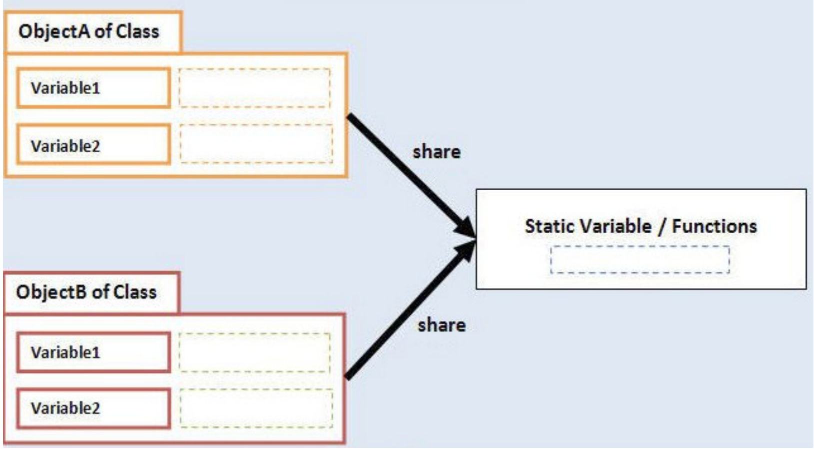
Static variable

Variable declared with keyword static is a static variable. It is a class level variable commonly shared by all objects of the class.

- Memory allocation for such variables only happens once when the class is loaded in the memory.
- scope of the static variable is class scope (accessible only inside the class)
- lifetime is global (memory is assigned till the class is removed by JVM).
- Automatically initialized to 0.
- It is accessible using ClassName.variablename
- Static variables can be accessed directly in static and non-static methods.

Example :

Without static	With static
class StaticEx{	class StaticEx{
int no=10;	static int no=10;
StaticEx(){	StaticEx(){
System.out.println(no);	System.out.println(no);
no++;	no++;
}	}
}	}
public class Main{	public class Main{
public static void main(String[] args)	public static void main(String[] args)
{	{
StaticEx obj1=new StaticEx();	StaticEx obj1=new StaticEx();
StaticEx obj2=new StaticEx();	StaticEx obj2=new StaticEx();
StaticEx obj3=new StaticEx();	StaticEx obj3=new StaticEx();
}	}
}	}
Sample Output:	*Sample Output:*
10	10
10	11
10	12



Static Method

The method declared with static keyword is known as static method. *main()* is most common static method.

- It belongs to the class and not to object of a class.
- A static method can directly access only static variables of class and directly invoke only static methods of the class.
- Static methods cannot access non-static members(instance variables or instance methods) of the class
- Static method cant access this and super references
- It can be called through the name of class without creating any instance of that class. For example, ClassName.methodName()

Example:

```
class StaticEx{
   static int x;
   int y=10;
   static void display(){
      System.out.println("Static Method "+x);   // static method accessing static variable
   }
   public void show(){
      System.out.println("Non static method "+y);
      System.out.println("Non static method "+x); // non-static method can access static variable
```

```
    }
}
public class Main
{
    public static void main(String[] args) {
            StaticEx obj=new StaticEx();
            StaticEx.display();      // static method invoked without using object
            obj.show();
    }
}
```

Sample Output:

```
Static Method 0
Non static method 10
Non static method 0
```

In this example, class StaticEx consists of a static variable x and static method display(). The static method cannot access a non-static variable. If you try to access y inside static method display(), it will result in compilation error.

```
static void display(){        →  /*non-static variable y cannot be referred from a
                                  static context*/
  System.out.println("Static Method "+x+y);
}
```

Static Block

A static block is a block of code enclosed in braces, preceded by the keyword *static*.

- The statements within the static block are first executed automatically before main when the class is loaded into JVM.
- A class can have any number of static blocks.
- JVM combines all the static blocks in a class as single block and executes them.
- Static methods can be invoked from the static block and they will be executed as and when the static block gets executed.

Syntax:

```
static{
```

```
                …………….
          }
```

Example:

```
class StaticBlockEx{
   StaticBlockEx (){
      System.out.println("Constructor");
   }
   static {
      System.out.println("First static block");
   }
   static void show(){
      System.out.println("Inside method");
   }
   static{
      System.out.println("Second static block");
      show();
   }

   public static void main(String[] args) {
               StaticBlockEx obj=new StaticBlockEx ();
     }
     static{
        System.out.println("Static in main");
     }
}
```

Sample Output:

```
First static block
Second static block
Inside method
```

```
Static in main
Constructor
```

Nested class (static class)

Nested class is a class declared inside another class. The inner class must be a static class declared using keyword static. The static nested class can refer directly to static members of the enclosing classes, even if those members are private.

Syntax:

```
class OuterClass{
            ……..
            static class InnerClass{
                  ……….
            }
      }
```

We can create object for static nested class directly without creating object for outer class. For example:

```
OuterClassName.InnerClassName=new OuterClassName.InnerClassName();
```

Example:

```
class Outer{
   static int x=10;
   static class Inner{
      int y=20;
      public void show(){
         System.out.println(x+y);          // nested class accessing its own data & outer class static data
      }
   }
}
class Main{
   public static void main(String args[]){
      Outer.Inner obj=new Outer.Inner();  // Creating object for static nested class
      obj.show();
   }
}
```

Sample Output:

```
30
```

Static Import

The static import allows the programmer to access any static members of imported class directly. There is no need to qualify it by its name.

Syntax:

```
Import static package_name;
```

Advantage:

- Less coding is required if you have access any static member of a class oftenly.

Disadvantage:

- Overuse of static import makes program unreadable and unmaintable.

Example:

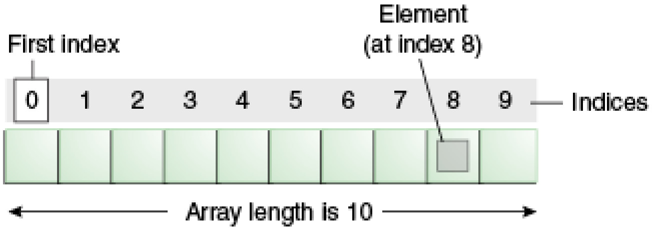
```
import static java.lang.System.*;
class StaticImportEx{
  public static void main(String args[]){
   out.println("Static Import Example");                //Now no need of System.out
}
}
```

Sample Output:

```
Static Import Example
```

1.23 ARRAYS

Array is a collection of elements of similar data type stored in contiguous memory location. The array size is fixed i.e we can't increase/decrease its size at runtime. It is index based and the first element is stored at 0th index.



Advantages of Array

- Code Optimization: Multiple values can be stored under common name. Date retrieval or sorting is an easy process.
- Random access: Data at any location can be retrieved randomly using the index.

Disadvantages of Array

- Inefficient memory usage: Array is static. It is not resizable at runtime based on number of user's input. To overcome this limitation, Java introduce collection concept.

Types of Array

There are two types of array.

- One Dimensional Array
- Multidimensional Array

One Dimensional Array

Declaring Array Variables

The syntax for declaring an array variable is

Syntax:

```
                    dataType[] arrayName; //preferred way
                              Or
                    dataType arrayName [];
```

Here datatype can be a primitive data type like: int, char, Double, byte etc. arrayName is an identifier.

Example:

```
          int[] a;
```

Instantiation of an Array

Array can be created using the *new* keyword. To allocate memory for array elements we must mention the array size. The size of an array must be specified by an *int* value and not *long* or *short*. The default initial value of elements of an array is 0 for numeric types and *false* for boolean.

Syntax:

```
 arrayName=new datatype[size];
              Or
dataType[] arrayName=new datatype[size]; //declaration and instantiation
```

Example:

```
int[] a=new int[5];        //defining an integer array for 5 elements
```

Alternatively, we can create and initialize array using following syntax.

Syntax:

```
dataType[] arrayName=new datatype[]{list of values separated by comma};
                    Or
dataType[] arrayName={ list of values separated by comma};
```

Example:

```
int[] a={12,13,14};
int[] a=new int[]{12,13,14};
```

The built-in *length* property is used to determine length of the array i.e. number of elements present in an array.

Accessing array elements

The array elements can be accessed by using indices. The index starts from 0 and ends at (array size-1). Each element in an array can be accessed using *for* loop.

Example:

Program to access array elements.

```
class Main{
   public static void main(String args[]){
      int a[]=new int[]{10,20,30,40};//declaration and initialization
      //printing array
      for(int i=0;i<a.length;i++)//length is the property of array
         System.out.println(a[i]);
   }
}
```

Sample Output:

```
10
20
30
40
```

The for-each loop

The for-each loop is used to traverse the complete array sequentially without using an index variable. It's commonly used to iterate over an array or a Collections class (eg, ArrayList).

Syntax:

```
for(type var:arrayName){
    Statements using var;
}
```

Example:

Program to calculate sum of array elements.

```
class Main{
  public static void main(String args[]){
    int a[]=new int[]{10,20,30,40};//declaration and initialization
    int sum=0;
    for(int i:a)      // calculate sum of array elements
      sum+=i;
    System.out.println("Sum:"+sum);
  }
}
```

Sample Output:

```
Sum:100
```

Multidimensional Arrays

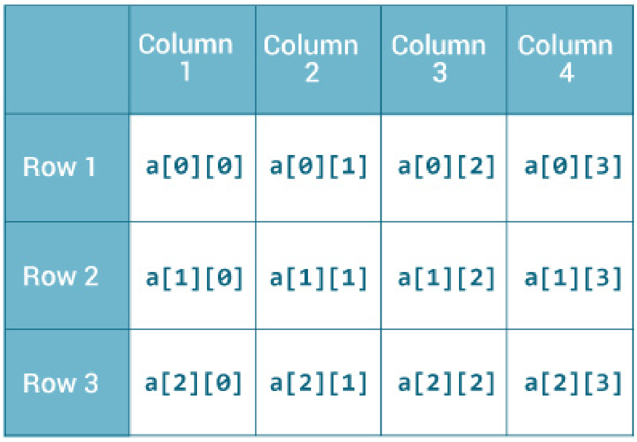
Multidimensional arrays are arrays of arrays with each element of the array holding the reference of other array. These are also known as Jagged Arrays.

Syntax:

```
dataType[][] arrayName=new datatype[rowsize][columnnsize];    // 2 dimensional array
dataType[][][] arrayName=new datatype[][][];                  // 3 dimensional array
```

Example:

```
int[][] a=new int[3][4];
```



	Column 1	Column 2	Column 3	Column 4
Row 1	a[0][0]	a[0][1]	a[0][2]	a[0][3]
Row 2	a[1][0]	a[1][1]	a[1][2]	a[1][3]
Row 3	a[2][0]	a[2][1]	a[2][2]	a[2][3]

Example:

Program to access 2D array elements

```
class TwoDimEx
{
   public static void main(String args[])
   {
      // declaring and initializing 2D array
      int arr[][] = { {1,1,12},{2,16,1},{12,42,2} };
       // printing 2D array
      for (int i=0; i< arr.length; i++)
      {
         for (int j=0; j < arr[i].length ; j++)
            System.out.print(arr[i][j] + “ “);
          System.out.println();
      }
   }
}
```

Sample Output:

```
1 1 12
2 16 1
12 42 2
```

Jagged Array

Jagged array is an array of arrays with different row size i.e. with different dimensions.

Example:

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

    int[][] a = {
        {11, 3, 43},
        {3, 5, 8, 1},
        {9},
    };

    System.out.println(“Length of row 1: “ + a[0].length);
    System.out.println(“Length of row 2: “ + a[1].length);
    System.out.println(“Length of row 3: “ + a[2].length);
  }
}
```

Sample Output:

```
Length of row 1: 3
Length of row 2: 4
Length of row 3: 1
```

Passing an array to a method

An array can be passed as parameter to method.

Example:

Program to find minimum element in an array

```
class Main{
   static void min(int a[]){
      int min=a[0];
      for(int i=1;i<a.length;i++)
         if(min>a[i])
            min=a[i];
```

```
        System.out.println("Minimum:"+min);
    }
    public static void main(String args[]){
        int a[]={12,13,14,5};
        min(a);//passing array to method
    }
}
```

Sample Output:

```
Minimum:5
```

Returning an array from a method

A method may also return an array.

Example:

Program to sort array elements in ascending order.

```
class Main{
    static int[] sortArray(int a[]){
        int tmp;
        for(int i=0;i<a.length-1;i++)  {            //code for sorting
            for(int j=i+1;j<=a.length-1;j++)  {
                if(a[i]>a[j]){
                    tmp=a[i];
                    a[i]=a[j];
                    a[j]=tmp;
                }
            }
        }
        return(a);        // returning array
    }
    public static void main(String args[]){
        int a[]={33,43,24,5};
        a=sortArray(a);//passing array to method
```

UNIT -2

INHERITANCE AND INTERFACES

2.1 INHERITANCE

Inheritance is the mechanism in java by which one class is allow to inherit the features (fields and methods) of another class. It is process of deriving a new class from an existing class. A class that is inherited is called a *superclass* and the class that does the inheriting is called a *subclass.* Inheritance represents the IS-A relationship, also known as *parent-child* relationship. The keyword used for inheritance is extends.

Syntax:

```
class Subclass-name extends Superclass-name
{
 //methods and fields
}
```

Here, the extends keyword indicates that we are creating a new class that derives from an existing class.

Note: The constructors of the superclass are never inherited by the subclass

Advantages of Inheritance:

- Code reusability - public methods of base class can be reused in derived classes
- Data hiding – private data of base class cannot be altered by derived class
- Overriding--With inheritance, we will be able to override the methods of the base class in the derived class

Example:

```
// Create a superclass.
class BaseClass{
   int a=10,b=20;
   public void add(){
      System.out.println(“Sum:”+(a+b));
```

```
    }
}
// Create a subclass by extending class BaseClass.
public class Main extends BaseClass
{
    public void sub(){
        System.out.println("Difference:"+(a-b));
    }
      public static void main(String[] args) {
          Main obj=new Main();
/*The subclass has access to all public members of its superclass*/
          obj.add();
          obj.sub();
      }
}
```

Sample Output:

```
Sum:30
Difference:-10
```

In this example, Main is the subclass and BaseClass is the superclass. Main object can access the field of own class as well as of BaseClass class i.e. code reusability.

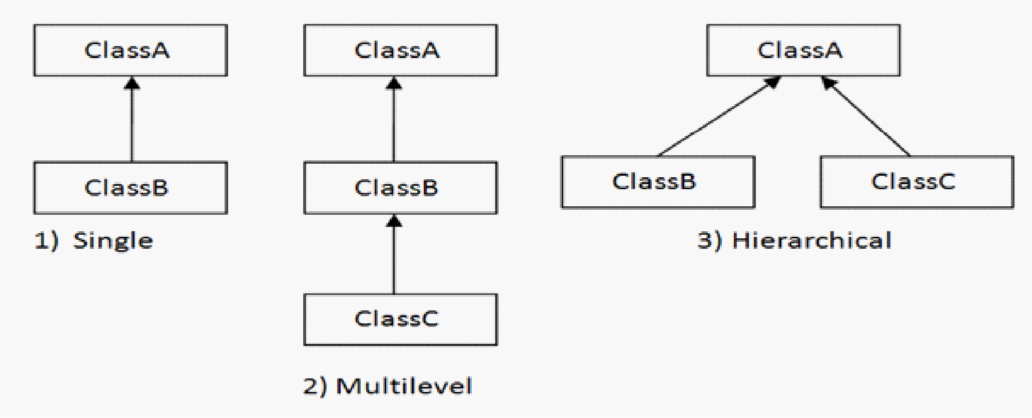
Types of inheritance

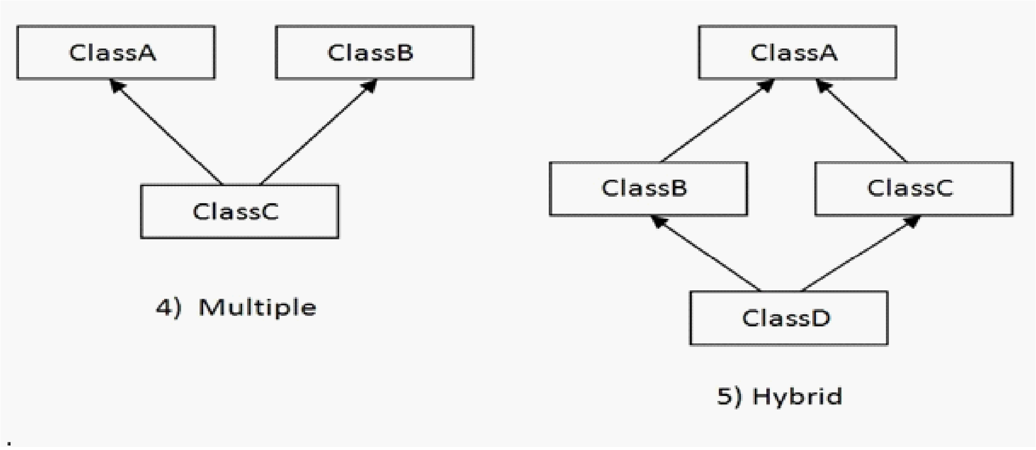
Single Inheritance :

In single inheritance, a subclass inherit the features of one superclass.

Example:

```
class Shape{
    int a=10,b=20;
}
class Rectangle extends Shape{
    public void rectArea(){
        System.out.println("Rectangle Area:"+(a*b));
    }
}
public class Main
{
      public static void main(String[] args) {
          Rectangle obj=new Rectangle();
                  obj.rectArea();
      }
}
```





Multilevel Inheritance:

In Multilevel Inheritance, a derived class will be inheriting a base class and as well as the derived class also act as the base class to other class i.e. a derived class in turn acts as a base class for another class.

Example:

```
class Numbers{
   int a=10,b=20;
}
class Add2 extends Numbers{
   int c=30;
   public void sum2(){
      System.out.println("Sum of 2 nos.:"+(a+b));
   }
}
class Add3 extends Add2{
   public void sum3(){
      System.out.println("Sum of 3 nos.:"+(a+b+c));
   }
}
public class Main
{
     public static void main(String[] args) {
        Add3 obj=new Add3();
              obj.sum2();
              obj.sum3();
     }
}
```

Sample Output:

```
Sum of 2 nos.:30
Sum of 3 nos.:60
```

Hierarchical Inheritance:

In Hierarchical Inheritance, one class serves as a superclass (base class) for more than one sub class.

Example:

```
class Shape{
   int a=10,b=20;
}
class Rectangle extends Shape{
   public void rectArea(){
      System.out.println(“Rectangle Area:”+(a*b));
   }
}
class Triangle extends Shape{
   public void triArea(){
      System.out.println(“Triangle Area:”+(0.5*a*b));
   }
}
public class Main
{
     public static void main(String[] args) {
        Rectangle obj=new Rectangle();
              obj.rectArea();
        Triangle obj1=new Triangle();
              obj1.triArea();
     }
}
```

Sample Output:

```
Rectangle Area:200
Triangle Area:100.0
```

Multiple inheritance

Java does not allow multiple inheritance:

- To reduce the complexity and simplify the language
- To avoid the ambiguity caused by multiple inheritance

For example, Consider a class C derived from two base classes A and B. Class C inherits A and B features. If A and B have a method with same signature, there will be ambiguity to call method of A or B class. It will result in compile time error.

```
class A{
void msg(){System.out.println("Class A");}
}
class B{
void msg(){System.out.println("Class B ");}
}
class C extends A,B{//suppose if it were
  Public Static void main(String args[]){
  C obj=new C();
  obj.msg();//Now which msg() method would be invoked?
}
}
```

Sample Output:

```
                Compile time error
```

Direct implementation of multiple inheritance is not allowed in Java. But it is achievable using Interfaces. The concept about interface is discussed in chapter.2.7.

Access Control in Inheritance

The following rules for inherited methods are enforced −

- Variables declared public or protected in a superclass are inheritable in subclasses.
- Variables or Methods declared private in a superclass are not inherited at all.
- Methods declared public in a superclass also must be public in all subclasses.
- Methods declared protected in a superclass must either be protected or public in subclasses; they cannot be private.

Example:

```
// Create a superclass
class A{
   int x;              // default specifier
   private int y;      // private to A
```

```
    public void set_xy(int a,int b){
        x=a;
        y=b;
    }
}
// A’s y is not accessible here.
class B extends A{
    public void add(){
        System.out.println(“Sum:”+(x+y)); //Error: y has private access in A – not inheritable
    }
}
class Main{
    public static void main(String args[]){
        B obj=new B();
        obj.set_xy(10,20);
        obj.add();
    }
}
```

In this example since y is declared as private, it is only accessible by its own class members. Subclasses have no access to it.

2.2 USING SUPER

The super keyword refers to immediate parent class object. Whenever you create the instance of subclass, an instance of parent class is created implicitly which is referred by super reference variable.

- It an be used to refer immediate parent class instance variable when both parent and child class have member with same name
- It can be used to invoke immediate parent class method when child class has overridden that method.
- super() can be used to invoke immediate parent class constructor.

Use of super with variables:

When both parent and child class have member with same name, we can use super keyword to access mamber of parent class.

Example:

```
class SuperCls
{
    int x = 20;
}
 /* sub class  SubCls extending SuperCls */
class SubCls extends SuperCls
{
    int x = 80;
     void display()
    {
        System.out.println("Super Class x: " + super.x); //print x of super class
        System.out.println("Sub Class x: " + x);  //print x of subclass
    }
}
 /* Driver program to test */
class Main
{
    public static void main(String[] args)
    {
        SubCls obj = new SubCls();
        obj.display();
    }
}
```

Sample Output:

```
Super Class x: 20
Sub Class x: 80
```

In the above example, both base class and subclass have a member x. We could access x of base class in sublcass using super keyword.

Use of super with methods:

The super keyword can also be used to invoke parent class method. It should be used if subclass contains the same method as parent class (Method Overriding).

```
class SuperCls
{
   int x = 20;
   void display(){   //display() in super class
      System.out.println("Super Class x: " + x);
   }
}
 /* sub class  SubCls extending SuperCls */
class SubCls extends SuperCls
{
   int x = 80;
    void display()    //display() redefined in sub class – method overriding
   {
      System.out.println("Sub Class x: " + x);
      super.display();          // invoke super class display()
   }
}
 /* Driver program to test */
class Main
{
   public static void main(String[] args)
   {
      SubCls obj = new SubCls();
      obj.display();
   }
}
```

Sample Output:

```
Sub Class x: 80
Super Class x: 20
```

In the above example, if we only call method display() then, the display() of sub class gets invoked. But with the use of super keyword, display() of superclass could also be invoked.

Use of super with constructors:

The super keyword can also be used to invoke the parent class constructor.

Syntax:

```
super();
```

- super() if present, must always be the first statement executed inside a subclass constructor.
- When we invoke a super() statement from within a subclass constructor, we are invoking the immediate super class constructor

Example:

```
class SuperCls
{
   SuperCls(){
      System.out.println("In Super Constructor");
   }
}
 /* sub class  SubCls extending SuperCls */
class SubCls extends SuperCls
{
   SubCls(){
      super();
      System.out.println("In Sub Constructor");
   }
}
 /* Driver program to test */
class Main
{
   public static void main(String[] args)
   {
        SubCls obj = new SubCls();
    }
}
```

Sample Output:

```
In Super Constructor
In Sub Constructor
```

2.3 ORDER OF CONSTRUCTOR INVOCATION

- Constructors are invoked in the order of their derivation
- If a subclass constructor does not explicitly invoke a superclass constructor using super() in the first line, the Java compiler automatically inserts a call to the no-argument constructor of the superclass. If the superclass does not have a no-argument constructor, it will generate a compile-time error.

Example:

```
class A
{
    A(){
        System.out.println("A's Constructor");
    }
}
 /* sub class  B extending A */
class B extends A
{
    B(){
        super();
        System.out.println("B's Constructor");
    }
}
/* sub class  C extending B */
class C extends B{
    C(){
        super();
        System.out.println("C's Constructor");
    }
}
 /* Driver program to test */
class Main
{
    public static void main(String[] args)
    {
        C obj = new C();
    }
}
```

Sample Output:

```
A's Constructor
B's Constructor
C's Constructor
```

Invoking Superclass Parameterized Constructor

To call parameterized constructor of superclass, we must use the super keyword as shown below.

Syntax:

```
        super(value);
```

Example:

```
class SuperCls{
    int x;
    SuperCls(int x){
        this.x=x;                    // this refers to current invoking object
    }
}
class SubCls extends SuperCls{
    int y;
    SubCls(int x,int y){
        super(x);                    // invoking parameterized constructor of superclass
        this.y=y;
    }
    public void display(){
        System.out.println("x: "+x+" y: "+y);
    }
}
public class Main
{
    public static void main(String[] args) {
        SubCls obj=new SubCls(10,20);
        obj.display();
    }
}
```

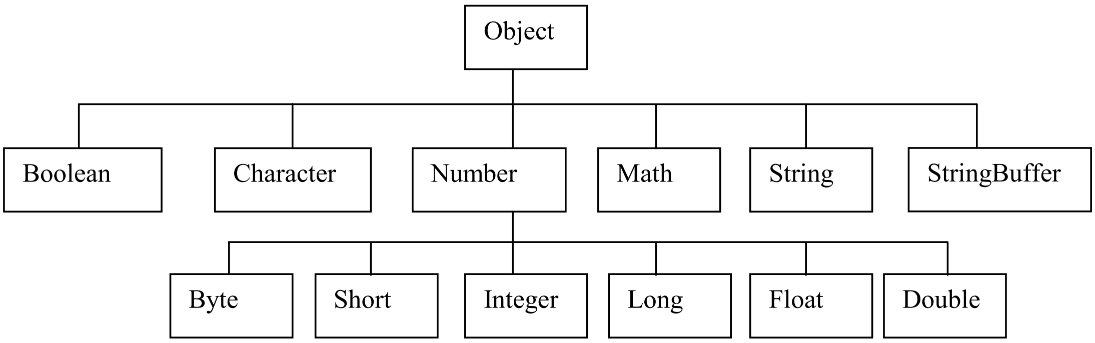
Sample Output:

```
x: 10 y: 20
```

The program contains a superclass and a subclass, where the superclass contains a parameterized constructor which accepts a integer value, and we used the super keyword to invoke the parameterized constructor of the superclass.

2.4 THE OBJECT CLASS

The Object class is the parent class of all the classes in java by default (directly or indirectly). The java.lang.Object class is the root of the class hierarchy. Some of the Object class are Boolean, Math, Number, String etc.



Some of the important methods defined in Object class are listed below.

Object class Methods	**Description**
boolean equals(Object)	Returns true if two references point to the same object.
String toString()	Converts object to String
void notify() void notifyAll() void wait()	Used in synchronizing threads
void finalize()	Called just before an object is garbage collected
Object clone()	Returns a new object that are exactly the same as the current object
int hashCode()	Returns a hash code value for the object.

Example:

```
public class Test
{
   public static void main(String[] args)
   {
                                          /*hashcode is the unique number generated by JVM*/
      Test t = new Test();
      System.out.println(t);
      System.out.println(t.toString());         // provides String representation of an Object
      System.out.println(t.hashCode());
      t = null;                      /*calling garbage collector explicitly to dispose system resources, perform
      System.gc();                   clean-up activities and minimize memory leaks*/
      System.out.println(“end”);
   }
   protected void finalize()  // finalize() is called just once on an object
   {
      System.out.println(“finalize method called”);
   }
}
```

Sample Output:

```
Test@2a139a55
Test@2a139a55
705927765
end
finalize method called
```

In the above program, the default toString() method for class Object returns a string consisting of the name of the class Test of which the object is an instance, the at-sign character `@', and the unsigned hexadecimal representation of the hash code of the object.

2.5 ABSTRACT CLASSES AND METHODS

Abstract class

A class that is declared as abstract is known as **abstract class**. It can have abstract and non-abstract methods (method with body). It needs to be extended and its method implemented. It cannot be instantiated.

Syntax:

```
abstract class classname
{
}
```

Abstract method

A method that is declared as abstract and does not have implementation is known as abstract method. The method body will be defined by its subclass.

Abstract method can never be final and static. Any class that extends an abstract class must implement all the abstract methods declared by the super class.

Note:

A normal class (non-abstract class) cannot have abstract methods.

Syntax:

```
abstract returntype functionname (); //No definition
```

Syntax for abstract class and method:

```
modifier abstract class className
{
   //declare fields
   //declare methods
   abstract dataType methodName();
}
modifier class childClass extends className
{
dataType methodName()
{
}
}
```

Why do we need an abstract class?

Consider a class Animal that has a method sound() and the subclasses of it like Dog Lion, Horse Cat etc. Since the animal sound differs from one animal to another, there is no point to implement this method in parent class. This is because every child class must override this method to give its own implementation details, like `Lion` class will say “Roar” in this method and Horse class will say “Neigh”.

So when we know that all the animal child classes will and should override this method, then there is no point to implement this method in parent class. Thus, making this method abstract would be the good choice. This makes this method abstract and all the subclasses to implement this method. We need not give any implementation to this method in parent class.

Since the Animal class has an abstract method, it must be declared as abstract.

Now each animal must have a sound, by making this method abstract we made it compulsory to the child class to give implementation details to this method. This way we ensure that every animal has a sound.

Rules

1. Abstract classes are not Interfaces.
2. An abstract class may have concrete (complete) methods.
3. An abstract class may or may not have an abstract method. But if any class has one or more abstract methods, it must be compulsorily labeled abstract.
4. Abstract classes can have Constructors, Member variables and Normal methods.
5. Abstract classes are never instantiated.
6. For design purpose, a class can be declared abstract even if it does not contain any abstract methods.
7. Reference of an abstract class can point to objects of its sub-classes thereby achieving run-time polymorphism Ex: Shape obj = new Rectangle();

8. A class derived from the abstract class must implement all those methods that are declared as abstract in the parent class.
9. If a child does not implement all the abstract methods of abstract parent class, then the child class must need to be declared abstract as well.

Example 1

```
//abstract parent class
abstract class Animal
{
  //abstract method
  public abstract void sound();
}
//Lion class extends Animal class
public class Lion extends Animal
{
  public void sound()
{
     System.out.println(“Roars”);
  }
  public static void main(String args[])
{
     Animal obj = new Lion();
     obj.sound();
  }
}
```

Output:

```
Roars
```

In the above code, Animal is an abstract class and Lion is a concrete class.

Example 2

```
abstract class Bank
{
abstract int getRateOfInterest();
}
class SBI extends Bank
{
int getRateOfInterest()
{
   return 7;
}
}
class PNB extends Bank
{
int getRateOfInterest()
{
   return 8;
}
}
public class TestBank
{
public static void main(String args[])
{
Bank b=new SBI();//if object is PNB, method of PNB will be invoked
int interest=b.getRateOfInterest();
System.out.println(“Rate of Interest is: “+interest+” %”);
b=new PNB();
System.out.println(“Rate of Interest is: “+b.getRateOfInterest()+” %”);
}
}
```

Output:

```
Rate of Interest is: 7 %
Rate of Interest is: 8 %
```

Abstract class with concrete (normal) method

Abstract classes can also have normal methods with definitions, along with abstract methods.

Sample Code:

```
abstract class A
{
 abstract void callme();
 public void normal()
 {
  System.out.println("this is a normal (concrete) method.");
 }
}
public class B extends A
{
 void callme()
 {
  System.out.println("this is an callme (abstract) method.");
 }
 public static void main(String[] args)
 {
  B b = new B();
  b.callme();
  b.normal();
 }
}
```

Output:

```
this is an callme (abstract) method.
this is a normal (concrete) method.
```

Observations about abstract classes in Java

1. **An instance of an abstract class cannot be created; But, we can have references of abstract class type though.**

Sample Code:

```
abstract class Base
{
   abstract void fun();
}
class Derived extends Base
{
   void fun()
{
System.out.println(“Derived fun() called”);
}
}
public class Main
{
   public static void main(String args[])
{
        // Base b = new Base(); Will lead to error
        // We can have references of Base type.
       Base b = new Derived();
       b.fun();
   }
}
```

Output:

```
Derived fun() called
```

2. An abstract class can contain constructors in Java. And a constructor of abstract class is called when an instance of a inherited class is created.

Sample Code:

```
abstract class Base
{
    Base()
    {
        System.out.println("Within Base Constructor");
    }
    abstract void fun();
}
class Derived extends Base
{
    Derived()
    {
        System.out.println("Within Derived Constructor");
    }
    void fun()
    {
        System.out.println(" Within Derived fun()");
    }
}
public class Main
{
    public static void main(String args[])
    {
       Derived d = new Derived();
    }
}
```

Output:

```
Within Base Constructor
Within Derived Constructor
```

3. **We can have an abstract class without any abstract method. This allows us to create classes that cannot be instantiated, but can only be inherited.**

Sample Code:

```
abstract class Base
{
   void fun()
   {
      System.out.println("Within Base fun()");
   }
}
class Derived extends Base
{
}
public class Main
{
   public static void main(String args[])
   {
      Derived d = new Derived();
      d.fun();
   }
}
```

Output:

```
Within Base fun()
```

4. **Abstract classes can also have final methods (methods that cannot be overridden).**

Sample Code:

```
abstract class Base
{
   final void fun()
   {
        System.out.println("Within Derived fun()");
    }
}
class Derived extends Base
{
}
public class Main
{
    public static void main(String args[])
    {
        Base b = new Derived();
        b.fun();
    }
}
```

Output:

```
Within Derived fun()
```

2.6 FINAL METHODS AND CLASSES

The final keyword in java is used to restrict the user. The java final keyword can be applied to:

- variable
- method
- class

Java final variable	-	To prevent constant variables
Java final method	-	To prevent method overriding
Java final class	-	To prevent inheritance

Figure: Uses of final in java

Java final variable

The final keyword can be applied with the variables, a final variable that have no value it is called blank final variable or uninitialized final variable. It can be initialized in the constructor only. The blank final variable can be static also which will be initialized in the static block only.

Sample Code:

A final variable speedlimit is defined within a class Vehicle. When we try to change the value of this variable, we get an error. This is due to the fact that the value of final variable cannot be changed, once a value is assigned to it.

```
public class Vehicle
{
 final int speedlimit=60;//final variable
 void run()
 {
  speedlimit=400;
 }
 public static void main(String args[])
 {
 Vehicle obj=new  Vehicle();
 obj.run();
 }
}
```

Output:

```
/Vehicle.java:6: error: cannot assign a value to final variable speedlimit
  speedlimit=400;
  ^
1 error
```

Blank final variable

A final variable that is not initialized at the time of declaration is known as blank final variable. We must initialize the blank final variable in constructor of the class otherwise it will throw a compilation error.

Sample Code:

```
public class Vehicle
{
 final int speedlimit; //blank final variable
 void run()
 {
 }
 public static void main(String args[])
 {
 Vehicle obj=new  Vehicle();
 obj.run();
 }
}
```

Output:

```
/Vehicle.java:3: error: variable speedlimit not initialized in the default constructor
 final int speedlimit; //blank final variable
           ^
1 error
```

Java Final Method

A Java method with the final keyword is called a final method and it cannot be overridden in the subclass.

In general, final methods are faster than non-final methods because they are not required to be resolved during run-time and they are bonded at compile time.

Sample Code:

```
class XYZ
{
  final void demo()
  {
    System.out.println("XYZ Class Method");
  }
}
public class ABC extends XYZ
{
  void demo()
  {
    System.out.println("ABC Class Method");
  }
  public static void main(String args[])
  {
    ABC obj= new ABC();
    obj.demo();
  }
}
```

Output:

```
/ABC.java:11: error: demo() in ABC cannot override demo() in XYZ
  void demo()
      ^
 overridden method is final
1 error
```

The following code will run fine as the final method demo() is not overridden. This shows that final methods are inherited but they cannot be overridden.

Sample Code:

```
class XYZ
{
  final void demo()
  {
    System.out.println("XYZ Class Method");
  }
}
public class ABC extends XYZ
{
  public static void main(String args[])
  {
    ABC obj= new ABC();
    obj.demo();
  }
}
```

Output:

```
    XYZ Class Method
```

Points to be remembered while using final methods:

- Private methods of the superclass are automatically considered to be final.
- Since the compiler knows that final methods cannot be overridden by a subclass, so these methods can sometimes provide performance enhancement by removing calls to final methods and replacing them with the expanded code of their declarations at each method call location.
- Methods made inline should be small and contain only few lines of code. If it grows in size, the execution time benefits become a very costly affair.
- A final's method declaration can never change, so all subclasses use the same method implementation and call to one can be resolved at compile time. This is known as *static binding.*

Java Final Class

- Final class is a class that cannot be extended i.e. it cannot be inherited.
- A final class can be a subclass but not a superclass.
- Declaring a class as final implicitly declares all of its methods as final.
- It is illegal to declare a class as both abstract and final since an abstract class is incomplete by itself and relies upon its subclasses to provide complete implementations.
- Several classes in Java are final e.g. String, Integer, and other wrapper classes.
- The final keyword can be placed either before or after the access specifier.

Syntax:

```
final public class A

{

   //code

}
```

OR

```
public final class A

{

   //code

}
```

Sample Code:

```
final class XYZ
{
}

public class ABC extends XYZ
{
  void demo()
  {
    System.out.println("My Method");
  }
  public static void main(String args[])
  {
    ABC obj= new ABC();
    obj.demo();
  }
}
```

Output:

```
/ABC.java:5: error: cannot inherit from final XYZ
public class ABC extends XYZ
                         ^
1 error
```

Important points on final in Java

- Final keyword can be applied to a member variable, local variable, method or class in Java.
- Final member variable must be initialized at the time of declaration or inside the constructor, failure to do so will result in compilation error.
- We cannot reassign value to a final variable in Java.
- The local final variable must be initialized during declaration.
- A final method cannot be overridden in Java.
- A final class cannot be inheritable in Java.

- Final is a different than finally keyword which is used to Exception handling in Java.
- Final should not be confused with finalize() method which is declared in Object class and called before an object is a garbage collected by JVM.
- All variable declared inside Java interface are implicitly final.
- Final and abstract are two opposite keyword and a final class cannot be abstract in Java.
- Final methods are bonded during compile time also called static binding.
- Final variables which are not initialized during declaration are called blank final variable and must be initialized in all constructor either explicitly or by calling this(). Failure to do so compiler will complain as “final variable (name) might not be initialized”.
- Making a class, method or variable final in Java helps to improve performance because JVM gets an opportunity to make assumption and optimization.

2.7 INTERFACES

An interface is a reference type in Java. It is similar to class. It is a collection of abstract methods. Along with abstract methods, an interface may also contain constants, default methods, static methods, and nested types. Method bodies exist only for default methods and static methods.

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 byte code 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.

Uses of interface:

- Since java does not support multiple inheritance in case of class, it can be achieved by using interface.
- It is also used to achieve loose coupling.
- Interfaces are used to implement abstraction.

Defining an Interface

An interface is defined much like a class.

Syntax:

```
accessspecifier interface interfacename
{
return-type method-name1(parameter-list);
return-type method-name2(parameter-list);
type final-varname1 = value;
type final-varname2 = value;
// ...
return-type method-nameN(parameter-list);
type final-varnameN = value;
}
```

When no access specifier is included, then default access results, and the interface is only available to other members of the package in which it is declared. When it is declared as public, the interface can be used by any other code.

- The java file must have the same name as the interface.
- The methods that are declared have no bodies. They end with a semicolon after the parameter list. They are abstract methods; there can be no default implementation of any method specified within an interface.
- Each class that includes an interface must implement all of the methods.
- Variables can be declared inside of interface declarations. They are implicitly final and static, meaning they cannot be changed by the implementing class. They must also be initialized.
- All methods and variables are implicitly public.

Sample Code:

The following code declares a simple interface Animal that contains two methods called eat() and travel() that take no parameter.

```
/* File name : Animal.java */
interface Animal {
  public void eat();
  public void travel();
}
```

Implementing an Interface

Once an interface has been defined, one or more classes can implement that interface. To implement an interface, the ‘implements’ clause is included in a class definition and then the methods defined by the interface are created.

Syntax:

```
class classname [extends superclass] [implements interface [,interface...]]
{
// class-body
}
```

Properties of java interface

- If a class implements more than one interface, the interfaces are separated with a comma.
- If a class implements two interfaces that declare the same method, then the same method will be used by clients of either interface.
- The methods that implement an interface must be declared public.
- The type signature of the implementing method must match exactly the type signature specified in the interface definition.

Rules

- A class can implement more than one interface at a time.
- A class can extend only one class, but can implement many interfaces.
- An interface can extend another interface, in a similar way as a class can extend another class.

Sample Code 1:

The following code implements an interface Animal shown earlier.

```
/* File name : MammalInt.java */
public class Mammal implements Animal
{
  public void eat()
{
    System.out.println(“Mammal eats”);
  }
  public void travel()
{
    System.out.println("Mammal travels");
  }
  public int noOfLegs()
{
    return 0;
  }
  public static void main(String args[])
{
    Mammal m = new Mammal();
    m.eat();
    m.travel();
  }
}
```

Output:

```
Mammal eats
Mammal travels
```

It is both permissible and common for classes that implement interfaces to define additional members of their own. In the above code, Mammal class defines additional method called noOfLegs().

Sample Code 2:

The following code initially defines an interface 'Sample' with two members. This interface is implemented by a class named 'testClass'.

```
import java.io.*;
// A simple interface
interface Sample
{
   final String name = "Shree";
   void display();
}
```

```
 // A class that implements interface.
public class testClass implements Sample
{
   public void display()
   {
      System.out.println("Welcome");
   }
   public static void main (String[] args)
   {
      testClass t = new testClass();
      t.display();
      System.out.println(name);
   }
}
```

Output:

```
Welcome
Shree
```

Sample Code 3:

In this example, Drawable interface has only one method. Its implementation is provided by Rectangle and Circle classes.

```
interface Drawable
{
void draw();
}
class Rectangle implements Drawable
{
public void draw()
{
   System.out.println("Drawing rectangle");
}
}
class Circle implements Drawable
{
public void draw()
{
   System.out.println(“Drawing circle”);
}
}
public class TestInterface
{
public static void main(String args[])
{
Drawable d=new Circle();
d.draw();
}
}
```

Output:

```
Drawing circle
```

Nested Interface

An interface can be declared as a member of a class or another interface. Such an interface is called a member interface or a nested interface. A nested interface can be declared as public, private, or protected.

Sample Code:

```
interface MyInterfaceA
{
   void display();
   interface MyInterfaceB
   {
      void myMethod();
   }
}
public class NestedInterfaceDemo1 implements MyInterfaceA.MyInterfaceB
{
    public void myMethod()
    {
        System.out.println("Nested interface method");
    }
    public static void main(String args[])
    {
        MyInterfaceA.MyInterfaceB obj= new NestedInterfaceDemo1();
     obj.myMethod();
    }
}
```

Output:

```
Nested interface method
```

Differences between classes and interfaces

Both classes and Interfaces are used to create new reference types. A class is a collection of fields and methods that operate on fields. A class creates reference types and these reference types are used to create objects. A class has a signature and a body. The syntax of class declaration is shown below:

```
class class_Name extends superclass implements interface_1,….interface_n
    // class signature
    {
    //body of class.
    }
```

Signature of a class has class's name and information that tells whether the class has inherited another class. The body of a class has fields and methods that operate on those fields. A Class is created using a keyword **class**.

When a class is instantiated, each object created contains a copy of fields and methods with them. The fields and members declared inside a class can be static or nonstatic. Static members value is constant for each object whereas, the non-static members are initialized by each object differently according to its requirement.

Members of a class have access specifiers that decide the visibility and accessibility of the members to the user or to the subclasses. The access specifiers are public, private and protected. A class can be inherited by another class using the access specifier which will decide the visibility of members of a superclass (inherited class) in a subclass (inheriting class).

An interface has fully abstract methods (methods with nobody). An interface is syntactically similar to the class but there is a major difference between class and interface that is a class can be instantiated, but an interface can never be instantiated.

An interface is used to create the reference types. The importance of an interface in Java is that, a class can inherit only a single class. To circumvent this restriction, the designers of Java introduced a concept of interface. An interface declaration is syntactically similar to a class, but there is no field declaration in interface and the methods inside an interface do not have any implementation. An interface is declared using a keyword **interface**.

Aspect for comparison	Class	Interface
Basic	A class is instantiated to create objects.	An interface can never be instantiated as the methods are unable to perform any action on invoking.
Keyword	class	Interface
Access specifier	The members of a class can be private, public or protected.	The members of an interface are always public.
Methods	The methods of a class are defined to perform a specific action.	The methods in an interface are purely abstract.
inheritance	A class can implement any number of interfaces and can extend only one class.	An interface can extend multiple interfaces but cannot implement any interface.
Inheritance keyword	extends	implements
Constructor	A class can have constructors to initialize the variables.	An interface can never have a constructor as there is hardly any variable to initialize.

Declaration Syntax	class class_Name
	{
	//fields
	//Methods
	}
		Interface interface_Name
		{
		Type var_name=value;
		Type method1(parameter-list);
		Type method2(parameter-list);
		..
		}

The following example shows that a class that implements one interface:

```
public interface interface_example
{
public void method1();
public string method2();
}
public class class_name implements interface_example
{
public void method1()
{
..
}
public string method2()
{
…
}
}
```

Inheritance between concrete (non-abstract) and abstract classes use extends keyword.

It is possible to extend only one class to another. Java does not support multiple inheritance. However, multilevel inheritance i.e., any number of classes in a ladder is possible. For example, in the following code class C extends the class B, where the class B extends class A.

```
class A {}
class B extends A { }
class C extends B { }
```

Inheritance between classes (including abstract classes) and interfaces, use implements keyword.

To support multiple inheritance, it uses interfaces. So after implements keyword, there can be any number of interfaces. For example, in the following code, class B extends only one class A and two interfaces I1 and I2.

```
interface I1 {}
interface I2 {}
class A
class B extends A implements I1, I2
{
}
```

Inheritance between two interfaces, is possible with the use of extends keyword only.

For example, in the following code, interface I2 extends the interface I1.

```
interface I1 { }
interface I2 extends I1{ }
```

2.8 OBJECT CLONING

Object cloning refers to creation of exact copy of an object. It creates a new instance of the class of current object and initializes all its fields with exactly the contents of the corresponding fields of this object. In Java, there is no operator to create copy of an object. Unlike C++, in Java, if we use assignment operator then it will create a copy of reference variable and not the object. This can be explained by taking an example. Following program demonstrates the same.

// Java program to demonstrate that assignment operator creates a new reference to same object.

```
import java.io.*;
class sample
{
    int a;
    float b;
    sample()
    {
            a = 10;
            b = 20;
    }
}
class Mainclass
{
    public static void main(String[] args)
    {
            sample ob1 = new sample();
            System.out.println(ob1.a + “ “ + ob1.b);
            sample ob2 = ob1;
            ob2.a = 100;
            System.out.println(ob1.a+” “+ob1.b);
            System.out.println(ob2.a+” “+ob2.b);
    }
}
```

Output:

```
    10 20.0
    100 20.0
    100 20.0
```

Creating a copy using clone() method

The class whose object’s copy is to be made must have a public clone method in it or in one of its parent class.

- Every class that implements clone() should call super.clone() to obtain the cloned object reference.
- The class must also implement java.lang.Cloneable interface whose object clone we want to create otherwise it will throw CloneNotSupportedException when clone method is called on that class’s object.
- Syntax:
- protected Object clone() throws CloneNotSupportedException

```
    import java.util.ArrayList;
    class sample1
    {
        int a, b;
    }
    class sample2 implements Cloneable
    {
        int c;
        int d;
        sample1 s = new sample1();
        public Object clone() throws CloneNotSupportedException
        {
                return super.clone();
        }
    }
    public class Mainclass
    {
        public static void main(String args[]) throws CloneNotSupportedException
        {
        sample2 ob1 = new sample2();
        ob1.c = 10;
        ob1.d = 20;
        ob1.s.a = 30;
        ob1.s.b = 40;
        sample2 ob2 = (sample2)ob1.clone();
        ob2.d = 100; //Change in primitive type of ob2 will not be reflected in ob1 field
        ob2.s.a = 300; //Change in object type field will be reflected in both ob2 and
ob1(shallow copy)
        System.out.println(ob1.c + “ “ + ob1.d + “ “ +ob1.s.a + “ “ + ob1.s.b);
        System.out.println(ob2.c + “ “ + ob2.d + “ “ +ob2.s.a + “ “ + ob2.s.b);
        }
    }
```

Types of Object cloning

1. Deep Copy
2. Shallow Copy

Shallow copy

Shallow copy is method of copying an object. It is the default in cloning. In this method the fields of an old object ob1 are copied to the new object ob2. While copying the object type field the reference is copied to ob2 i.e. object ob2 will point to same location as pointed out by ob1. If the field value is a primitive type it copies the value of the primitive type. So, any changes made in referenced objects will be reflected in other object.

Note:

Shallow copies are cheap and simple to make.

Deep Copy

To create a deep copy of object ob1 and place it in a new object ob2 then new copy of any referenced objects fields are created and these references are placed in object ob2. This means any changes made in referenced object fields in object ob1 or ob2 will be reflected only in that object and not in the other. A deep copy copies all fields, and makes copies of dynamically allocated memory pointed to by the fields. A deep copy occurs when an object is copied along with the objects to which it refers.

```
//Java program for deep copy using clone()
import java.util.ArrayList;
    class Test
    {
        int a, b;
    }
    class Test2 implements Cloneable
    {
        int c, d;
        Test ob1 = new Test();
        public Object clone() throws CloneNotSupportedException
        {
                // Assign the shallow copy to new refernce variable t
                Test2 t1 = (Test2)super.clone();
                t1.ob1 = new Test();
                // Create a new object for the field c
                // and assign it to shallow copy obtained,
                // to make it a deep copy
                return t1;
        }
}
public class Main
{
        public static void main(String args[]) throws CloneNotSupportedException
        {
        Test2 t2 = new Test2();
        t2.c = 10;
        t2.d = 20;
        t2.ob1.a = 30;
        t2.ob1.b = 40;
        Test2 t3 = (Test2)t2.clone();
        t3.c = 100;
        t3.ob1.a = 300;
        System.out.println (t2.c + " " + t2.d + " " + t2.ob1.a + " " + t2.ob1.b);
        System.out.println (t3.c + " " + t3.d + " " + t3.ob1.a + " " + t3.ob1.b);
        }
}
```

Output

```
10 20 30 40
100 20 300 0
```

Advantages of clone method:

- If we use assignment operator to assign an object reference to another reference variable then it will point to same address location of the old object and no new copy of the object will be created. Due to this any changes in reference variable will be reflected in original object.
- If we use copy constructor, then we have to copy all of the data over explicitly i.e. we have to reassign all the fields of the class in constructor explicitly. But in clone method this work of creating a new copy is done by the method itself. So to avoid extra processing we use object cloning.

2.9 NESTED CLASSES

In Java, a class can have another class as its member. The class written within another class is called the nested class, and the class that holds the inner class is called the outer class.

Java inner class is defined inside the body of another class. Java inner class can be declared private, public, protected, or with default access whereas an outer class can have only public or default access. The syntax of nested class is shown below:

```
class Outer_Demo {
  class Nested_Demo {
     }
   }
```

Types of Nested classes

There are two types of nested classes in java. They are non-static and static nested classes. The non-static nested classes are also known as inner classes.

- Non-static nested class (inner class)
    - Member inner class
    - Method Local inner class
    - Anonymous inner class
- Static nested class

Type	Description
Member Inner Class	A class created within class and outside method.
Anonymous Inner Class	A class created for implementing interface or extending class. Its name is decided by the java compiler.
Method Local Inner Class	A class created within method.
Static Nested Class	A static class created within class.
Nested Interface	An interface created within class or interface.

2.10 INNER CLASSES (NON-STATIC NESTED CLASSES)

Inner classes can be used as the security mechanism in Java. Normally, a class cannot be related with the access specifier **private.** However if a class is defined as a member of other class, then the inner class can be made private. This class can have access to the private members of a class.

The three types of inner classes are

- Member Inner Class
- Method-local Inner Class
- Anonymous Inner Class

Member Inner Class

The Member inner class is a class written within another class. Unlike a class, an inner class can be private and once you declare an inner class private, it cannot be accessed from an object outside the class.

The following program is an example for member inner class.

```
 class Outer_class {
   int n=20;
   private class Inner_class {
   public void display() {
   System.out.println("This is an inner class");
   System.out.println("n:"+n);
      }
   }
     void print_inner() {
      Inner_class inn = new Inner_class();
      inn.display();
   }
}
   public class Myclass {
   public static void main(String args[]) {
     Outer_class out= new Outer_class();
     out.print_inner();
   }
}
```

Output:

```
This is an inner class
```

Method-local Inner Class

In Java, a class can be written within a method. Like local variables of the method, the scope of the inner class is restricted within the method. A method-local inner class can be instantiated only within the method where the inner class is defined. The following program shows how to use a method-local inner class. The following program is an example for Method-local Inner Class

```
public class Outer_class {
    void Method1() {
         int n = 100;
        class MethodInner_class {
        public void display() {
            System.out.println(“This is method inner class “);
            System.out.println(“n:”+n);
         }
      }
      MethodInner_class inn= new MethodInner_class();
      inn.display();
   }
   public static void main(String args[]) {
      Outer_class out = new Outer_class();
      out.Method1();
   }
}
```

```
Output:
This is method inner class
n: 100
```

Anonymous Inner Class

An inner class declared without a class name is known as an anonymous inner class. The anonymous inner classes can be created and instantiated at the same time. Generally, they are used whenever you need to override the method of a class or an interface. The syntax of an anonymous inner class is as follows –

```
abstract class Anonymous_Inner {
  public abstract void Method1();
}
```

The following program is an example for anonymous inner class.

```
public class Outer_class {
  public static void main(String args[]) {
    Anonymous_Inner inn = new Anonymous_Inner() {
      public void Method1() {
        System.out.println("This is the anonymous inner class");
      }
    };
    inn.Method1();
  }
}
```

Output:

```
This is the anonymous inner class
```

Static Nested Class

A static inner class is a nested class which is a static member of the outer class. It can be accessed without instantiating the outer class, using other static members. Just like static members, a static nested class does not have access to the instance variables and methods of the outer class. Instantiating a static nested class is different from instantiating an inner class. The following program shows how to use a static nested class.

```
public class Outer_class {
  static class inner_class{
    public void Method1() {
      System.out.println("This is the nested class");
    }
  }
    public static void main(String args[]) {
    Outer_class.inner_class obj = new Outer_class.inner_class();
    obj.Method1();
  }
}
```

Output:

```
This is the nested class
```

Advantage of java inner classes:

There are basically three advantages of inner classes in java. They are as follows:

- Nested classes represent a special type of relationship that is it can access all the members of outer class including private.
- Nested classes are used to develop more readable and maintainable code because it logically group classes and interfaces in one place only.
- It provides code optimization. That is it requires less code to write.

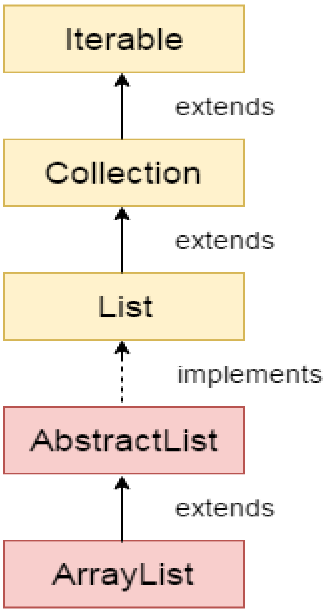
2.11 ARRAYLIST

ArrayList is a part of collection framework. It is present in java.util package. It provides us dynamic arrays in Java. Though, it may be slower than standard arrays but can be helpful in programs where lots of manipulation in the array is needed.

- ArrayList inherits AbstractList class and implements List interface.
- ArrayList is initialized by a size; however the size can increase if collection grows or shrink if objects are removed from the collection.
- Java ArrayList allows us to randomly access the list.
- ArrayList cannot be used for primitive types, like int, char, etc.
- ArrayList in Java is much similar to vector in C++.

Java ArrayList class

Java ArrayList class extends AbstractList class which implements List interface. The List interface extends Collection and Iterable interfaces in hierarchical order.



Java ArrayList class uses a dynamic array for storing the elements. It inherits AbstractList class and implements List interface.

The important points about Java ArrayList class are:

- Java ArrayList class can contain duplicate elements.
- Java ArrayList class maintains insertion order.
- Java ArrayList class is non synchronized.
- Java ArrayList allows random access because array works at the index basis.
- In Java ArrayList class, manipulation is slow because a lot of shifting needs to be occurred if any element is removed from the array list.

ArrayList class declaration

public class ArrayList<E> ***extends*** AbstractList<E> ***implements*** List<E>, RandomAccess, Cloneable, Serializable

Constructors of Java ArrayList

Constructor	Description
ArrayList()	It is used to build an empty array list.
ArrayList(Collection c)	It is used to build an array list that is initialized with the elements of the collection c.
ArrayList(int capacity)	It is used to build an array list that has the specified initial capacity.

Methods of Java ArrayList

Method	Description
void add(int index, Object element)	It is used to insert the specified element at the specified position index in a list.
boolean addAll (Collection c)	It is used to append all of the elements in the specified collection to the end of this list, in the order that they are returned by the specified collection's iterator.
void clear()	It is used to remove all of the elements from this list.
int lastIndexOf(Object o)	It is used to return the index in this list of the last occurrence of the specified element, or -1 if the list does not contain this element.
Object[] toArray()	It is used to return an array containing all of the elements in this list in the correct order.
Object[] toArray (Object[] a)	It is used to return an array containing all of the elements in this list in the correct order.
boolean add(Object o)	It is used to append the specified element to the end of a list.
boolean addAll(int index, Collection c)	It is used to insert all of the elements in the specified collection into this list, starting at the specified position.
Object clone()	It is used to return a shallow copy of an ArrayList.
int indexOf(Object o)	It is used to return the index in this list of the first occurrence of the specified element, or -1 if the List does not contain this element.
void trimToSize()	It is used to trim the capacity of this ArrayList instance to be the list's current size.

```
import java.util.*;
class Arraylist_example{
 public static void main(String args[]){
  ArrayList<String> a1=new ArrayList<String>();
  a1.add("Bala");
  a1.add("Mala");
  a1.add("Vijay");
  ArrayList<String> a2=new ArrayList<String>();
  a2.add("kala");
  a2.add("Banu");
  a1.addAll(a2);
  Iterator itr=a1.iterator();
  while(itr.hasNext()){
   System.out.println(itr.next());
  }
 }
}
```

2.12 JAVA STRING

In general string is a sequence of characters. String is an object that represents a sequence of characters. The java.lang.String class is used to create string object. In java, string is basically an object that represents sequence of char values. An array of characters works same as java string. For example:

Java String class provides a lot of methods to perform operations on string such as compare(), concat(), equals(), split(), length(), replace(), compareTo(), intern(), substring() etc.

The java.lang.String class implements *Serializable*, *Comparable* and *CharSequence* interfaces. The CharSequence interface is used to represent sequence of characters. It is implemented by String, StringBuffer and StringBuilder classes. It means can create string in java by using these 3 classes.

The string objects can be created using two ways.

1. By String literal
2. By new Keyword

String Literal

Java String literal is created by using double quotes. For Example:

1. String s=”welcome”;

Each time you create a string literal, the JVM checks the string constant pool first. If the string already exists in the pool, a reference to the pooled instance is returned. If string doesn’t exist in the pool, a new string instance is created and placed in the pool. For example:

String s1=”Welcome”;

String s2=”Welcome”;

In the above example only one object will be created. Firstly JVM will not find any string object with the value “Welcome” in string constant pool, so it will create a new object. After that it will find the string with the value “Welcome” in the pool, it will not create new object but will return the reference to the same instance. To make Java more memory efficient (because no new objects are created if it exists already in string constant pool).

2. By new keyword

String s=**new** String(“Welcome”);

In such case, JVM will create a new string object in normal (non pool) heap memory and the literal “Welcome” will be placed in the string constant pool. The variable s will refer to the object in heap (non pool).

The java String is immutable i.e. it cannot be changed. Whenever we change any string, a new instance is created. For mutable string, you can use StringBuffer and StringBuilder classes.

The following program explains the creation of strings

```
public class String_Example{
public static void main(String args[]){
String s1=”java”;
char c[]={‘s’,’t’,’r’,’i’,’n’,’g’};
String s2=new String(c);
String s3=new String(“example”);
System.out.println(s1);
System.out.println(s2);
System.out.println(s3);
}}
```

Java String class methods

The java.lang.String class provides many useful methods to perform operations on sequence of char values.

Method	Description
char charAt(int index)	returns char value for the particular index
int length()	returns string length
static String format(String format, Object... args)	returns formatted string
static String format(Locale l, String format, Object... args)	returns formatted string with given locale
String substring(int beginIndex)	returns substring for given begin index
String substring(int beginIndex, int endIndex)	returns substring for given begin index and end index
boolean contains(CharSequence s)	returns true or false after matching the sequence of char value

static String join(CharSequence delimiter, CharSequence... elements)	returns a joined string
boolean equals(Object another)	checks the equality of string with object
boolean isEmpty()	checks if string is empty
String concat(String str)	concatinates specified string
String replace(char old, char new)	replaces all occurrences of specified char value
String replace(CharSequence old, CharSequence new)	replaces all occurrences of specified CharSequence
static String equalsIgnoreCase(String another)	compares another string. It doesn’t check case.
String[] split(String regex)	returns splitted string matching regex
String[] split(String regex, int limit)	returns splitted string matching regex and limit
String intern()	returns interned string
int indexOf(int ch)	returns specified char value index
int indexOf(int ch, int fromIndex)	returns specified char value index starting with given index
int indexOf(String substring)	returns specified substring index
int indexOf(String substring, int fromIndex)	returns specified substring index starting with given index
String toLowerCase()	returns string in lowercase.
String toLowerCase(Locale l)	returns string in lowercase using specified locale.
String toUpperCase()	returns string in uppercase.
String toUpperCase(Locale l)	returns string in uppercase using specified locale.
String trim()	removes beginning and ending spaces of this string.
static String valueOf(int value)	converts given type into string. It is overloaded

The following program is an example for String concat function:

```
class string_method{
 public static void main(String args[]){
   String s=”Java”;
```

```
    s=s.concat(" Programming");
    System.out.println(s);
  }
 }
```

Output:

```
Java Programming
```

UNIT - 3

EXCEPTION HANDLING AND I/O

3.1 EXCEPTIONS

An exception is an unexpected event, which may occur during the execution of a program (at run time), to disrupt the normal flow of the program's instructions. This leads to the abnormal termination of the program, which is not always recommended.

Therefore, these exceptions are needed to be handled. The exception handling in java is one of the powerful mechanisms to handle the runtime errors so that normal flow of the application can be maintained.

An exception may occur due to the following reasons. They are.

- Invalid data as input.
- Network connection may be disturbed in the middle of communications
- JVM may run out of memory.
- File cannot be found/opened.

These exceptions are caused by user error, programmer error, and physical resources.

Based on these, the exceptions can be classified into three categories.

- ***Checked exceptions*** − A checked exception is an exception that occurs at the compile time, also called as compile time (static time) exceptions. These exceptions cannot be ignored at the time of compilation. So, the programmer should handle these exceptions.
- ***Unchecked exceptions*** − An unchecked exception is an exception that occurs at run time, also called as **Runtime Exceptions**. These include programming bugs, such as logic errors or improper use of an API. Runtime exceptions are ignored at the time of compilation.
- ***Errors*** − Errors are not exceptions, but problems may 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.

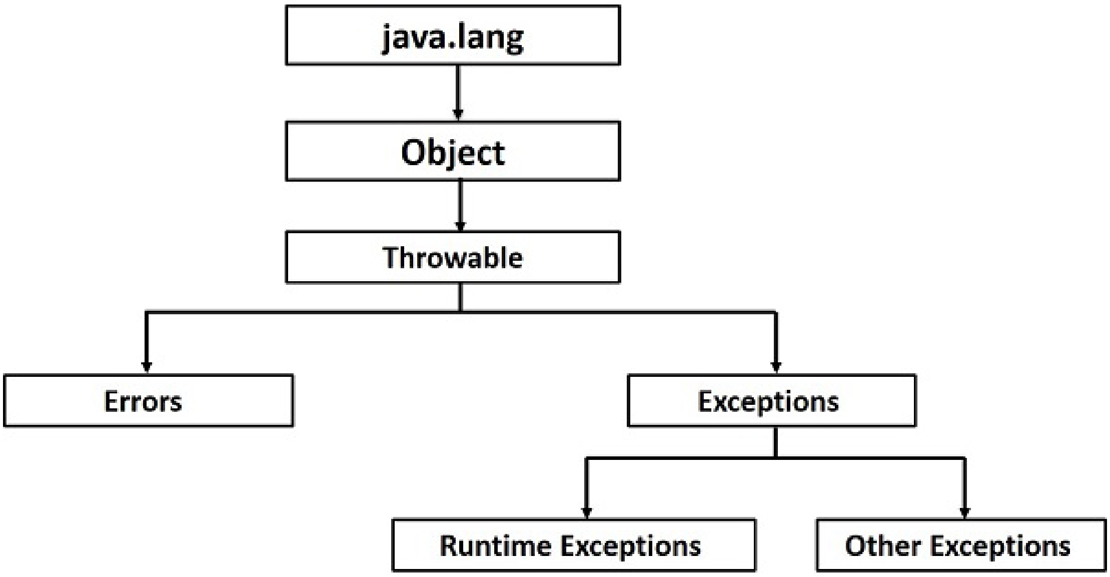
- ***Error:*** An Error indicates serious problem that a reasonable application should not try to catch.
- ***Exception:*** Exception indicates conditions that a reasonable application might try to catch.

3.2 EXCEPTION HIERARCHY

The java.lang.Exception class is the base class for all exception classes. All exception and errors types are sub classes of class Throwab**le**, which is base class of hierarchy. One branch is headed by Exception. This class is used for exceptional conditions that user programs should catch. NullPointerException is an example of such an exception. Another branch, Error are used by the Java run-time system(JVM) to indicate errors having to do with the run-time environment itself(JRE). StackOverflowError is an example of such an error.

Errors are abnormal conditions that happen in case of severe failures, these 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.



Exceptions Methods

Method	Description
public String getMessage()	Returns a detailed message about the exception that has occurred. This message is initialized in the Throwable constructor.
public Throwable getCause()	Returns the cause of the exception as represented by a Throwable object.
public String toString()	Returns the name of the class concatenated with the result of getMessage().
public void printStackTrace()	Prints the result of toString() along with the stack trace to System.err, the error output stream.
public StackTraceElement [] getStackTrace()	Returns an array containing each element on the stack trace. The element at index 0 represents the top of the call stack, and the last element in the array represents the method at the bottom of the call stack.
public Throwable fillInStackTrace()	Fills the stack trace of this Throwable object with the current stack trace, adding to any previous information in the stack trace.

Exception handling in java uses the following Keywords

1. try
2. catch
3. finally
4. throw
5. throws

The try/catch block is used as follows:

```
try {
// block of code to monitor for errors
// the code you think can raise an exception
}
catch (ExceptionType1 exOb) {
// exception handler for ExceptionType1
}
catch (ExceptionType2 exOb) {
```

```
// exception handler for ExceptionType
}
// optional
finally {
// block of code to be executed after try block ends
}
```

Throwing and catching exceptions

Catching Exceptions

A method catches an exception using a combination of the **try** and **catch** keywords. The program code that may generate an exception should be placed inside the try/catch block. The syntax for try/catch is depicted as below−

Syntax

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

The code which is prone to exceptions is placed in the try block. When an exception occurs, that exception is handled by catch block associated with it. Every try block should be immediately followed either by a catch block or finally block.

A catch statement involves declaring the type of exception that might be tried to catch. If an exception occurs, then the catch block (or blocks) which follow the try block is checked. If the type of exception that occurred is listed in a catch block, the exception is passed to the catch block similar to an argument that is passed into a method parameter.

To illustrate the try-catch blocks the following program is developed.

```
class  Exception_example {
public static void main(String args[])
{
 int  a,b;
try { // monitor a block of code.
 a = 0;
 b = 10 / a; //raises the arithmetic exception
System.out.println(“Try block.”);
}
 catch (ArithmeticException e)
{ // catch divide-by-zero error
System.out.println(“Division by zero.”);
 }
 System.out.println(“After try/catch block.”);
 }
 }
```

Output:

```
Division by zero.
After try/catch block.
```

Multiple catch Clauses

In some cases, more than one exception could be raised by a single piece of code. To handle this multiple exceptions, two or more catch clauses can be specified. Here, each catch block catches different type of exception. When an exception is thrown, each catch statement is inspected in order, and the first one whose type matches that of the exception is executed. After one catch statement executes, the others are bypassed, and execution continues after the try/catch block. The following example traps two different exception types:

```
class MultiCatch_Example {
 public static void main(String args[])  {
try  {
int a,b;
a = args.length;
System.out.println(“a = “ + a);
b = 10 / a;  //may cause division-by-zero error
int arr[] = { 10,20 };
 c[5] =100;
}
catch(ArithmeticException e)
{
```

```
System.out.println("Divide by 0: " + e);
}
 catch(ArrayIndexOutOfBoundsException e)
{
 System.out.println("Array index oob: " + e);
}
 System.out.println("After try/catch blocks.");
 }
 }
```

Here is the output generated by the execution of the program in both ways:

```
C:\>java MultiCatch_Example
a = 0
Divide by 0: java.lang.ArithmeticException: / by zero
 After try/catch blocks.
C:\>java MultiCatch_Example arg1
 a = 1
 Array index oob: java.lang.ArrayIndexOutOfBoundsException:5
After try/catch blocks.
```

While the multiple catch statements is used, it is important to remember that exception subclasses must come before their superclasses. A catch statement which uses a superclass will catch exceptions of that type plus any of its subclasses. Thus, a subclass would never be reached if it came after its superclass. And also, in Java, unreachable code is an error. For example, consider the following program:

```
class MultiCatch_Example {
 public static void main(String args[])  {
try  {
int a,b;
a = args.length;
System.out.println("a = " + a);
b = 10 / a;  //may cause division-by-zero error
int arr[] = { 10,20 };
```

```
 c[5] =100;
}
catch(Exception e) {
System.out.println(“Generic Exception catch.”);
}
catch(ArithmeticException e)
{
System.out.println(“Divide by 0: “ + e);
}
 catch(ArrayIndexOutOfBoundsException e)
{
 System.out.println(“Array index oob: “ + e);
}
 System.out.println(“After try/catch blocks.”);
 }
 }
```

The exceptions such as ArithmeticException, and ArrayIndexOutOfBoundsException are the subclasses of Exception class. The catch statement after the base class catch statement is raising the unreachable code exception.

Nested try block

Sometimes a situation may arise where a part of a block may cause one error and the entire block itself may cause another error. In such cases, exception handlers have to be nested.

```
try
{
   statement 1;
   statement 2;
   try
   {
      statement 1;
      statement 2;
   }
```

```
    catch(Exception e)
    {
    }
}
catch(Exception e)
{
}
....
```

The following program is an example for Nested try statements.

```
class Nestedtry_Example{
 public static void main(String args[]){
  try{
    try{
     System.out.println(“division”);
     int a,b;
     a=0;
     b =10/a;
    }
    catch(ArithmeticException e)
   {
   System.out.println(e);
  }
  try
  {
   int a[]=new int[5];
   a[6]=3;
   }
   catch(ArrayIndexOutOfBoundsException e)
   {
     System.out.println(e);
   }
   System.out.println("other statement);
  }
  catch(Exception e)
 {
 System.out.println("handeled");}
 System.out.println("normal flow..");
 }
}
```

Throw keyword

The Java throw keyword is used to explicitly throw an exception. The general form of throw is shown below:

```
throw ThrowableInstance;
```

Here, ThrowableInstance must be an object of type Throwable or a subclass of Throwable. Primitive types, such as int or char, as well as non-Throwable classes, such as String and Object, cannot be used as exceptions.

There are two ways to obtain a Throwable object:

1. using a parameter in a catch clause
2. creating one with the new operator.

The following program explains the use of throw keyword.

```
public class TestThrow1{
static void validate(int age){
 try{
     if(age<18)
      throw new ArithmeticException("not valid");
     else
      System.out.println("welcome to vote");
   }
    Catch(ArithmeticException e)
 {
   System.out.println(“Caught inside ArithmeticExceptions.”);
  throw e; // rethrow the exception
 }
 }
   public static void main(String args[]){
try{
validate(13);
}
Catch(ArithmeticException e)
 {
   System.out.println(“ReCaught ArithmeticExceptions.”);
 }
}
}
```

The flow of execution stops immediately after the throw statement and any subsequent statements that are not executed. The nearest enclosing try block is inspected to see if it has a catch statement that matches the type of exception. If it does find a match, control is transferred to that statement. If not, then the next enclosing try statement is inspected, and so on. If no matching catch is found, then the default exception handler halts the program and prints the stack trace.

The Throws/Throw Keywords

If a method does not handle a checked exception, the method must be declared using the throws keyword. The throws keyword appears at the end of a method’s signature.

The difference between throws and throw keywords is that, *throws* is used to postpone the handling of a checked exception and *throw* is used to invoke an exception explicitly.

The following method declares that it throws a Remote Exception −

Example

```
import java.io.*;
public class throw_Example1 {
public void function(int a) 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 ArithmeticException −

```
import java.io.*;
public class throw_Example2 {
public void function(int a) throws RemoteException,ArithmeticException  {
// Method implementation
  }
 // Remainder of class definition
}
```

The Finally Block

The finally block follows a try block or a catch block. A finally block of code always executes, irrespective of the occurrence of an Exception. A finally block appears at the end of the catch blocks that follows the below syntax.

Syntax

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

Example

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

```
int a,b;
        a=0;
       b=10/a;
   } catch (ArithmeticException e) {
   System.out.println("Exception thrown  :" + e);
  }finally {
   System.out.println("The finally block is executed");
   }
 }
 }
```

Points to remember:

- A catch clause cannot exist without a try statement.
- It is not compulsory to have finally clauses whenever a try/catch block is present.
- The try block cannot be present without either catch clause or finally clause.
- Any code cannot be present in between the try, catch, finally blocks.

3.3 BUILT-IN EXCEPTIONS

Built-in exceptions are the exceptions which are available in Java libraries. These exceptions are suitable to explain certain error situations. Below is the list of important built-in exceptions in Java.

Exceptions	Description
Arithmetic Exception	It is thrown when an exceptional condition has occurred in an arithmetic operation.
Array Index Out Of Bound Exception	It is thrown to indicate that an array has been accessed with an illegal index. The index is either negative or greater than or equal to the size of the array.
ClassNotFoundException	This Exception is raised when we try to access a class whose definition is not found.
FileNotFoundException	This Exception is raised when a file is not accessible or does not open.
IOException	It is thrown when an input-output operation failed or interrupted.
InterruptedException	It is thrown when a thread is waiting, sleeping, or doing some processing, and it is interrupted.

NoSuchFieldException	It is thrown when a class does not contain the field (or variable) specified.
NoSuchMethodException	It is thrown when accessing a method which is not found.
NullPointerException	This exception is raised when referring to the members of a null object. Null represents nothing.
NumberFormatException	This exception is raised when a method could not convert a string into a numeric format.
RuntimeException	This represents any exception which occurs during runtime.
StringIndexOutOfBoundsException	It is thrown by String class methods to indicate that an index is either negative than the size of the string

The following Java program explains NumberFormatException

```
class  NumberFormat_Example
{
   public static void main(String args[])
   {
      try {
         int num = Integer.parseInt (“hello”) ;
          System.out.println(num);
      }
      catch(NumberFormatException e) {
         System.out.println(“Number format exception”);
      }
   }
}
```

The following Java program explains StackOverflowError exception.

```
class Example {
public static void main(String[] args)
     {
              fun1();
     }
public static void fun1()
    {
            fun2();
    }
public static void fun2()
    {
            fun1();
    }
}
```

Output:

```
Exception in thread “main” java.lang.StackOverflowError
at Example.fun2(File.java:14)
at Example.fun1(File.java:10)
```

3.4 USER DEFINED EXCEPTION IN JAVA

Java allows the user to create their own exception class which is derived from built-in class Exception. The Exception class inherits all the methods from the class Throwable. The `Throwable` class is the superclass of all errors and exceptions in the Java language. It contains a snapshot of the execution stack of its thread at the time it was created. It can also contain a message string that gives more information about the error.

- The Exception class is defined in java.lang package.
- User defined exception class must inherit Exception class.
- The user defined exception can be thrown using throw keyword.

Syntax:

```
class User_defined_name extends Exception{
        ………..
        }
```

Some of the methods defined by Throwable are shown in below table.

Methods	Description
Throwable fillInStackTrace( )	Fills in the execution stack trace and returns a Throwable object.
String getLocalizedMessage()	Returns a localized description of the exception.
String getMessage()	Returns a description of the exception.

void printStackTrace( )	Displays the stack trace.
String toString( )	Returns a String object containing a description of the Exception.
StackTraceElement[ ]get StackTrace( )	Returns an array that contains the stack trace, one element at a time, as an array of StackTraceElement.

Two commonly used constructors of Exception class are:

- Exception() - Constructs a new exception with null as its detail message.
- Exception(String message) - Constructs a new exception with the specified detail message.

Example:

```
//creating a user-defined exception class derived from Exception class
public class MyException extends Exception
{
public String toString(){                // overriding toString() method
   return “User-Defined Exception”;
}
public static void main(String args[]){
              MyException obj= new MyException();
              try
              {
                        throw new MyException();      // customized exception  is raised
     }
                                              /*Printing object e makes a call to toString() method which
              catch(MyException e)
                                              returns String error message*/
              {
               System.out.println(“Exception handled - “+ e);
              }
}
}
```

Sample Output:

```
Exception handled - User-Defined Exception
```

In the above example, a custom defined exception class MyException is created by inheriting it from Exception class. The toString() method is overridden to display the customized method on catch. The MyException is raised using the throw keyword.

Example:

Program to create user defined exception that test for odd numbers.

```
import java.util.Scanner;
class OddNumberException extends Exception
{
        OddNumberException()          //default constructor
        {
            super(“Odd number exception”);
        }
        OddNumberException(String msg)     //parameterized constructor
        {
            super(msg);
        }
}
public class UserdefinedExceptionDemo{
        public static void main(String[] args)
        {
            int num;
            Scanner Sc = new Scanner(System.in); // create Scanner object to read input
            System.out.println(“Enter a number : “);
            num = Integer.parseInt(Sc.nextLine());
            try
            {
                if(num%2 != 0)   // test for odd number
                   throw(new OddNumberException());  // raise the exception if number is odd
                else
                    System.out.println(num + “ is an even number”);
            }
            catch(OddNumberException Ex)
            {
               System.out.print("\n\tError : " + Ex.getMessage());
            }
         }
      }
```

Sample Output1:

```
   Enter a number : 11
   Error : Odd number exception
```

Sample Output2:

```
   10 is an even number
```

Odd Number Exception class is derived from the Exception class. To implement user defined exception we need to throw an exception object explicitly. In the above example, If the value of num variable is odd, then the throw keyword will raise the user defined exception and the catch block will get execute.

3.5 CHAINED EXCEPTIONS

Chained Exceptions allows to relate one exception with another exception, i.e one exception describes cause of another exception. For example, consider a situation in which a method throws an ArithmeticException because of an attempt to divide by zero but the actual cause of exception was an I/O error which caused the divisor to be zero. The method will throw only ArithmeticException to the caller. So the caller would not come to know about the actual cause of exception. Chained Exception is used in such type of situations.

Throwable constructors that supports chained exceptions are:

1. Throwable(Throwable cause) :- Where cause is the exception that causes the current exception.
2. Throwable(String msg, Throwable cause) :- Where msg is the exception message and cause is the exception that causes the current exception.

Throwable methods that supports chained exceptions are:

1. getCause() method :- This method returns actual cause of an exception.
2. initCause(Throwable cause) method :- This method sets the cause for the calling exception.

Example:

```
import java.io.IOException;
public class ChainedException
 {
  public static void divide(int a, int b)
   {
    if(b==0)
    {
     ArithmeticException ae = new ArithmeticException(“top layer”);
     ae.initCause( new IOException(“cause”) );
     throw ae;
    }
    else
    {
     System.out.println(a/b);
    }
   }
 public static void main(String[] args)
 {
  try {
    divide(5, 0);
   }
  catch(ArithmeticException ae) {
    System.out.println( “caught : “ +ae);
    System.out.println(“actual cause: “+ae.getCause());
   }
 }
}
```

Sample Output:

```
caught : java.lang.ArithmeticException: top layer
actual cause: java.io.IOException: cause
```

In this example, the top-level exception is ArithmeticException. To it is added a cause exception, IOException. When the exception is thrown out of divide( ), it is caught by main( ). There, the top-level exception is displayed, followed by the underlying exception, which is obtained by calling getCause( ).

3.6 STACK TRACE ELEMENT

The StackTraceElement class element represents a single stack frame which is a stack trace when an exception occurs. Extracting stack trace from an exception could provide useful information such as class name, method name, file name, and the source-code line number. The getStackTrace( ) method of the Throwable class returns an array of StackTraceElements.

StackTraceElement class constructor

StackTraceElement(String declaringClass, String methodName, String fileName, int lineNumber)

This creates a stack trace element representing the specified execution point.

Stack Trace Element class methods

Method	Description
boolean equals(Object obj)	Returns true if the invoking StackTraceElement is the same as the one passed in obj. Otherwise, it returns false.
String getClassName()	Returns the class name of the execution point
String getFileName( )	Returns the filename of the execution point
int getLineNumber( )	Returns the source-code line number of the execution point
String getMethodName( )	Returns the method name of the execution point
String toString( )	Returns the String equivalent of the invoking sequence

Example:

```
public class StackTraceEx{
      public static void main(String[] args) {
       try{
                   throw new RuntimeException(“go”);  //raising an runtime exception
       }
      catch(Exception e){
                 System.out.println(“Printing stack trace:”);
//create array of stack trace elements
final StackTraceElement[] stackTrace = e.getStackTrace();
for (StackTraceElement s : stackTrace) {
    System.out.println("\tat " + s.getClassName() + "." + s.getMethodName()
            + "(" + s.getFileName() + ":" + s.getLineNumber() + ")");
            }
    }
    }
}
```

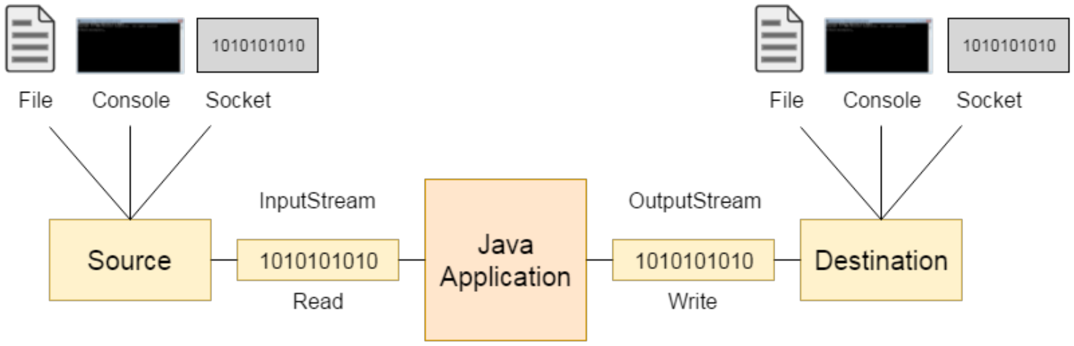
Sample Output:

```
Printing stack trace:
    at StackTraceEx.main(StackTraceEx.java:5)
```

3.7 INPUT/OUTPUT BASICS

Java I/O (Input and Output) is used to process the input and produce the output. Java uses the concept of stream to make I/O operation fast. All the classes required for input and output operations are declared in java.io package.

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

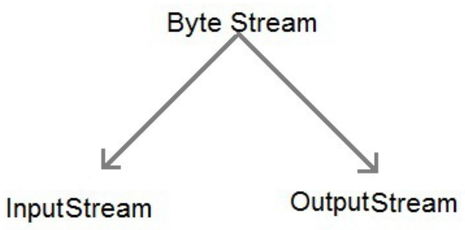


Java defines two types of streams. They are,

1. ***Byte Stream :*** It is used for handling input and output of 8 bit bytes. The frequently used classes are FileInputStream and FileOutputStream.
2. ***Character Stream :*** It is used for handling input and output of characters. Character stream uses 16 bit Unicode. The frequently used classes are FileReader and File Writer.

Byte Stream Classes

The byte stream classes are topped by two abstract classes InputStream and OutputStream.



InputStream class

InputStream class is an abstract class. It is the super class of all classes representing an input stream of bytes.

- The Input Strearn class is the superclass for all byte-oriented input stream classes.
- All the methods of this class throw an IOException.
- Being an abstract class, the InputStrearn class cannot be instantiated hence, its subclasses are used

Some of the Input Stream classes are listed below

Class	Description
Buffered Input Stream	Contains methods to read bytes from the buffer (memory area)
Byte Array Input Stream	Contains methods to read bytes from a byte array
Data Input Stream	Contains methods to read Java primitive data types
File Input Stream	Contains methods to read bytes from a file
Filter Input Stream	Contains methods to read bytes from other input streams which it uses as its basic source of data
Object Input Stream	Contains methods to read objects
Piped Input Stream	Contains methods to read from a piped output stream. A piped input stream must be connected to a piped output stream
Sequence Input Stream	Contains methods to concatenate multiple input streams and then read from the combined stream

Some of the useful methods of InputStream are listed below.

Method	Description
public abstract int read() throws IOException	Reads the next byte of data from the input stream. It returns -1 at the end of file.
public int available() throws IOException	Returns an estimate of the number of bytes that can be read from the current input stream.
public void close() throws IOException	Close the current input stream

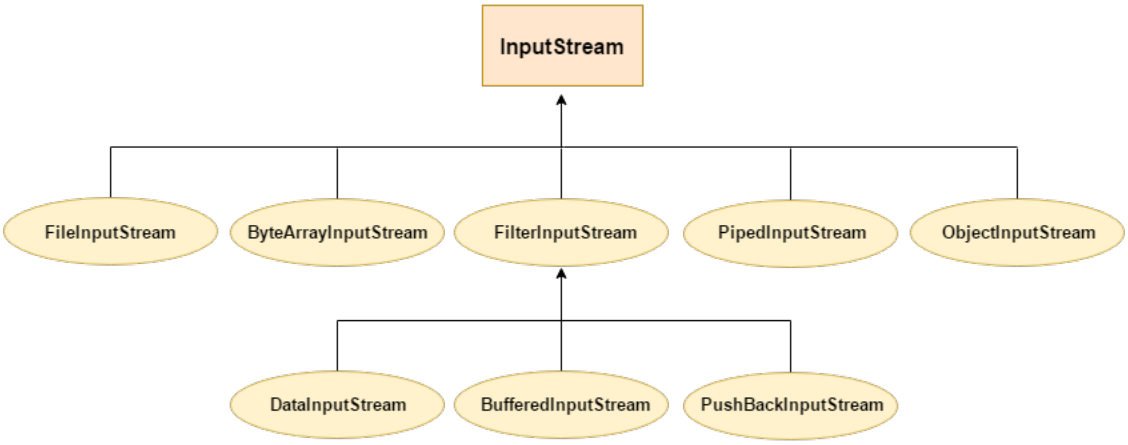


Fig. InputStream class Hierarchy

OutputStream class

OutputStream class is an abstract class. It is the super class of all classes representing an output stream of bytes. An output stream accepts output bytes and sends them to some sink.

Class	Description
Buffered Output Stream	Contains methods to write bytes into the buffer
Byte Array Output Stream	Contains methods to write bytes into a byte array
Data Output Stream	Contains methods to write Java primitive data types
File Output Stream	Contains methods to write bytes to a file
Filter Output Stream	Contains methods to write to other output streams
Object Output Stream	Contains methods to write objects
Piped Output Stream	Contains methods to write to a piped output stream
Print Stream	Contains methods to print Java primitive data types

Some of the useful methods of OutputStream class are listed below.

Method	Description
public void write(int)throws IO Exception	Write a byte to the current output stream.
public void write(byte[]) throws IO Exception	Write an array of byte to the current output stream.
public void flush()throws IO Exception	Flushes the current output stream.
public void close()throws IO Exception	close the current output stream.

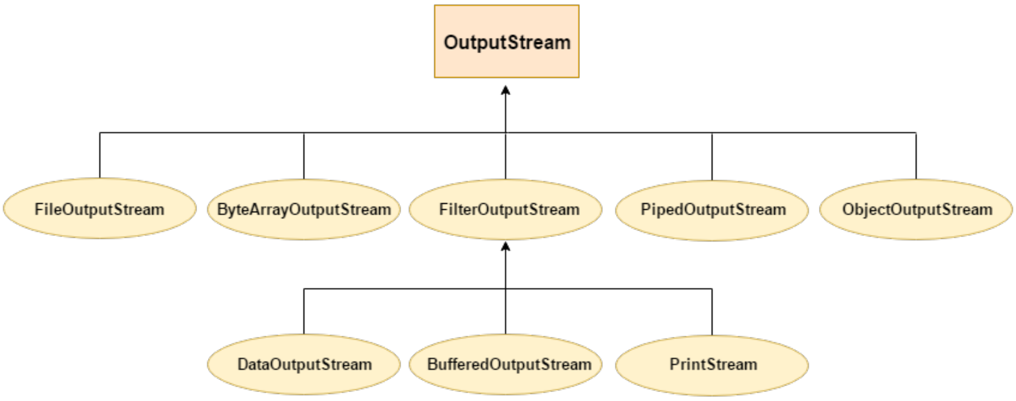
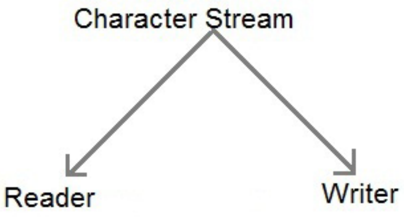


Fig. OutputStream class Hierarchy

Character Stream Classes

The character stream classes are also topped by two abstract classes Reader and Writer.



Some important Character stream reader classes are listed below.

Reader classes are used to read 16-bit unicode characters from the input stream.

- The Reader class is the superclass for all character-oriented input stream classes.
- All the methods of this class throw an IO Exception.
- Being an abstract class, the Reader class cannot be instantiated hence its subclasses are used.

Reader class	**Description**
BufferedReader	Contains methods to read characters from the buffer
FileReader	Contains methods to read from a file
InputStreamReader	Contains methods to convert bytes to characters
Reader	Abstract class that describes character stream input

The Reader class defines various methods to perform reading operations on data of an input stream. Some of these methods are listed below.

Method	**Description**
int read()	returns the integral representation of the next available character of input. It returns -1 when end of file is encountered
int read (char buffer [])	attempts to read buffer. length characters into the buffer and returns the total number of characters successfully read. It returns -I when end of file is encountered
int read (char buffer [], int loc, int nChars)	attempts to read 'nChars' characters into the buffer starting at buffer [loc] and returns the total number of characters successfully read. It returns -1 when end of file is encountered
long skip (long nChars)	skips 'nChars' characters of the input stream and returns the number of actually skipped characters
void close ()	closes the input source. If an attempt is made to read even after closing the stream then it generates IOException

Some important Character stream writer classes are listed below.

Writer classes are used to write 16-bit Unicode characters onto an outputstream.

- The Writer class is the superclass for all character-oriented output stream classes .
- All the methods of this class throw an IOException.
- Being an abstract class, the Writer class cannot be instantiated hence, its subclasses are used.

Writer class	**Description**
BufferedWriter	Contains methods to write characters to a buffer
FileWriter	Contains methods to write to a file
OutputStreamReader	Contains methods to convert from bytes to character
PrintWriter	Output stream that contains print( ) and println( )
Writer	Abstract class that describes character stream output

The Writer class defines various methods to perform writing operations on output stream. Some of these methods are listed below.

Method	Description
void write ()	writes data to the output stream
void write (int i)	Writes a single character to the output stream
void write (char buffer [])	writes an array of characters to the output stream
void write(char buffer [],int loc, int nChars)	writes 'n' characters from the buffer starting at buffer [loc] to the output stream
void close ()	closes the output stream. If an attempt is made to perform writing operation even after closing the stream then it generates IOException
void flush ()	flushes the output stream and writes the waiting buffered output characters

Predefined Streams

Java provides the following three standard streams −

- Standard Input − refers to the standard InputStream which is the keyboard by default. This is used to feed the data to user's program and represented as **System.in**.
- Standard Output − refers to the standard OutputStream by default,this is console 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 for standard error stream and represented as **System.err**.

The System class is defined in java.lang package. It contains three predefined stream variables: in, out, err. These are declared as public and static within the system.

3.8 READING CONSOLE INPUT

Reading characters

The read() method is used with BufferedReader object to read characters. As this function returns integer type value has we need to use typecasting to convert it into char type.

Syntax:

```
int read() throws IOException
```

Example:

Read character from keyboard

```
import java.io.*;
class Main
{
 public static void main( String args[]) throws IOException
{
 BufferedReader br = new BufferedReader(new InputStreamReader(System.in));
 char c;
 System.out.println("Enter characters, @ to quit");
 do{
    c = (char)br.read();       //Reading character
     System.out.println(c);
 }while(c!='@');
}
}
```

Sample Output:

```
Enter characters, @ to quit
abcd23@
a
b
c
d
2
3
@
```

Example:

Read string from keyboard

The readLine() function with BufferedReader class's object is used to read string from keyboard.

Syntax:

String readLine() throws IOException

Example :

```
import java.io.*;
public class Main{
public static void main(String args[])throws Exception{
    InputStreamReader r=new InputStreamReader(System.in);
    BufferedReader br=new BufferedReader(r);
    System.out.println("Enter your name");
    String name=br.readLine();
    System.out.println("Welcome "+name);
  }
  }
```

Sample Output :

```
Enter your name
Priya
Welcome Priya
```

3.9 WRITING CONSOLE OUTPUT

- Console output is most easily accomplished with print( ) and println( ). These methods are defined by the class PrintStream (which is the type of object referenced by System. out).
- Since PrintStream is an output stream derived from OutputStream, it also implements the low-level method write( ).
- So, write( ) can be used to write to the console.

Syntax:

void write(int byteval)

This method writes to the stream the byte specified by byteval.

The following java program uses write( ) to output the character "A" followed by a new-line to the screen:

```
// Demonstrate System.out.write().
class WriteDemo
 {
public static void main(String args[])
{
 int b;
b = 'A';
System.out.write(b);
System.out.write('\n');
 }
 }
```

3.10THE PRINTWRITER CLASS

- Although using System.out to write to the console is acceptable, its use is recommended mostly for debugging purposes or for sample programs.
- For real-world programs, the recommended method of writing to the console when using Java is through a PrintWriter stream.
- PrintWriter is one of the character-based classes.
- Using a character-based class for console output makes it easier to internationalize our program.
- PrintWriter defines several constructors.

Syntax:

```
PrintWriter(OutputStream outputStream, boolean flushOnNewline)
```

Here,

- output Stream is an object of type OutputStream
- flushOnNewline controls whether Java flushes the output stream every time a println( ) method is called.
- If flushOnNewline is true, flushing automatically takes place. If false, flushing is not automatic.
- PrintWriter supports the print( ) and println( ) methods for all types including Object.
- Thus, we can use these methods in the same way as they have been used with System. out.
- If an argument is not a simple type, the PrintWriter methods call the object's toString( ) method and then print the result.
- To write to the console by using a PrintWriter, specify System.out for the output stream and flush the stream after each newline.

For example, the following code creates a PrintWriter that is connected to console output:

```
PrintWriter pw = new PrintWriter(System.out, true);
```

The following application illustrates using a PrintWriter to handle console output:

```
// Demonstrate PrintWriter
import java.io.*;
public class PrintWriterDemo
{
 public static void main(String args[])
{
PrintWriter pw = new PrintWriter(System.out, true);
pw.println("This is a string");
 int i = -7;
pw.println(i);
double d = 4.5e-7;
pw.println(d);
 }
 }
```

Sample Output:

```
This is a string
-7
 4.5E-7
```

3.11 READING AND WRITING FILES

In Java, all files are byte-oriented, and Java provides methods to read and write bytes from and to a file.

Two of the most often-used stream classes are FileInputStream and FileOutputStream, which create byte streams linked to files.

File Input Stream

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.

The two constructors which can be used to create a FileInputStream object:

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

```
InputStream f = new FileInputStream("filename ");
```

ii) 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);

Methods to read to stream or to do other operations on the stream.

Method	Description
public void close() throws IOException{}	• Closes the file output stream. • Releases any system resources associated with the file. • Throws an IOException.
protected void finalize()throws IOException {}	• Ceans up the connection to the file. • Ensures that the close method of this file output stream is called when there are no more references to this stream. • Throws an IOException.
public int read(int r)throws IOException{}	• Reads the specified byte of data from the InputStream. • Returns an int. • Returns the next byte of data and -1 will be returned if it's the end of the file.
public int read(byte[] r) throws IOException{}	• Reads r.length bytes from the input stream into an array. • Returns the total number of bytes read. If it is the end of the file, -1 will be returned.
public int available() throws IOException{}	• Gives the number of bytes that can be read from this file input stream. • Returns an int.

File Output Stream

FileOutputStream is used to create a file and write data into it.

The stream would create a file, if it doesn't already exist, before opening it for output.

The two constructors which can be used to create a FileOutputStream object:

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

OutputStream f = new FileOutputStream("filename");

ii) 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);
```

Methods to write to stream or to do other operations on the stream

Method	Description
public void close() throws IO-Exception{}	• Closes the file output stream. • Releases any system resources associated with the file. • Throws an IOException.
protected void finalize()throws IOException {}	• Cleans up the connection to the file. • Ensures that the close method of this file output stream is called when there are no more references to this stream. • Throws an IOException.
public void write(int w)throws IOException{}	• Writes the specified byte to the output stream.
public void write(byte[] w)	• Writes w.length bytes from the mentioned byte array to the OutputStream.

Following code demonstrates the use of InputStream and OutputStream.

```
import java.io.*;
public class fileStreamTest
{
 public static void main(String args[])
{
try
{
        byte bWrite [] = {11,21,3,40,5};
        OutputStream os = new FileOutputStream("test.txt");
        for(int x = 0; x < bWrite.length ; x++)
       {
           os.write( bWrite[x] );   // writes the bytes
        }
        os.close();
        InputStream is = new FileInputStream(“test.txt”);
        int size = is.available();
        for(int i = 0; i < size; i++)
       {
           System.out.print((char)is.read() + “  “);
        }
                              is.close();
               }
catch (IOException e)
{
        System.out.print(“Exception”);
          }
      }
}
```

The above code creates a file named test.txt and writes given numbers in binary format. The same will be displayed as output on the stdout screen.

UNIT-4

MULTI THREADING AND GENERIC PROGRAMMING

4.1 MULTITHREADING AND MULTI-TASKING

In programming, there are two main ways to improve the throughput of a program:

i) by using multi-threading

ii) by using multitasking

Both these methods take advantage of parallelism to efficiently utilize the power of CPU and improve the throughput of program.

Difference between multithreading and multi-tasking

1. The basic difference between multitasking and multithreading is that in multitasking, the system allows executing multiple programs and tasks at the same time, whereas, in multithreading, the system executes multiple threads of the same or different processes at the same time.
2. Multi-threading is more granular than multi-tasking. In multi-tasking, CPU switches between multiple programs to complete their execution in real time, while in multithreading CPU switches between multiple threads of the same program. Switching between multiple processes has more context switching cost than switching between multiple threads of the same program.
3. Processes are heavyweight as compared to threads. They require their own address space, which means multi-tasking is heavy compared to multithreading.
4. Multitasking allocates separate memory and resources for each process/program whereas, in multithreading threads belonging to the same process shares the same memory and resources as that of the process.

Comparison between multithreading and multi-tasking

Parameter	Multi Tasking	Multi Threading
Basic	Multitasking lets CPU to execute multiple tasks at the same time.	Multithreading lets CPU to execute multiple threads of a process simultaneously.
Switching	In multitasking, CPU switches between programs frequently.	In multithreading, CPU switches between the threads frequently.
Memoryand Resource	In multitasking, system has to allocate separate memory and resources to each program that CPU is executing.	In multithreading, system has to allocate memory to a process, multiple threads of that process shares the same memory and resources allocated to the process.

Multitasking

Multitasking is when a single CPU performs several tasks (program, process, task, threads) at the same time. To perform multitasking, the CPU switches among these tasks very frequently so that user can interact with each program simultaneously.

In a multitasking operating system, several users can **share the system** simultaneously. CPU rapidly switches among the tasks, so a little time is needed to switch from one user to the next user. This puts an impression on a user that entire computer system is dedicated to him.

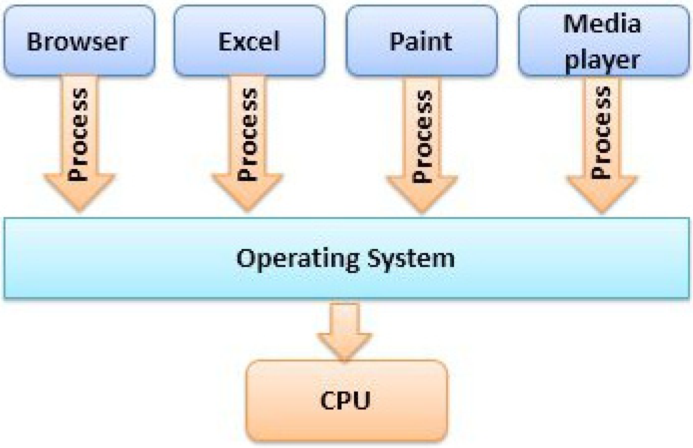


Figure: Multitasking

When several users are sharing a multitasking operating system, CPU scheduling and multiprogramming makes it possible for each user to have at least a small portion of Multitasking OS and let each user have at least one program in the memory for execution.

Multi threading

Multithreading is different from multitasking in a sense that multitasking allows multiple tasks at the same time, whereas, the Multithreading allows multiple threads of a single task (program, process) to be processed by CPU at the same time.

A thread is a basic execution unit which has its own program counter, set of the register and stack. But it shares the code, data, and file of the process to which it belongs. A process can have multiple threads simultaneously, and the CPU switches among these threads so frequently making an impression on the user that all threads are running simultaneously.

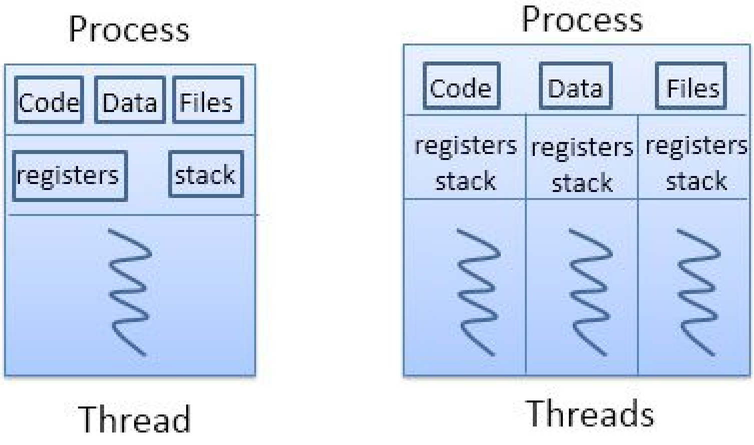


Figure: Multithreading

Benefits of Multithreading

- Multithreading increases the **responsiveness** of system as, if one thread of the application is not responding, the other would respond in that sense the user would not have to sit idle.
- Multithreading allows **resource sharing** as threads belonging to the same process can share code and data of the process and it allows a process to have multiple threads at the same time active in **same address space**.
- Creating a different process is costlier as the system has to allocate different memory and resources to each process, but creating threads is easy as it does not require allocating separate memory and resources for threads of the same process.

4.2 THREAD LIFECYCLE

A thread in Java at any point of time exists in any one of the following states. A thread lies only in one of the shown states at any instant:

1) New
2) Runnable
3) Blocked
4) Waiting
5) Timed Waiting
6) Terminated

The following figure represents various states of a thread at any instant of time:

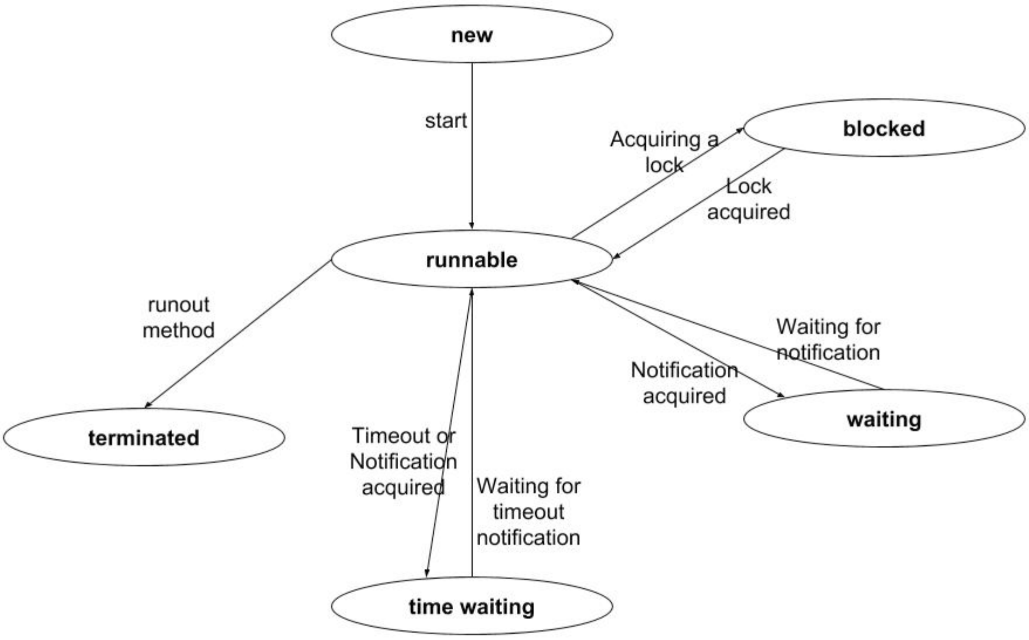


Figure: Life Cycle of a thread

1. New Thread:

- When a new thread is created, it is in the new state.
- The thread has not yet started to run when thread is in this state.
- When a thread lies in the new state, it's code is yet to be run and hasn't started to execute.

2. Runnable State:

- A thread that is ready to run is moved to runnable state.
- In this state, a thread might actually be running or it might be ready run at any instant of time.

- It is the responsibility of the thread scheduler to give the thread, time to run.
- A multi-threaded program allocates a fixed amount of time to each individual thread. Each and every thread runs for a short while and then pauses and relinquishes the CPU to another thread, so that other threads can get a chance to run. When this happens, all such threads that are ready to run, waiting for the CPU and the currently running thread lies in runnable state.

3. Blocked/Waiting state:

- When a thread is temporarily inactive, then it's in one of the following states:
  - Blocked
  - Waiting
- For example, when a thread is waiting for I/O to complete, it lies in the blocked state. It's the responsibility of the thread scheduler to reactivate and schedule a blocked/ waiting thread.
- A thread in this state cannot continue its execution any further until it is moved to runnable state. Any thread in these states do not consume any CPU cycle.
- A thread is in the blocked state when it tries to access a protected section of code that is currently locked by some other thread. When the protected section is unlocked, the schedule picks one of the threads which is blocked for that section and moves it to the runnable state. A thread is in the waiting state when it waits for another thread on a condition. When this condition is fulfilled, the scheduler is notified and the waiting thread is moved to runnable state.
- If a currently running thread is moved to blocked/waiting state, another thread in the runnable state is scheduled by the thread scheduler to run. It is the responsibility of thread scheduler to determine which thread to run.

4. Timed Waiting:

- A thread lies in timed waiting state when it calls a method with a time out parameter.
- A thread lies in this state until the timeout is completed or until a notification is received.
- For example, when a thread calls sleep or a conditional wait, it is moved to timed waiting state.

5. Terminated State:

- A thread terminates because of either of the following reasons:
  - Because it exits normally. This happens when the code of thread has entirely executed by the program.

    - ○ Because there occurred some unusual erroneous event, like segmentation fault or an unhandled exception.

- A thread that lies in terminated state does no longer consumes any cycles of CPU.

Creating Threads

- Threading is a facility to allow multiple tasks to run concurrently within a single process. Threads are independent, concurrent execution through a program, and each thread has its own stack.

In Java, There are two ways to create a thread:

1) By extending Thread class.
2) By implementing Runnable interface.

Java Thread Benefits

1. Java Threads are lightweight compared to processes as they take less time and resource to create a thread.
2. Threads share their parent process data and code
3. Context switching between threads is usually less expensive than between processes.
4. Thread intercommunication is relatively easy than process communication.

Thread class:

Thread class provide constructors and methods to create and perform operations on a thread. Thread class extends Object class and implements Runnable interface.

Commonly used Constructors of Thread class:

- Thread()
- Thread(String name)
- Thread(Runnable r)
- Thread(Runnable r, String name)

Commonly used methods of Thread class:

1. **public void run():** is used to perform action for a thread.
2. **public void start():** starts the execution of the thread. JVM calls the run() method on the thread.
3. **public void sleep(long miliseconds):** Causes the currently executing thread to sleep (temporarily cease execution) for the specified number of milliseconds.
4. **public void join():** waits for a thread to die.

5. **public void join(long miliseconds):** waits for a thread to die for the specified miliseconds.
6. **public int getPriority():** returns the priority of the thread.
7. **public int setPriority(int priority):** changes the priority of the thread.
8. **public String getName():** returns the name of the thread.
9. **public void setName(String name):** changes the name of the thread.
10. **public Thread currentThread():** returns the reference of currently executing thread.
11. **public int getId():** returns the id of the thread.
12. **public Thread.State getState():** returns the state of the thread.
13. **public boolean isAlive():** tests if the thread is alive.
14. **public void yield():** causes the currently executing thread object to temporarily pause and allow other threads to execute.
15. **public void suspend():** is used to suspend the thread(depricated).
16. **public void resume():** is used to resume the suspended thread(depricated).
17. **public void stop():** is used to stop the thread(depricated).
18. **public boolean isDaemon():** tests if the thread is a daemon thread.
19. **public void setDaemon(boolean b):** marks the thread as daemon or user thread.
20. **public void interrupt():** interrupts the thread.
21. **public boolean isInterrupted():** tests if the thread has been interrupted.
22. **public static boolean interrupted():** tests if the current thread has been interrupted.

Naming Thread

The Thread class provides methods to change and get the name of a thread. By default, each thread has a name i.e. thread-0, thread-1 and so on. But we can change the name of the thread by using setName() method. The syntax of setName() and getName() methods are given below:

public String getName(): is used to return the name of a thread.

public void setName(String name): is used to change the name of a thread.

Extending Thread

The first way to create a thread is to create a new class that extends Thread, and then to create an instance of that class. The extending class must override the run( ) method, which is the entry point for the new thread. It must also call start( ) to begin execution of the new thread.

Sample java program that creates a new thread by extending Thread:

```
// Create a second thread by extending Thread
class NewThread extends Thread
{
   NewThread()
   { // Create a new, second thread
      super(“Demo Thread”);
      System.out.println(“Child thread: “ + this);
      start(); // Start the thread
   }
   // This is the entry point for the second thread.
   public void run()
   {
      try
      {
         for(int i = 5; i > 0; i--)
         {
            System.out.println(“Child Thread: “ + i);
            Thread.sleep(500);
         }
      }
      catch (InterruptedException e)
      {
         System.out.println(“Child interrupted.”);
      }
      System.out.println(“Child thread is exiting”);
```

```
    }
}
public class ExtendThread
{
    public static void main(String args[])
    {
        new NewThread(); // create a new thread
        try
        {
            for(int i = 5; i > 0; i--)
            {
                System.out.println(“Main Thread: “ + i);
                Thread.sleep(1000);
            }
        }
        catch (InterruptedException e)
        {
            System.out.println(“Main thread interrupted.”);
        }
        System.out.println(“Main thread is exiting.”);
    }
}
```

Sample Output:

(output may vary based on processor speed and task load)

```
Child thread: Thread[Demo Thread,5,main]
Main Thread: 5
Child Thread: 5
Child Thread: 4
Main Thread: 4
Child Thread: 3
Child Thread: 2
Main Thread: 3
Child Thread: 1
Child thread is exiting.
Main Thread: 2
Main Thread: 1
Main thread is exiting.
```

The child thread is created by instantiating an object of NewThread, which is derived from Thread. The call to super( ) is inside NewThread. This invokes the following form of the Thread constructor:

```
public Thread(String threadName)
```

Here, threadName specifies the name of the thread.

Implementing Runnable

- The easiest way to create a thread is to create a class that implements the Runnable interface.
- Runnable abstracts a unit of executable code. We can construct a thread on any object that implements Runnable.
- To implement Runnable, a class need only implement a single method called run( ), which is declared as:

  ```
  public void run( )
  ```

- Inside run( ), we will define the code that constitutes the new thread. The run( ) can call other methods, use other classes, and declare variables, just like the main thread can. The only difference is that run( ) establishes the entry point for another, concurrent thread of execution within the program. This thread will end when run( ) returns.
- After we create a class that implements Runnable, we will instantiate an object of type Thread from within that class.
- After the new thread is created, it will not start running until we call its start( ) method, which is declared within Thread. In essence, start( ) executes a call to run( ).
- The start( ) method is shown as:

  ```
  void start( )
  ```

Sample java program that creates a new thread by implementing Runnable:

```
// Create a second thread
class NewThread implements Runnable
{
    Thread t;
    NewThread()
    {
        // Create a new, second thread
        t = new Thread(this, “Demo Thread”);
        System.out.println(“Child thread: “ + t);
        t.start(); // Start the thread
     }
// This is the entry point for the second thread.
    public void run()
    {
        try
        {
            for(int i = 5; i > 0; i--)
            {
                System.out.println(“Child Thread: “ + i);
                Thread.sleep(500);
            }
        }
        catch (InterruptedException e)
        {
            System.out.println(“Child interrupted.”);
        }
        System.out.println(“Child thread is exiting.”);
    }
}
```

```
public class ThreadDemo
{
   public static void main(String args[])
   {
      new NewThread(); // create a new thread
      try
      {
         for(int i = 5; i > 0; i--)
         {
            System.out.println(“Main Thread: “ + i);
            Thread.sleep(1000);
         }
      }
      catch (InterruptedException e)
      {
         System.out.println(“Main thread interrupted.”);
      }
      System.out.println(“Main thread is exiting.”);
   }
}
```

Inside NewThread’s constructor, a new Thread object is created by the following statement:

```
t = new Thread(this, “Demo Thread”);
```

Passing this as the first argument indicates that we want the new thread to call the run( ) method on this object. Next, start( ) is called, which starts the thread of execution beginning at the run( ) method. This causes the child thread’s for loop to begin. After calling start( ), NewThread’s constructor returns to main(). When the main thread resumes, it enters its for loop. Both threads continue running, sharing the CPU, until their loops finish.

Sample Output:

(output may vary based on processor speed and task load)

```
Child thread: Thread[Demo Thread,5,main]
Main Thread: 5
Child Thread: 5
Child Thread: 4
Main Thread: 4
Child Thread: 3
Child Thread: 2
Main Thread: 3
Child Thread: 1
Child thread is exiting.
Main Thread: 2
Main Thread: 1
Main thread is exiting.
```

In a multithreaded program, often the main thread must be the last thread to finish running. In fact, for some older JVMs, if the main thread finishes before a child thread has completed, then the Java run-time system may "hang." The preceding program ensures that the main thread finishes last, because the main thread sleeps for 1,000 milliseconds between iterations, but the child thread sleeps for only 500 milliseconds. This causes the child thread to terminate earlier than the main thread.

Choosing an Approach

The Thread class defines several methods that can be overridden by a derived class. Of these methods, the only one that must be overridden is run(). This is, of course, the same method required when we implement Runnable. Many Java programmers feel that classes should be extended only when they are being enhanced or modified in some way. So, if we will not be overriding any of Thread's other methods, it is probably best simply to implement Runnable.

Creating Multiple Threads

The following program creates three child threads:

```
// Create multiple threads.
class NewThread implements Runnable
{
    String name; // name of thread
    Thread t;
    NewThread(String threadname)
    {
        name = threadname;
        t = new Thread(this, name);
        System.out.println("New thread: " + t);
        t.start(); // Start the thread
    }
    // This is the entry point for thread.
    public void run()
    {
        try
        {
            for(int i = 5; i > 0; i--)
            {
                System.out.println(name + ": " + i);
                Thread.sleep(1000);
            }
        }
        catch (InterruptedException e)
        {
            System.out.println(name + "Interrupted");
        }
        System.out.println(name + " exiting.");
```

```
   }
}
public class MultiThreadDemo
{
   public static void main(String args[])
   {
      new NewThread(“One”); // start threads
      new NewThread(“Two”);
      new NewThread(“Three”);
      try
      {
         // wait for other threads to end
         Thread.sleep(10000);
      }
      catch (InterruptedException e)
      {
         System.out.println(“Main thread Interrupted”);
      }
      System.out.println(“Main thread exiting.”);
   }
}
```

The output from this program is shown here:

```
New thread: Thread[One,5,main]
New thread: Thread[Two,5,main]
New thread: Thread[Three,5,main]
One: 5
 Two: 5
Three: 5
One: 4
Two: 4
```

```
 Three: 4
One: 3
Three: 3
Two: 3
One: 2
Three: 2
Two: 2
One: 1
Three: 1
Two: 1
One exiting.
Two exiting.
Three exiting.
Main thread exiting.
```

As we can see, once started, all three child threads share the CPU. The call to sleep(10000) in main(). This causes the main thread to sleep for ten seconds and ensures that it will finish last.

Using isAlive( ) and join( )

We want the main thread to finish last. In the preceding examples, this is accomplished by calling sleep( ) within main( ), with a long enough delay to ensure that all child threads terminate prior to the main thread. However, this is hardly a satisfactory solution, and it also raises a larger question: How can one thread know when another thread has ended?

Two ways exist to determine whether a thread has finished or not.

- First, we can call isAlive( ) on the thread. This method is defined by Thread.

Syntax:

```
final boolean isAlive( )
```

The isAlive( ) method returns true, if the thread upon which it is called is still running. It returns false, otherwise.

- Second, we can use join() to wait for a thread to finish.

Syntax:

```
final void join( ) throws InterruptedException
```

This method waits until the thread on which it is called terminates. Its name comes from the concept of the calling thread waiting until the specified thread joins it.

Sample Java program using join() to wait for threads to finish.

```
class NewThread implements Runnable
{
    String name; // name of thread
    Thread t;
    NewThread(String threadname)
    {
        name = threadname;
        t = new Thread(this, name);
        System.out.println(“New thread: “ + t);
        t.start(); // Start the thread
    }
    // This is the entry point for thread.
    public void run()
    {
        try
        {
            for(int i = 5; i > 0; i--)
            {
                System.out.println(name + “: “ + i);
                Thread.sleep(1000);
            }
        }
        catch (InterruptedException e)
        {
            System.out.println(name + “ interrupted.”);
        }
        System.out.println(name + “ is exiting.”);
    }
}
public class DemoJoin
{
    public static void main(String args[])
    {
        NewThread ob1 = new NewThread(“One”);
        NewThread ob2 = new NewThread(“Two”);
        NewThread ob3 = new NewThread(“Three”);
        System.out.println(“Thread One is alive: “ + ob1.t.isAlive());
        System.out.println(“Thread Two is alive: “ + ob2.t.isAlive());
        System.out.println(“Thread Three is alive: “ + ob3.t.isAlive());
        // wait for threads to finish
        try
        {
            System.out.println(“Waiting for threads to finish.”);
            ob1.t.join();
            ob2.t.join();
            ob3.t.join();
        }
        catch (InterruptedException e)
        {
            System.out.println(“Main thread Interrupted”);
        }
        System.out.println(“Thread One is alive: “ + ob1.t.isAlive());
        System.out.println(“Thread Two is alive: “ + ob2.t.isAlive());
        System.out.println(“Thread Three is alive: “ + ob3.t.isAlive());
        System.out.println(“Main thread is exiting.”);
    }
}
```

Sample output:

(output may vary based on processor speed and task load)

```
New thread: Thread[One,5,main]
New thread: Thread[Two,5,main]
One: 5
New thread: Thread[Three,5,main]
Two: 5
Thread One is alive: true
Thread Two is alive: true
Thread Three is alive: true
Waiting for threads to finish.
Three: 5
One: 4
Two: 4
Three: 4
One: 3
Two: 3
Three: 3
One: 2
Two: 2
Three: 2
One: 1
Two: 1
Three: 1
One is exiting.
Two is exiting.
Three is exiting.
Thread One is alive: false
Thread Two is alive: false
Thread Three is alive: false
Main thread is exiting.
```

As we can see, after the calls to join( ) return, the threads have stopped executing.

4.3 SYNCHRONIZATION

- Synchronization in java is the capability *to control the access of multiple threads to any shared resource.*
- Java Synchronization is better option where we want to allow only one thread to access the shared resource.
- When two or more threads need access to a shared resource, they need some way to ensure that the resource will be used by only one thread at a time. The process by which this is achieved is called synchronization. Java provides unique, language-level support for it.
- Key to synchronization is the concept of the monitor (also called a semaphore).
- A monitor is an object that is used as a mutually exclusive lock, or mutex. Only one thread can own a monitor at a given time. When a thread acquires a lock, it is said to have entered the monitor. All other threads attempting to enter the locked monitor will be suspended until the first thread exits the monitor.
- These other threads are said to be waiting for the monitor. A thread that owns a monitor can reenter the same monitor if it so desires.
- Approaches:
    - ○ Using synchronized Method
    - ○ Using synchronized Statement

Using Synchronized Methods

Synchronization is easy in Java, because all objects have their own implicit monitor associated with them.

To enter an object's monitor, just call a method that has been modified with the synchronized keyword.

While a thread is inside a synchronized method, all other threads that try to call it (or any other synchronized method) on the same instance have to wait.

To exit the monitor and relinquish control of the object to the next waiting thread, the owner of the monitor simply returns from the synchronized method.

- To understand the need for synchronization, we will consider a simple example that does not use it—but should.
- The following program has three simple classes.
- The first one, Callme, has a single method named call( ). The call( ) method takes a String parameter called msg. This method tries to print the msg string inside of square brackets. After call( ) prints the opening bracket and the msg string, it calls Thread.

sleep(1000), which pauses the current thread for one second.

- The constructor of the next class, Caller, takes a reference to an instance of the Callme class and a String, which are stored in target and msg, respectively. The constructor also creates a new thread that will call this object's run( ) method. The thread is started immediately. The run( ) method of Caller calls the call( ) method on the target instance of Callme, passing in the msg string.
- Finally, the Synch class starts by creating a single instance of Callme, and three instances of Caller, each with a unique message string.
- The same instance of Callme is passed to each Caller.

// This program is not synchronized.

```
class Callme
{
    void call(String msg)
    {
        System.out.print("[" + msg);
        try
        {
            Thread.sleep(1000);
        }
        catch(InterruptedException e)
        {
            System.out.println("Interrupted");
        }
        System.out.println("]");
    }
}

class Caller implements Runnable
{
    String msg;
    Callme target;
    Thread t;
    public Caller(Callme targ, String s)
    {
        target = targ;
        msg = s;
        t = new Thread(this);
        t.start();
    }
    public void run()
    {
        target.call(msg);
    }
}

public class Synch
{
    public static void main(String args[])
    {
        Callme target = new Callme();
        Caller ob1 = new Caller(target, “Hello”);
        Caller ob2 = new Caller(target, “Synchronized”);
        Caller ob3 = new Caller(target, “World”);
        // wait for threads to end
        try
        {
            ob1.t.join();
            ob2.t.join();
            ob3.t.join();
        }
        catch(InterruptedException e)
        {
         System.out.println(“Interrupted”);
        }
    }
}
```

***Sample Output*:**

```
Hello[Synchronized[World]
 ]
]
```

As we can see, by calling sleep( ), the call( ) method allows execution to switch to another thread. This results in the mixed-up output of the three message strings.

In this program, nothing exists to stop all three threads from calling the same method, on the same object, at the same time. This is known as a race condition, because the three threads are racing each other to complete the method.

This example used sleep( ) to make the effects repeatable and obvious. In most situations, a race condition is more subtle and less predictable, because we can’t be sure when the context switch will occur. This can cause a program to run right one time and wrong the next.

To fix the preceding program, we must serialize access to call(). That is, we must restrict its access to only one thread at a time. To do this, we simply need to precede call()’s definition with the keyword synchronized, as shown here:

This prevents other threads from entering call( ) while another thread is using it.

```
class Callme
{
synchronized void call(String msg)
{
 ...
```

Following is the sample java program after synchronized has been added to call( ):

```
class Callme
{
    synchronized void call(String msg)
    {
        System.out.print(“[“ + msg);
        try
        {
            Thread.sleep(1000);
        }
        catch(InterruptedException e)
        {
            System.out.println(“Interrupted”);
        }
        System.out.println(“]”);
    }
}

class Caller implements Runnable
{
    String msg;
    Callme target;
    Thread t;
    public Caller(Callme targ, String s)
    {
        target = targ;
        msg = s;
        t = new Thread(this);
        t.start();
    }
    public void run()
    {
        target.call(msg);
    }
}
```

```
public class Synch
{
   public static void main(String args[])
   {
      Callme target = new Callme();
      Caller ob1 = new Caller(target, “Hello”);
      Caller ob2 = new Caller(target, “Synchronized”);
      Caller ob3 = new Caller(target, “World”);
      // wait for threads to end
      try
      {
         ob1.t.join();
         ob2.t.join();
         ob3.t.join();
      }
      catch(InterruptedException e)
      {
       System.out.println(“Interrupted”);
      }
   }
}
```

***Output*:**

```
 [Hello]
[Synchronized]
[World]
```

Using synchronized Statement

While creating synchronized methods within classes that we create is an easy and effective means of achieving synchronization, it will not work in all cases. We have to put calls to the methods defined by the class inside a synchronized block.

Syntax:

```
synchronized(object)
 {
// statements to be synchronized
 }
```

Here, object is a reference to the object being synchronized. A synchronized block ensures that a call to a method that is a member of object occurs only after the current thread has successfully entered object's monitor.

Here is an alternative version of the preceding example, using a synchronized block within the run( ) method:

// This program uses a synchronized block.

```
class Callme
{
    void call(String msg)
    {
        System.out.print("[" + msg);
        try
        {
            Thread.sleep(1000);
        }
        catch (InterruptedException e)
        {
            System.out.println("Interrupted");
        }
        System.out.println("]");
    }
}
class Caller implements Runnable
{
    String msg;
    Callme target;
    Thread t;
    public Caller(Callme targ, String s)
    {
        target = targ;
        msg = s;
        t = new Thread(this);
        t.start();
    }
    // synchronize calls to call()
    public void run()
    {
        synchronized(target)
        {
            // synchronized block
            target.call(msg);
        }
    }
}
public class Synch1
{
    public static void main(String args[])
    {
        Callme target = new Callme();
        Caller ob1 = new Caller(target, “Hello”);
        Caller ob2 = new Caller(target, “Synchronized”);
        Caller ob3 = new Caller(target, “World”);
        // wait for threads to end
        try
        {
            ob1.t.join();
            ob2.t.join();
            ob3.t.join();
         }
         catch(InterruptedException e)
         {
            System.out.println("Interrupted");
         }
      }
   }
```

Here, the call( ) method is not modified by synchronized. Instead, the synchronized statement is used inside Caller's run( ) method. This causes the same correct output as the preceding example, because each thread waits for the prior one to finish before proceeding.

Sample Output:

```
[Hello]
[World]
[Synchronized]
```

Priority of a Thread (Thread Priority):

Each thread has a priority. Priorities are represented by a number between 1 and 10. In most cases, thread schedular schedules the threads according to their priority (known as preemptive scheduling). But it is not guaranteed because it depends on JVM specification that which scheduling it chooses.

3 constants defined in Thread class:

1. public static int MIN_PRIORITY
2. public static int NORM_PRIORITY
3. public static int MAX_PRIORITY

Default priority of a thread is 5 (NORM_PRIORITY). The value of MIN_PRIORITY is 1 and the value of MAX_PRIORITY is 10.

Sample Java Program:

```
public class TestMultiPriority1 extends Thread
{
   public void run()
   {
```

```
   System.out.println("running thread name is:"+Thread.currentThread().getName());
   System.out.println("running thread priority is:"+Thread.currentThread().getPriority());
   }
   public static void main(String args[])
   {
      TestMultiPriority1 m1=new TestMultiPriority1();
      TestMultiPriority1 m2=new TestMultiPriority1();
      m1.setPriority(Thread.MIN_PRIORITY);
      m2.setPriority(Thread.MAX_PRIORITY);
      m1.start();
      m2.start();
   }
}
```

Output:

```
running thread name is:Thread-0
running thread priority is:10
running thread name is:Thread-1
running thread priority is:1
```

4.4 INTER-THREAD COMMUNICATION

Inter-process communication (IPC) is a mechanism that allows the exchange of data between processes. By providing a user with a set of programming interfaces, IPC helps a programmer organize the activities among different processes. IPC allows one application to control another application, thereby enabling data sharing without interference.

IPC enables data communication by allowing processes to use segments, semaphores, and other methods to share memory and information. IPC facilitates efficient message transfer between processes. The idea of IPC is based on Task Control Architecture (TCA). It is a flexible technique that can send and receive variable length arrays, data structures, and lists. It has the capability of using publish/subscribe and client/server data-transfer paradigms while supporting a wide range of operating systems and languages.

Inter-thread communication or Co-operation is all about allowing synchronized threads to communicate with each other. Interthread communication is important when you develop an application where two or more threads exchange some information.

Cooperation (Inter-thread communication) is a mechanism in which a thread is paused running in its critical section and another thread is allowed to enter (or lock) in the same critical section to be executed. It is implemented by following methods of Object class:

- wait()
- notify()
- notifyAll()

All these methods belong to object class as final so that all classes have them. They must be used within a synchronized block only.

1) wait() method

Causes current thread to release the lock and wait until either another thread invokes the notify() method or the notifyAll() method for this object, or a specified amount of time has elapsed. The current thread must own this object's monitor, so it must be called from the synchronized method only otherwise it will throw exception.

2) notify() method

Wakes up a single thread that is waiting on this object's monitor. If any threads are waiting on this object, one of them is chosen to be awakened. The choice is arbitrary and occurs at the discretion of the implementation. Syntax:

```
public final void notify()
```

3) notifyAll() method

Wakes up all threads that are waiting on this object's monitor. Syntax:

```
public final void notifyAll()
```

// Java program to demonstrate inter-thread communication (wait(), join() and notify()) in Java

```
import java.util.Scanner;
public class Thread_Example
{
    public static void main(String[] args) throws InterruptedException
    {
            final Producer_Consumer  pc = new Producer_Consumer ();
            Thread t1 = new Thread(new Runnable()
            {
            public void run()
                {
```

```
                try
                {
                    pc.producer();
                }
                catch(InterruptedException e)
                {
                    e.printStackTrace();
                }
            }
        });
        Thread t2 = new Thread(new Runnable()
        {
            public void run()
            {
                try
                {
                    pc.consumer();
                }
                catch(InterruptedException e)
                {
                    e.printStackTrace();
                }
            }
        });
        t1.start();
        t2.start();
        t1.join();
        t2.join();
    }
```

```
        public static class Producer_Consumer
        {
          public void producer()throws InterruptedException
                {
                        synchronized(this)
                        {
                                System.out.println("producer thread running");
                                wait();
                                System.out.println("Resumed");
                        }
                }
                public void consumer()throws InterruptedException
                {
                        Thread.sleep(1000);
                        Scanner ip = new Scanner(System.in);
                        synchronized(this)
                        {
                                System.out.println("Waiting for return key.");
                                ip.nextLine();
                                System.out.println("Return key pressed");
                                notify();
                                Thread.sleep(1000);
                        }
                }
        }
    }
```

The following statements explain how the above producer-Consumer program works.

- The use of synchronized block ensures that only one thread at a time runs. Also since there is a sleep method just at the beginning of consumer loop, the produce thread gets a kickstart.

- When the wait is called in producer method, it does two things.
  1. it releases the lock it holds on PC object.
  2. it makes the produce thread to go on a waiting state until all other threads have terminated, that is it can again acquire a lock on PC object and some other method wakes it up by invoking notify or notifyAll on the same object.
- Therefore we see that as soon as wait is called, the control transfers to consume thread and it prints -"Waiting for return key".
- After we press the return key, consume method invokes notify(). It also does 2 things- Firstly, unlike wait(), it does not releases the lock on shared resource therefore for getting the desired result, it is advised to use notify only at the end of your method. Secondly, it notifies the waiting threads that now they can wake up but only after the current method terminates.
- As you might have observed that even after notifying, the control does not immediately passes over to the produce thread. The reason for it being that we have called Thread.sleep() after notify(). As we already know that the consume thread is holding a lock on PC object, another thread cannot access it until it has released the lock. Hence only after the consume thread finishes its sleep time and thereafter terminates by itself, the produce thread cannot take back the control.
- After a 2 second pause, the program terminates to its completion.
- The following program is one more example for interthread communication

```
class InterThread_Example
{
  public static void main(String arg[])
  {
    final Client  c = new Client();
    new Thread()
    {
      public void run()
      {
        c.withdraw(15000);
      }
    }.start();
```

```
        new Thread()
        {
            public void run()
            {
                c.deposit(10000);
            }
        }.start();
        new Thread()
        {
            public void run()
            {
                c.deposit(10000);
            }
        }.start();
    }
}

class Client
{
    int amount = 10000;

    synchronized void withdraw(int amount)
    {
        System.out.println(“Available Balance “ + this. amount);
        System.out.println(“withdrawal amount.” + amount);

        if (this.amount < amount)
```

```
            {
                System.out.println(“Insufficient Balance waiting for deposit.”);
                try
                {
                    wait();
                } catch (Exception e)
                {
                    System.out.println(“Interruption Occured”);
                }
            }
            this.amount -= amount;
            System.out.println(“Detected amount: “ + amount);
            System.out.println(“Balance amount : “ + this.amount);
        }
        synchronized void deposit(int amount)
        {
            System.out.println(“Going to deposit “ + amount);
            this.amount += amount;
            System.out.println(“Available Balance “ + this.amount);
            System.out.println(“Transaction completed.\n”);
            notify();
        }
    }
```

4.5 DAEMON THREAD

Daemon thread is a low priority thread that runs in background to perform tasks such as garbage collection. Daemon thread in java is a service provider thread that provides services to the user thread. Its life depend on the mercy of user threads i.e. when all the user threads dies, JVM terminates this thread automatically.

There are many java daemon threads running automatically e.g. gc, finalizer etc.

- It provides services to user threads for background supporting tasks. It has no role in life than to serve user threads.
- Its life depends on user threads.
- It is a low priority thread.

The command jconsole typed in the command prompt provides information about the loaded classes, memory usage, running threads etc.

The purpose of the daemon thread is that it provides services to user thread for background supporting task. If there is no user thread, why should JVM keep running this thread. That is why JVM terminates the daemon thread if there is no user thread.

Properties:

- They cannot prevent the JVM from exiting when all the user threads finish their execution.
- JVM terminates itself when all user threads finish their execution
- If JVM finds running daemon thread, it terminates the thread and after that shutdown itself. JVM does not care whether Daemon thread is running or not.
- It is an utmost low priority thread.

Methods for Java Daemon thread by Thread class

The java.lang.Thread class provides two methods for java daemon thread.

Method	Description
public void setDaemon(boolean status)	used to mark the current thread as daemon thread or user thread.
public boolean isDaemon()	used to check that current is daemon.

// Java program to demonstrate the usage of setDaemon() and isDaemon() method.

```
public class DaemonThread extends Thread
{
    public void run()
    {
            // Checking whether the thread is Daemon or not
            if(Thread.currentThread().isDaemon())
            {
                    System.out.println("This is Daemon thread");
```

```
                }
        else
                {
                        System.out.println("This is User thread");
                }
        }
        public static void main(String[] args)
        {
                DaemonThread t1 = new DaemonThread();
                DaemonThread t2 = new DaemonThread();
                DaemonThread t3 = new DaemonThread();
                // Setting user thread t1 to Daemon
                t1.setDaemon(true);
                // starting all the threads
                t1.start();
                t2.start();
                t3.start();
                // Setting user thread t3 to Daemon
                t3.setDaemon(true);
        }
    }
```

Output:

```
    This is Daemon thread
    This is User thread
    This is Daemon thread
```

// Java program to demonstrate the usage of exception in Daemon() Thread

```
    public class DaemonThread extends Thread
    {
        public void run()
        {
```

```
System.out.println("Thread name: " + Thread.currentThread().getName());
                System.out.println("Check if its DaemonThread: "
                                                      + Thread.currentThread().isDaemon());
      }
      public static void main(String[] args)
      {
                DaemonThread t1 = new DaemonThread();
                DaemonThread t2 = new DaemonThread();
                t1.start();
                // Exception as the thread is already started
                t1.setDaemon(true);

                t2.start();
      }
}
```

Output:

```
Thread name: Thread-0
Check if its DaemonThread: false
```

Daemon vs User Threads

- **Priority:** When the only remaining threads in a process are daemon threads, the interpreter exits. This makes sense because when only daemon threads remain, there is no other thread for which a daemon thread can provide a service.
- **Usage:** Daemon thread is to provide services to user thread for background supporting task.

The following program is an example for daemon thread.

```
public class DaemonThread_example extends Thread{
 public void run(){
  if(Thread.currentThread().isDaemon()){//checking for daemon thread
   System.out.println("daemon thread work");
  }
  else{
```

```
  System.out.println("user thread work");
 }
 }
 public static void main(String[] args){
  TestDaemonThread1 t1=new TestDaemonThread1();//creating thread
  TestDaemonThread1 t2=new TestDaemonThread1();
  TestDaemonThread1 t3=new TestDaemonThread1();
  t1.setDaemon(true);//now t1 is daemon thread
  t1.start();//starting threads
  t2.start();
  t3.start();
 }
}
```

The following program is another example for daemon thread.

```
class DaemonThread1_example extends Thread{
 public void run(){
  System.out.println("Name: "+Thread.currentThread().getName());
  System.out.println("Daemon: "+Thread.currentThread().isDaemon());
 }
 public static void main(String[] args){
  TestDaemonThread2 t1=new TestDaemonThread2();
  TestDaemonThread2 t2=new TestDaemonThread2();
  t1.start();
  t1.setDaemon(true);//will throw exception here
  t2.start();
 }
}
```

4.6 THREAD GROUP IN JAVA

Java provides a convenient way to group multiple threads in a single object. In such way, we can suspend, resume or interrupt group of threads by a single method call. ThreadGroup creates a group of threads. It offers a convenient way to manage groups of threads as a unit. This is particularly valuable in situation in which you want to suspend and resume a number of related threads.

- The thread group form a tree in which every thread group except the initial thread group has a parent.
- A thread is allowed to access information about its own thread group but not to access information about its thread group’s parent thread group or any other thread group.

Constructors of ThreadGroup class

There are only two constructors of ThreadGroup class.

Constructor	Description
Thread Group (String name)	creates a thread group with given name.
Thread Group (ThreadGroup parent, String name)	creates a thread group with given parent group and name.

The following program is an example for ThreadGroup

```
public class ThreadGroup_example implements Runnable{
    public void run() {
          System.out.println(Thread.currentThread().getName());
    }
   public static void main(String[] args) {
      ThreadGroup_example runnable = new ThreadGroup_example();
          ThreadGroup tg1 = new ThreadGroup(“Parent ThreadGroup”);
           Thread t1 = new Thread(tg1, runnable,”one”);
          t1.start();
          Thread t2 = new Thread(tg1, runnable,”two”);
          t2.start();
          Thread t3 = new Thread(tg1, runnable,”three”);
          t3.start();
```

```
            System.out.println("Thread Group Name: "+tg1.getName());
           tg1.list();
         }
       }
```

Sample Output:

```
    one
    two
    three
    Thread Group Name: Parent ThreadGroup
    java.lang.ThreadGroup[name=Parent ThreadGroup,maxpri=10]
       Thread [one,5,Parent ThreadGroup]
       Thread [two,5,Parent ThreadGroup]
       Thread [three,5,Parent ThreadGroup]
```

Methods of ThreadGroup class

There are many methods in ThreadGroup class. A list of important methods are given below.

Method	Description
int activeCount()	returns no. of threads running in current group.
int activeGroupCount()	returns a no. of active group in this thread group.
void destroy()	destroys this thread group and all its sub groups.
String getName()	returns the name of this group.
ThreadGroup getParent()	returns the parent of this group.
void interrupt()	interrupts all threads of this group.
void list()	prints information of this group to standard console.

The following programs explain the threadgroup example.

// Java code illustrating activeCount() method

```
import java.lang.*;
class NewThread extends Thread
{
    NewThread(String threadname, ThreadGroup tgob)
    {
            super(tgob, threadname);
            start();
    }
public void run()
    {

            for (int i = 0; i < 1000; i++)
            {
                    try
                    {
                            Thread.sleep(10);
                    }
                    catch (InterruptedException ex)
                    {
                            System.out.println("Exception encounterted");
                    }
            }
    }
}
public class ThreadGroup_example
{
    public static void main(String arg[])
    {
```

```
                // creating the thread group
                ThreadGroup gfg = new ThreadGroup(“parent thread group”);

                NewThread t1 = new NewThread(“one”, gfg);
                System.out.println(“Starting one”);
                NewThread t2 = new NewThread(“two”, gfg);
                System.out.println(“Starting two”);

                // checking the number of active thread
                System.out.println(“number of active thread: “
                                                + gfg.activeCount());
        }
    }
```

Output:

```
Starting one
Starting two
number of active thread: 2
```

UNIT-5

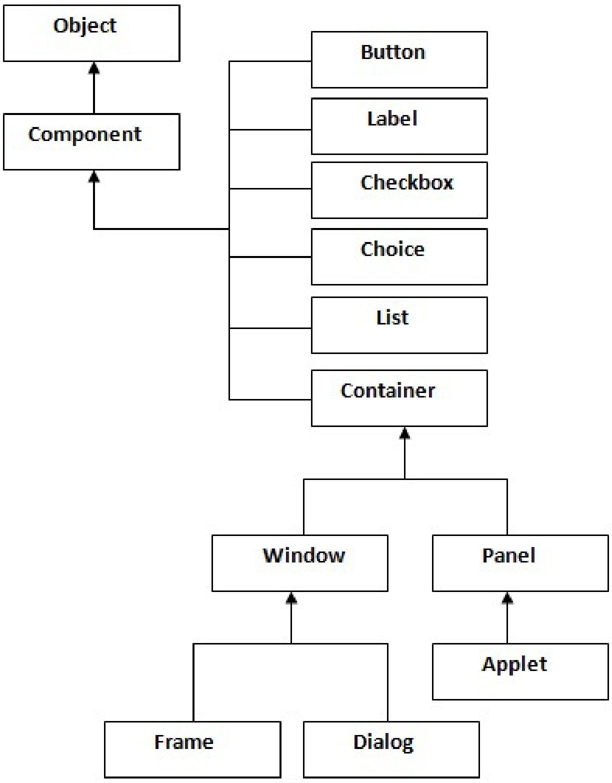
EVENT DRIVEN PROGRAMMING

Java AWT

The Abstract Window Toolkit (AWT) is Java's original platform-independent windowing, graphics, and user-interface widget toolkit. The AWT classes are contained in the java.awt package.

- Contains all of the classes for creating user interfaces and for painting graphics and images.
- *an API to develop GUI or window-based applications* in java.

The hierarchy of Java AWT classes are shown below.



Component

A component is an object having a graphical representation that can be displayed on the screen and that can interact with the user.

Examples :

buttons, checkboxes, and scrollbars

The Component class is the abstract superclass of all user interface elements that are displayed on the screen. A Component object remembers current text font, foreground and background color.

Container

The Container class is the subclass of Component. The container object is a component that can contain other AWT components. It is responsible for laying out any components that it contains.

Window

The class Window is a top level window with no border and no menubar. The default layout for a window is `BorderLayout`. A window must have either a frame, dialog, or another window defined as its owner when it's constructed.

Panel

The class Panel is the simplest container class. It provides space in which an application can attach any other component, including other panels. The default layout manager for a panel is the FlowLayout layout manager

Frame

A Frame is a top-level window with a title and a border. It uses BorderLayout as default layout manager.

Dialog

A Dialog is a top-level window with a title and a border that is typically used to take some form of input from the user.

Canvas

A Canvas component represents a blank rectangular area of the screen onto which the application can draw or from which the application can trap input events from the user. An application must subclass the Canvas class in order to get useful functionality such as creating a custom component. The paint method must be overridden in order to perform custom graphics on the canvas. It is not a part of hierarchy of Java AWT.

java.awt.Graphics class

The java.awt.Graphics class provides many methods for graphics programming. A graphics context is encapsulated by the Graphics class and is obtained in two ways:

- It is passed to an applet when one of its various methods, such as paint( ) or update( ) is called.
- It is returned by the getGraphics( ) method of Component.

Graphics Methods

The commonly used methods of Graphics class are as follows.

Method	Description
abstract Graphics create()	Creates a new Graphics object that is a copy of this Graphics object
abstract void drawString(String str, int x, int y)	Draws the text given by the specified string
void drawRect(int x, int y, int width, int height)	draws a rectangle with the specified width and height
void draw3DRect(int x, int y, int width, int height, boolean raised)	Draws a 3-D highlighted outline of the specified rectangle.
abstract void drawRoundRect(int x, int y, int width, int height, int arcWidth, int arcHeight)	Draws an outlined round-cornered rectangle using this graphics context's current color
abstract void fillRect(int x, int y, int width, int height)	fill rectangle with the default color and specified width and height.
abstract void drawPolygon(int[] xPoints, int[] yPoints, int nPoints)	Draws a closed polygon defined by arrays of x and y coordinates.
abstract void fillPolygon(int[] xPoints, int[] yPoints, int nPoints)	Fills a closed polygon defined by arrays of x and y coordinates.
abstract void drawOval(int x, int y, int width, int height)	draw oval with the specified width and height.
abstract void fillOval(int x, int y, int width, int height)	fill oval with the default color and specified width and height.
abstract void drawLine(int x1, int y1, int x2, int y2)	draw line between the points(x1, y1) and (x2, y2).
abstract boolean drawImage(Image img, int x, int y, ImageObserver observer)	draw the specified image.
abstract void drawArc(int x, int y, int width, int height, int startAngle, int arc Angle)	draw a circular or elliptical arc.

abstract void fillArc(int x, int y, int width, int height, int startAngle, int arcAngle)	fill a circular or elliptical arc.
abstract void setColor(Color c)	set the graphics current color to the specified color.
abstract void setFont(Font font)	set the graphics current font to the specified font.

Example:

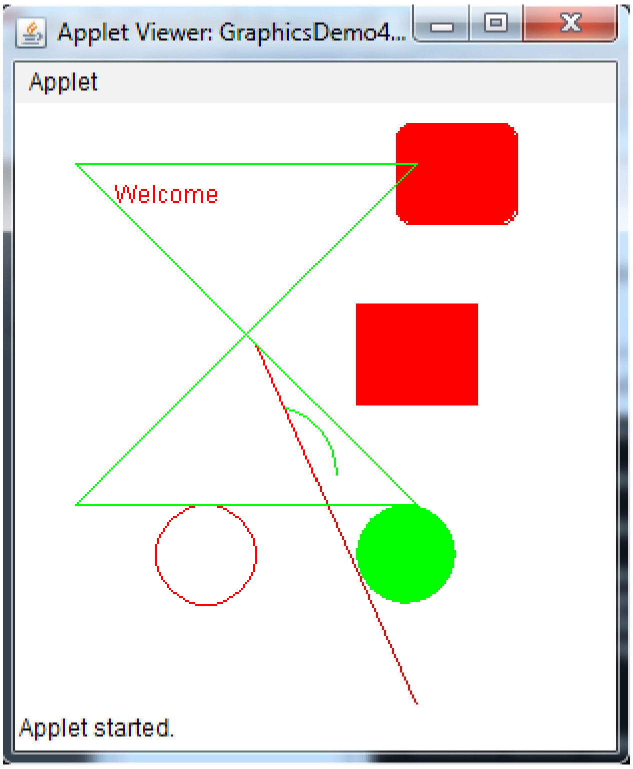
GraphicsDemo.java

```
import java.applet.Applet;
import java.awt.*;
public class GraphicsDemo extends Applet{
public void paint(Graphics g){
g.setColor(Color.red);        // set font color
g.drawString("Welcome",50, 50);  // display text
g.drawLine(120,120,200,300);                    // draw a line
// draw and fill rectangle
g.drawRect(170,100,60,50);
g.fillRect(170,100,60,50);
// draw and fill rounded rectangle
g.drawRoundRect(190, 10, 60, 50, 15, 15);
g.fillRoundRect(190, 10, 60, 50, 15, 15);
// draw and fill oval
g.drawOval(70,200,50,50);
g.setColor(Color.green);
g.fillOval(170,200,50,50);
// draw and fill arc
g.drawArc(90,150,70,70,0,75);
g.fillArc(270,150,70,70,0,75);
// draw a polygon
int xpoints[] = {30, 200, 30, 200, 30};
int ypoints[] = {30, 30, 200, 200, 30};
int num = 5;
g.drawPolygon(xpoints, ypoints, num);
}
}
```

<u>Test.html</u>

```
<html>
<body>
<applet code=”GraphicsDemo4.class” width=”300” height=”300”>
</applet>
</body>
</html>
```

Sample Output:



Note:

Steps to be followed to compile and run applet in DOS.

1. Compile the java file using javac command (for example, javac GraphicsDemo. java).
2. Create a separate html file. Mention the name of the java class file in the applet code parameter (for example code=”GraphicsDemo.class”)
3. Run the html file using appletviewer command (for example, appletviewer test.html)

Frames

A Frame is a top-level window with a title and a border. Frames are capable of generating the following types of window events: WindowOpened, WindowClosing, WindowClosed, WindowIconified, WindowDeiconified, WindowActivated, WindowDeactivated.

Frame Constructor

Frame()

Constructs a new instance of `Frame` that is initially invisible.

Frame(String)

Constructs a new, initially invisible `Frame` object with the specified title.

Some of the commonly used methods of Frame class are as follows.

Methods	Description
String getTitle()	Gets the title of the frame.
void setBackground(Color bgColor)	Sets the background color of this window.
void setResizable (boolean resizable)	Sets whether this frame is resizable by the user.
void setShape (Shape shape)	Sets the shape of the window.
void setTitle (String title)	Sets the title for this frame to the specified string.
void setSize (Dimension d)	Resizes this component so that it has width d. width and height d.height.
void setVisible(boolean b)	Shows or hides this Window depending on the value of parameter b.
public void show()	Makes the Window visible
void setMenuBar (MenuBar) mb)	Sets the menu bar for this frame to the specified menu bar

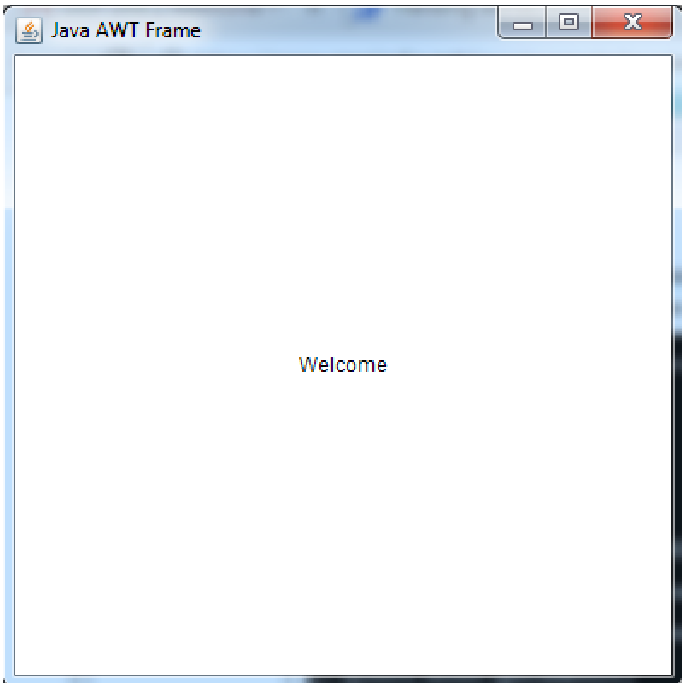
Creating a Frame

We can generate a window by creating an instance of Frame. The created frame can be made visible by calling setVisible( ). When created, the window is given a default height and width. The size of the window can be changed explicitly by calling the setSize( ) method. A label can be added to the current frame by creating an Label instance and calling the add() method.

Example:

```
import java.awt.*;

public class AwtFrame{

public static void main(String[] args){

Frame frm = new Frame("Java AWT Frame");

Label lbl = new Label("Welcome",Label.CENTER);

frm.add(lbl);

frm.setSize(400,400);

frm.setVisible(true);

  }

}
```

Sample Output:



Welcome

Creating an Frame Window in an Applet

The steps to be followed to create a child frame within an applet are as follows.

1. Create a subclass of Frame
2. Override any of the standard window methods, such as init(),start(),stop(),and paint().
3. Implement the windowClosing() method of the windowListener interface,calling setVisible(false) when the window is closed
4. Once you have defined a Frame subclass, you can create an object of that class. But it will note be initially visible
5. When created, the window is given a default height and width
6. You can set the size of the window explicitly by calling the setSize() method

Example:

AppletFrame.java

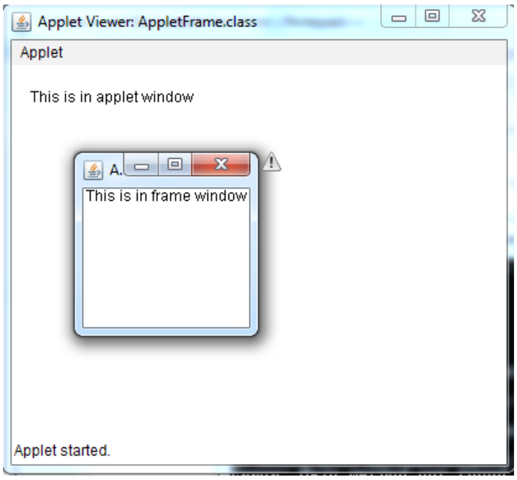
```
// Create a child frame window from within an applet.
import java.awt.*;
import java.awt.event.*;
import java.applet.*;
```

```
// Create a subclass of Frame.
class SampleFrame extends Frame {
SampleFrame(String title) {
super(title);
// create an object to handle window events
MyWindowAdapter adapter = new MyWindowAdapter(this);
// register it to receive those events
addWindowListener(adapter);
}
public void paint(Graphics g) {
g.drawString(“This is in frame window”, 10, 40);
}
}
class MyWindowAdapter extends WindowAdapter {
SampleFrame sampleFrame;
public MyWindowAdapter(SampleFrame sampleFrame) {
this.sampleFrame = sampleFrame;
}
public void windowClosing(WindowEvent we) {
sampleFrame.setVisible(false);
}
}
// Create frame window.
public class AppletFrame extends Applet {
Frame f;
//init(), start(), paint(), and stop() methods are called automatically in the specified sequence.
public void init() {
f = new SampleFrame(“A Frame Window”);
f.setSize(150, 150);
f.setVisible(true);
}
public void start() {
f.setVisible(true);   // make the window visible
}
public void stop() {
f.setVisible(false);  // hide the window
}
public void paint(Graphics g) {
g.drawString(“This is in applet window”, 15, 30);  // Display the given text in the window
}
}
```

Test1.html

```
<html>
<body>
<applet code=”AppletFrame.class” width=”400” height=”300”>
</applet>
</body>
</html>
```

Sample Output:



This is in applet window

This is in frame window

Applet started.

Components

Java AWT Component classes exist in java.awt package. The Component class is a super class of all components such as buttons, checkboxes, scrollbars, etc.

Component class constructor:

Component()   // constructs a new component

Properties of Java AWT Components:

- A Component object represents a graphical interactive area displayable on the screen that can be used by the user.
- Any subclass of a Component class is known as a component. For example, button is a component.
- Only components can be added to a container, like frame.

Some of the commonly used methods of Component class are as follows.

Method	Description
setBackground(Color)	Sets the background color of this component.
setBounds(int, int, int, int)	Moves and resizes this component.
setEnabled(boolean)	Enables or disables this component, depending on the value of the parameter b.
setFont(Font)	Sets the font of this component.
setForeground(Color)	Sets the foreground color of this component.
setLocation(int, int)	Moves this component to a new location.
setSize(int, int)	Resizes this component so that it has width width and height.
setVisible(boolean)	Shows or hides this component depending on the value of parameter b.
update(Graphics)	Updates this component.
repaint()	Repaints this component.
repaint(int, int, int, int)	Repaints the specified rectangle of this component.
add(Component c)	Inserts a component on this component.
remove(Component c)	Removes the specified component from this component.

Working with 2D shapes

Java supports 2-dimensional shapes, text and images using methods available in Graphics2D class. The Graphics2D class extends the Graphics class to provide more sophisticated control over geometry, coordinate transformations, color management, and text layout.

Graphics2D class Constructor

Graphics2D()   //Constructs a new Graphics2D object.

This class inherits the methods from java.lang.Object. Some of the commonly used methods of Graphics2D class are as follows.

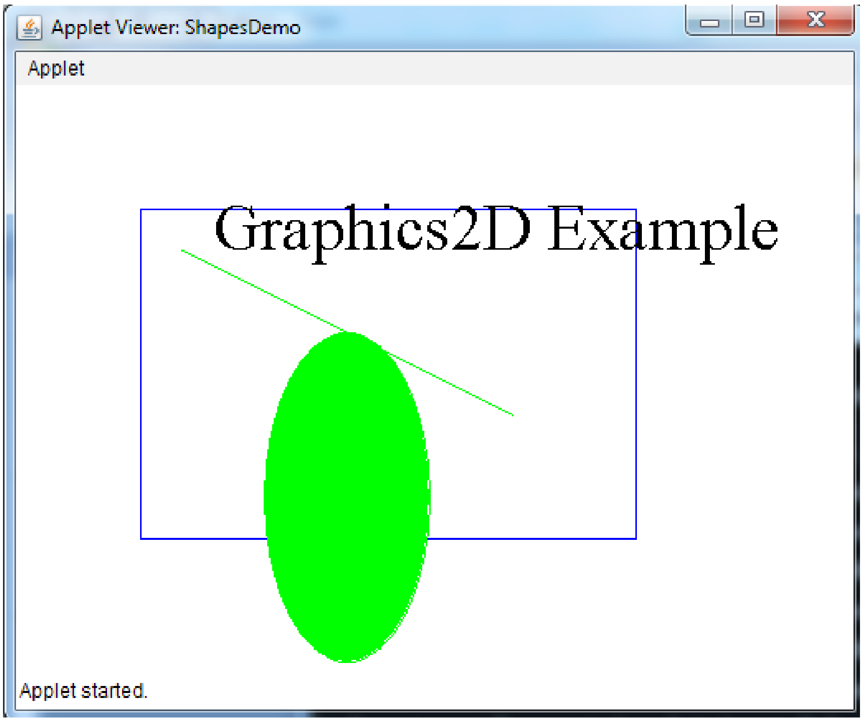
Method	Description
void draw(Shape s)	Strokes the outline of a Shape using the settings of the current Graphics2D context
void draw3DRect(int x, int y, int width, int height, boolean raised)	Draws a 3-D highlighted outline of the specified rectangle.
void drawImage(BufferedImage img, BufferedImageOp op, int x, int y)	Renders a BufferedImage that is filtered with a BufferedImageOp.
boolean drawImage(Image img, Affine Transform xform, ImageObserver obs)	Renders an image, applying a transform from image space into user space before drawing.
void drawString(String str, float x, float y)	Renders the text specified by the specified String, using the current text attribute state in the Graphics2D context
void fill(Shape s)	Fills the interior of a Shape using the settings of the Graphics2D context.
void rotate(double theta)	Concatenates the current Graphics2D Transform with a rotation transform.
void scale(double sx, double sy)	oncatenates the current Graphics2D Transform with a scaling transformation Subsequent rendering is resized according to the specified scaling factors relative to the previous scaling.
void setBackground(Color color)	Sets the background color for the Graphics2D context.
void setPaint(Paint paint)	Sets the Paint attribute for the Graphics2D context.
void setStroke(Stroke s)	Sets the Stroke for the Graphics2D context.
void shear(double shx, double shy)	Concatenates the current Graphics2D Transform with a shearing transform.
void transform(AffineTransform Tx)	Composes an AffineTransform object with the Transform in this Graphics2D according to the rulelast-specified-first-applied.

void translate(int x, int y)	Translates the origin of the Graphics2D context to the point (x, y) in the current coordinate system.

Example:

```
import java.awt.*;
import java.applet.*;
/*
<applet code=”ShapesDemo” width=350 height=300>
</applet>
*/
public class ShapesDemo extends Applet {
public void init() {}
 public void paint(Graphics g) {
Graphics2D g2d = (Graphics2D)g;
g2d.setColor(Color.blue);
g2d.drawRect(75,75,300,200);
Font exFont = new Font(“TimesRoman”,Font.PLAIN,40);
g2d.setFont(exFont);
g2d.setColor(Color.black);
g2d.drawString(“Graphics2D Example”,120.0f,100.0f);
g2d.setColor(Color.green);
g2d.drawLine(100,100,300,200);
g2d.drawOval(150,150,100,200);
g2d.fillOval(150,150,100,200);
}
}
```

Sample Output:



Graphics2D Example

Applet started.

Colors in Java

To support different colors Java package comes with the Color class. The Color class states colors in the default sRGB color space or colors in arbitrary color spaces identified by a ColorSpace.

Color class static color variables available are:

Color.black
Color.blue
Color.cyan
Color.darkGray
Color.gray
Color.green
Color.yellow
Color.lightGray
Color.magenta
Color.orange
Color.pink
Color.red
Color.white
.

Color class constructor

Color(float r, float g, float b) – create color with specified red, green, and blue values in the range (0.0 - 1.0)

Color(int r, int g, int b)- create color with the specified red, green, and blue values in the range (0 - 255).

Some of the commonly used methods supported by the Color class are as follows.

Method	Description
int getRed()	Returns the red component in the range 0-255 in the default sRGB space.
int getGreen()	Returns the green component in the range 0-255 in the default sRGB space.
int getBlue()	Returns the blue component in the range 0-255 in the default sRGB space.
Color getHSBColor(float h, float s, float b)	Creates a Color object based on the specified values for the HSB color model.

The current graphics color can be changed using setColor() method defined in Graphics class.

```
void setColor(Color newColor)          // newColor indicates new drawing color
```

The current color detail can be obtained using getColor() method. Its syntax is.

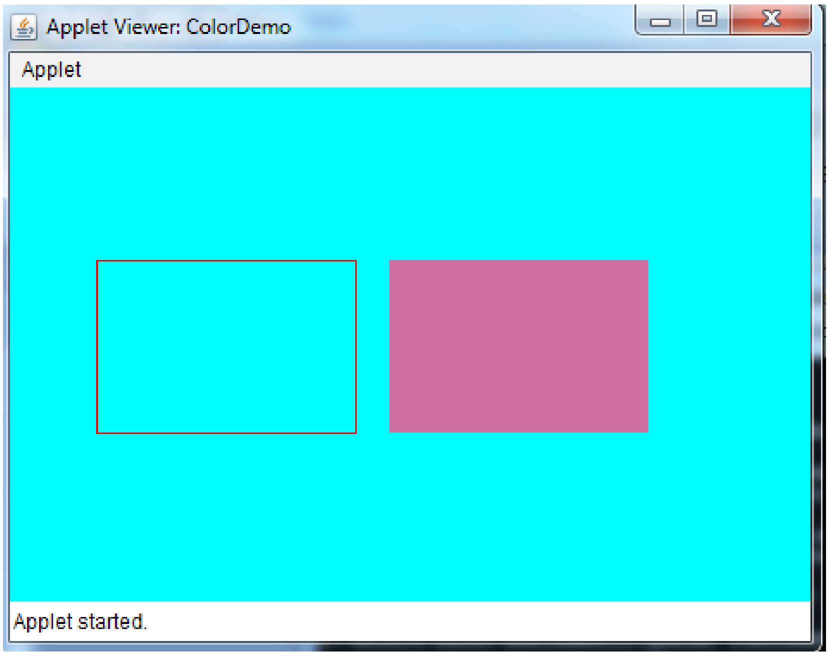
```
Color getColor()
```

Example:

```
import java.awt.*;
import java.applet.*;
/*
<applet code=”ColorDemo” width=350 height=300>
</applet>
*/
public class ColorDemo extends Applet {
      public void init() {
setBackground(Color.CYAN);
}
 public void paint(Graphics g) {
    g.setColor(Color.red);          // predefined color
    g.drawRect(50, 100, 150, 100);   // rectangle outline is red color
    Color clr = new Color(200, 100, 150);
    g.setColor(clr);
```

```
        g.fillRect(220,100, 150, 100);   // rectangle filled with clr color
    }
    }
```

Sample Output:



Fonts in Java

The Font class states fonts, which are used to render text in a visible way.

Font class constructor

Font(Font font)   //Creates a new Font from the specified font.

Font(String name, int style, int size)   //Creates a new Font from the specified name, style and point size.

Font variables available in Font class are:

Font.BOLD	Font. SANS_SERIF
Font.ITALIC	Font. CENTER_BASELINE
Font. PLAIN	Font. DIALOG
Font. MONOSPACED	Font. SERIF
Font. TRUETYPE_FONT	Font. TYPE1_FONT
int size	int style
float pointSize	String name

Some of the commonly used methods supported by the Font class are as follows.

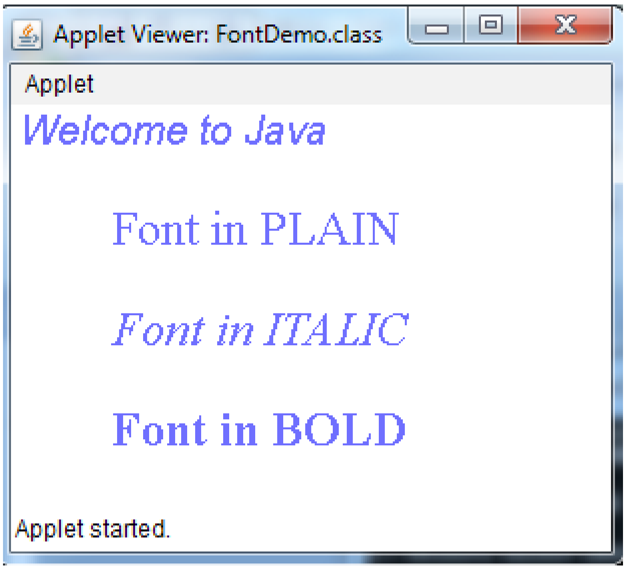
Method	Description
String getFamily()	Returns the family name of this Font.
int getStyle()	Returns the style of this Font.
boolean isBold()	Indicates whether or not this Font object's style is BOLD
boolean isItalic()	Indicates whether or not this Font object's style is ITALIC.
boolean isPlain()	Indicates whether or not this Font object's style is PLAIN.
static Font getFont(String nm)	Returns a Font object fom the system properties list.
static Font decode(String str)	Returns the Font that the str argument describes.
String toString()	Converts this Font object to a String representation.

Example:

```
import java.applet.Applet;
import java.awt.*;
import java.awt.event.*;
/* <APPLET CODE =”FontDemo.class” WIDTH=300 HEIGHT=200> </APPLET> */
public class FontDemo extends java.applet.Applet
  {
      Font f;
      String m;
       public void init()
         {
            f=new Font(“Arial”,Font.ITALIC,20);
            m=”Welcome to Java”;
            setFont(f);
         }
            public void paint(Graphics g)
               {
                   Color c=new Color(100,100,255);
                   g.setColor(c);
                   g.drawString(m,4,20);
Font plainFont = new Font(“Serif”, Font.PLAIN, 24);
```

```
  g.setFont(plainFont);
   g.drawString(“Font in PLAIN”, 50, 70);
   Font italicFont = new Font(“Serif”, Font.ITALIC, 24);
   g.setFont(italicFont);
   g.drawString(“Font in ITALIC”, 50, 120);
   Font boldFont = new Font(“Serif”, Font.BOLD, 24);
   g.setFont(boldFont);
   g.drawString(“Font in BOLD”, 50, 170);
   Font boldItalicFont = new Font(“Serif”, Font.BOLD+Font.ITALIC, 24);
   g.setFont(boldItalicFont);
   g.drawString(“Font in BOLD ITALIC”, 50, 220);
                  }
    }
```

Sample Output:



```
Welcome to Java

Font in PLAIN

Font in ITALIC

Font in BOLD

Applet started.
```

Images in Java

Image control is superclass for all image classes representing graphical images.

Image class constructor

Image()   // create an Image object

Some of the commonly used methods supported by the Image class are as follows.

Method	Description
Graphics getGraphics()	Creates a graphics context for drawing to an off-screen image.
int getHeight(ImageObserver observer)	Determines the height of the image.
Image getScaledInstance(int width, int height, int hints)	Creates a scaled version of this image.
ImageProducer getSource()	Gets the object that produces the pixels for the image.
int getWidth(ImageObserver observer)	Determines the width of the image.

The java.applet.Applet class provides following methods to access image.

1. getImage() method that returns the object of Image. Its syntax is as follows.

   public Image getImage(URL u, String image){}

2. getDocumentBase() method returns the URL of the document in which applet is embedded.

   public URL getDocumentBase(){}

3. URL getCodeBase() method returns the base URL.

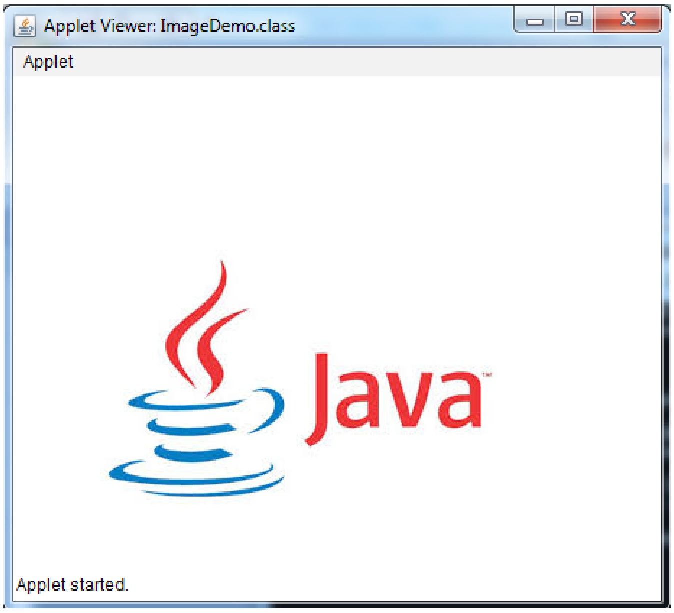
   public URL getCodeBase()

Example:

```
import java.applet.Applet;
import java.awt.*;
import java.awt.event.*;
import java.net.URL;
/* <APPLET CODE =”ImageDemo.class” WIDTH=300 HEIGHT=200> </APPLET> */
public class ImageDemo extends java.applet.Applet
  {
     Image img;
      public void init()
        {
         }
           public void paint(Graphics g)
              {
```

```
                                URL url1 = getCodeBase();
                                img = getImage(url1,"java.jpg");
                            g.drawImage(img, 60, 120, this);
                }
    }
```

Sample Output:



Event Handling

Any change in the state of any object is called event. For Example: Pressing a button, entering a character in Textbox, Clicking or dragging a mouse, etc. The three main components in event handling are:

- ***Events:*** An event is a change in state of an object. For example, mouseClicked, mousePressed.
- ***Events Source:*** Event source is an object that generates an event. Example: a button, frame, textfield.
- ***Listeners:*** A listener is an object that listens to the event. A listener gets notified when an event occurs. When listener receives an event, it process it and then return. Listeners are group of interfaces and are defined in java.awt.event package. Each component has its own listener. For example MouseListener handles all MouseEvent.

Some of the event classes and Listener interfaces are listed below.

Event Classes	**Generated when**	**Listener Interfaces**
ActionEvent	button is pressed, menu-item is selected, list-item is double clicked	Action Listener
MouseEvent	mouse is dragged, moved, clicked, pressed or released and also when it enters or exit a component	Mouse Listener and Mouse Motion Listener
MouseWheelEvent	mouse wheel is moved	Mouse Wheel Listener
KeyEvent	input is received from keyboard	Key Listener
ItemEvent	check-box or list item is clicked	Item Listener
TextEvent	value of textarea or textfield is changed	Text Listener
AdjustmentEvent	scroll bar is manipulated	Adjustment Listener
WindowEvent	window is activated, deactivated, deiconified, iconified, opened or closed	Window Listener
ComponentEvent	component is hidden, moved, resized or set visible	Component Listener
ContainerEvent	component is added or removed from container	Container Listener
FocusEvent	component gains or losses keyboard focus	Focus Listener

Java program for handling keyboard events.

Test.java

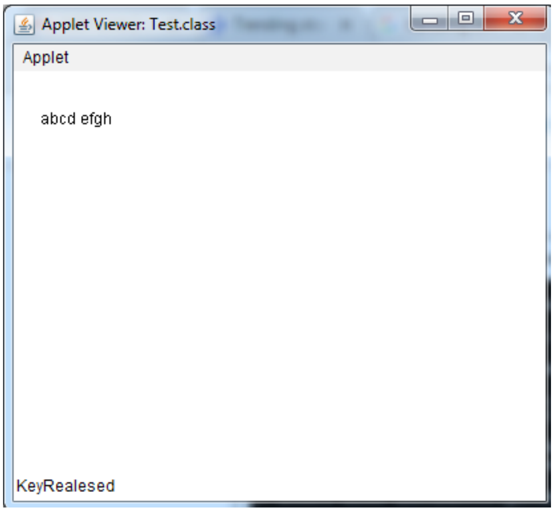
```
import java.awt.event.*;
import java.applet.*;
import java.applet.*;
import java.awt.event.*;
import java.awt.*;
//Implementing KeyListener interface to handle keyboard events
public class Test extends Applet implements KeyListener
{
 String msg="”;
 public void init()
 {
   addKeyListener(this);                //use keyListener to monitor key events
```

```
  }
  public void keyPressed(KeyEvent k)        // invoked when any key is pressed down
  {
    showStatus(“KeyPressed”);
  }
  public void keyReleased(KeyEvent k)       // invoked when  key is released
  {
    showStatus(“KeyRealesed”);
  }
 //keyTyped event is called first followed by key pressed or key released event
  public void keyTyped(KeyEvent k)          //invoked when a textual key is pressed
  {
    msg = msg+k.getKeyChar();
    repaint();
  }
  public void paint(Graphics g)
  {
    g.drawString(msg, 20, 40);
  }
 }
```

<u>Test1.html</u>

```
<html>
<body>
<applet code=”Test.class” width=”400” height=”300”>
</applet>
</body>
</html>
```

Sample Output:



```
abcd efgh
```

```
KeyRealesed
```

Adapter Classes

An adapter class provides the default implementation of all methods in an event listener interface. Adapter classes are very useful when you want to process only few of the events that are handled by a particular event listener interface. For example MouseAdapter provides empty implementation of MouseListener interface. It is useful because very often you do not really use all methods declared by interface, so implementing the interface directly is very lengthy.

- Adapter class is a simple java class that implements an interface with only EMPTY implementation.
- Instead of implementing interface if we extends Adapter class ,we provide implementation only for require method

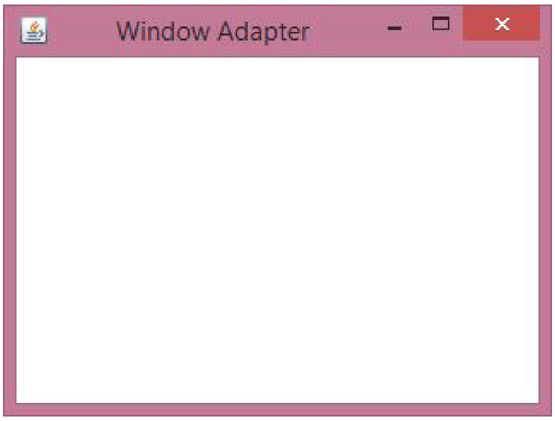
The adapter classes are found in **java.awt.event**, **java.awt.dnd** and **javax.swing.event** packages. The Adapter classes with their corresponding listener interfaces are as follows.

Adapter Class	Listener Interface
Window Adapter	Window Listener
Key Adapter	Key Listener
Mouse Adapter	Mouse Listener
Mouse Motion Adapter	Mouse Motion Listener
Focus Adapter	Focus Listener
Component Adapter	Component Listener
Container Adapter	Container Listener
HierarchyBoundsAdapter	HierarchyBoundsListener

Example:

```
import java.awt.*;
import java.awt.event.*;
public class AdapterExample{
   Frame f;
   AdapterExample(){
      f=new Frame("Window Adapter");
      f.addWindowListener(new WindowAdapter(){
         public void windowClosing(WindowEvent e) {
            f.dispose();
         }
      });

      f.setSize(400,400);
      f.setLayout(null);
      f.setVisible(true);
   }
public static void main(String[] args) {
   new AdapterExample();
}
}
```

Sample Output:



Actions

The Java Action interface and AbstractAction class are terrific ways of encapsulating behaviors (logic), especially when an action can be triggered from more than one place in your Java/Swing application.

javax.swing

Interface Action

An Action can be used to separate functionality and state from a component. For example, if you have two or more components that perform the same function, consider using an Action object to implement the function.

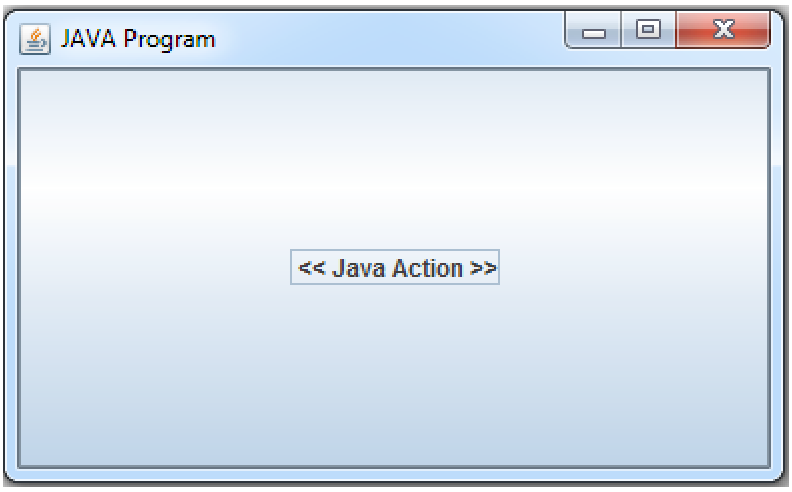
An Action object is an action listener that provides not only action-event handling, but also centralized handling of the state of action-event-firing components such as tool bar buttons, menu items, common buttons, and text fields. The state that an action can handle includes text, icon, mnemonic, enabled, and selected status.

The most common way an action event can be triggered from multiple places in a Java/Swing application is through the Java menubar (JMenuBar) and toolbar (JToolBar)

```
import java.awt.event.ActionEvent;
import java.awt.event.ActionListener;
import javax.swing.JButton;
import javax.swing.JFrame;
 public class ButtonAction {
    private static void createAndShowGUI()  {
       JFrame frame1 = new JFrame("JAVA Program");
      frame1.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
       JButton button = new JButton(" << Java Action >>");
      //Add action listener to button
      button.addActionListener(new ActionListener() {
         public void actionPerformed(ActionEvent e)
         {
            System.out.println("You clicked the button");
         }
      });
       frame1.getContentPane().add(button);
```

```
        frame1.pack();
        frame1.setVisible(true);
    }
    public static void main(String[] args) {
        javax.swing.SwingUtilities.invokeLater(new Runnable() {
            public void run() {
                createAndShowGUI();
            }
        });
    }
```

Output:



MouseEvent:

An event which indicates that a mouse action occurred in a component. A mouse action is considered to occur in a particular component if and only if the mouse cursor is over the unobscured part of the component's bounds when the action happens. For lightweight components, such as Swing's components, mouse events are only dispatched to the component if the mouse event type has been enabled on the component.

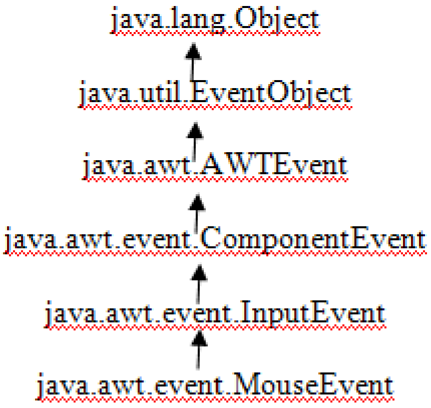
A mouse event type is enabled by adding the appropriate mouse-based EventListener to the component (Mouse Listener or Mouse Motion Listener), or by invoking Component.enableEvents (long) with the appropriate mask parameter

(AWTEvent.MOUSE_EVENT_MASK or AWTEvent.MOUSE_MOTION_EVENT_MASK).

If the mouse event type has not been enabled on the component, the corresponding mouse events are dispatched to the first ancestor that has enabled the mouse event type.Iif a MouseListener has been added to a component, or enableEvents(AWTEvent.MOUSE_EVENT_MASK) has been invoked, then all the events defined by MouseListener are dispatched to the component.

On the other hand, if MouseMotionListener has not been added and enableEvents has not been invoked with AWTEvent.MOUSE_MOTION_EVENT_MASK, then mouse motion events are not dispatched to the component. Instead the mouse motion events are dispatched to the first ancestor that has enabled mouse motion events.

The hierarchy of MouseEvent class is shown below.


java.lang.Object
java.util.EventObject
java.awt.AWTEvent
java.awt.event.ComponentEvent
java.awt.event.InputEvent
java.awt.event.MouseEvent

- Mouse Events are
  - a mouse button is pressed
  - a mouse button is released
  - a mouse button is clicked (pressed and released)
  - the mouse cursor enters the unobscured part of component’s geometry
  - the mouse cursor exits the unobscured part of component’s geometry
- Mouse Motion Events are
  - the mouse is moved
  - the mouse is dragged

A MouseEvent object is passed to every MouseListener or MouseAdapter object which is registered to receive the “interesting” mouse events using the component’s addMouseListener method. The MouseAdapter objects implement the MouseListener interface. Each such listener object gets a MouseEvent containing the mouse event.

A MouseEvent object is also passed to every MouseMotionListener or MouseMotionAdapter object which is registered to receive mouse motion events using the component's addMouseMotionListener method. (MouseMotionAdapter objects implement the MouseMotionListener interface.) Each such listener object gets a MouseEvent containing the mouse motion event.

When a mouse button is clicked, events are generated and sent to the registered MouseListeners. The state of modal keys can be retrieved using InputEvent.getModifiers() and InputEvent.getModifiersEx(). The button mask returned by InputEvent.getModifiers() reflects only the button that changed state, not the current state of all buttons.. To get the state of all buttons and modifier keys, use InputEvent.getModifiersEx(). The button which has changed state is returned by getButton().

For example, if the first mouse button is pressed, events are sent in the following order:

id modifiers button

MOUSE_PRESSED: BUTTON1_MASK BUTTON1

MOUSE_RELEASED: BUTTON1_MASK BUTTON1

MOUSE_CLICKED: BUTTON1_MASK BUTTON1

When multiple mouse buttons are pressed, each press, release, and click results in a separate event.

For example, if the user presses button 1 followed by button 2, and then releases them in the same order, the following sequence of events is generated:

id modifiers button

MOUSE_PRESSED: BUTTON1_MASK BUTTON1

MOUSE_PRESSED: BUTTON2_MASK BUTTON2

MOUSE_RELEASED: BUTTON1_MASK BUTTON1

MOUSE_CLICKED: BUTTON1_MASK BUTTON1

MOUSE_RELEASED: BUTTON2_MASK BUTTON2

MOUSE_CLICKED: BUTTON2_MASK BUTTON2

If **button 2** is released first, the MOUSE_RELEASED/MOUSE_CLICKED pair for BUTTON2_MASK arrives first, followed by the pair for BUTTON1_MASK.

MOUSE_DRAGGED events are delivered to the Component in which the mouse button was pressed until the mouse button is released (regardless of whether the mouse position is within the bounds of the Component). Due to platform-dependent Drag&Drop implementations, MOUSE_DRAGGED events may not be delivered during a native Drag&Drop operation.

In a multi-screen environment mouse drag events are delivered to the Component even if the mouse position is outside the bounds of the Graphics Configuration associated with that Component. However, the reported position for mouse drag events in this case may differ from the actual mouse position:

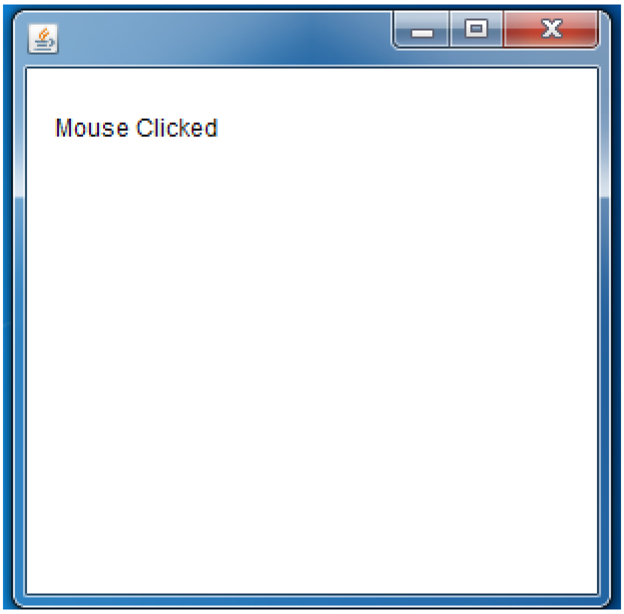
- In a multi-screen environment without a virtual device: The reported coordinates for mouse drag events are clipped to fit within the bounds of the GraphicsConfiguration associated with the Component.
- In a multi-screen environment with a virtual device: The reported coordinates for mouse drag events are clipped to fit within the bounds of the virtual device associated with the Component.

The following program is an example for MouseEvent.

```
  import java.awt.*;
  import java.awt.event.*;
          pubic class MouseListenerExample extends Frame implements MouseListener{
Label l;
MouseListenerExample(){
   addMouseListener(this);
   l=new Label();
   l.setBounds(20,50,100,20);
   add(l);
   setSize(300,300);
   setLayout(null);
   setVisible(true);
}
public void mouseClicked(MouseEvent e) {
   l.setText(“Mouse Clicked”);
}
public void mouseEntered(MouseEvent e) {
   l.setText(“Mouse Entered”);
}
public void mouseExited(MouseEvent e) {
   l.setText(“Mouse Exited”);
```

```
        }
        public void mousePressed(MouseEvent e) {
          l.setText("Mouse Pressed");
        }
        public void mouseReleased(MouseEvent e) {
          l.setText("Mouse Released");
        }
    public static void main(String[] args) {
        new MouseListenerExample();
    }
    }
```

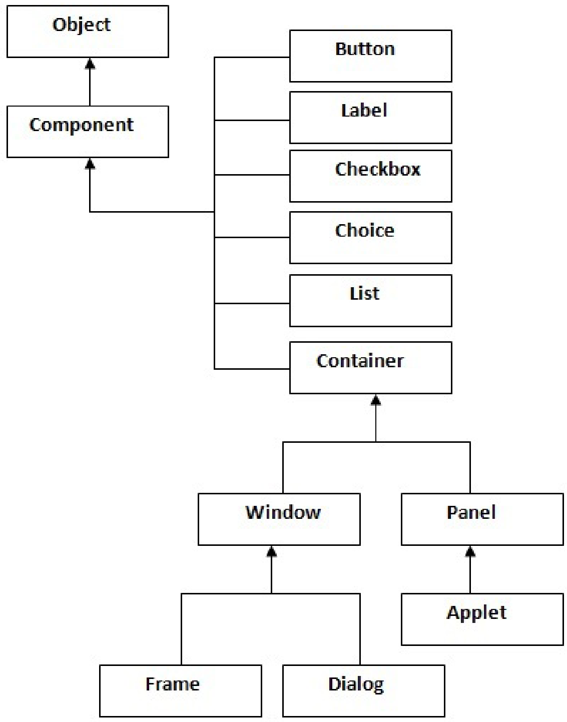
Output:



Mouse Clicked

Java AWT Hierarchy

The hierarchy of Java AWT classes are given below.



Container

The Container is a component in AWT that can contain another component like buttons, textfields, labels etc. The classes that extend Container class are known as container such as Frame, Dialog and Panel.

Window

The window is the container that has no borders and menu bars. You must use frame, dialog or another window for creating a window.

Panel

The Panel is the container that doesn't contain title bar and menu bars. It can have other components like button, textfield etc.

Frame

The Frame is the container that contain title bar and can have menu bars. It can have other components like button, textfield etc.

The following table gives the methods of Component class:

Method	Description
public void add(Component c)	inserts a component on this component.
public void setSize(int width,int height)	sets the size (width and height) of the component.
public void setLayout(Layout Manager m)	defines the layout manager for the component.
public void setVisible(boolean status)	changes the visibility of the component, by default false.

The following programs are the examples of Java AWT:

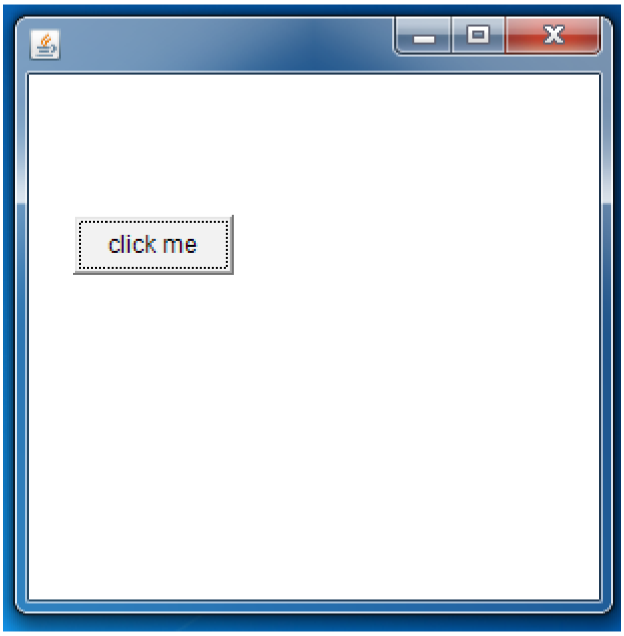
To create simple awt program, we need to create a frame. There are two ways to create a frame in AWT.

- By extending Frame class (inheritance)
- By creating the object of Frame class (association)

Example program using by extending Frame class (inheritance)

```
import java.awt.*;
class First extends Frame{
First(){
Button b=new Button("click me");
b.setBounds(30,100,80,30);// setting button position
add(b);//adding button into frame
setSize(300,300);//frame size 300 width and 300 height
setLayout(null);//no layout manager
setVisible(true);//now frame will be visible, by default not visible
}
public static void main(String args[]){
First f=new First();
}}
```

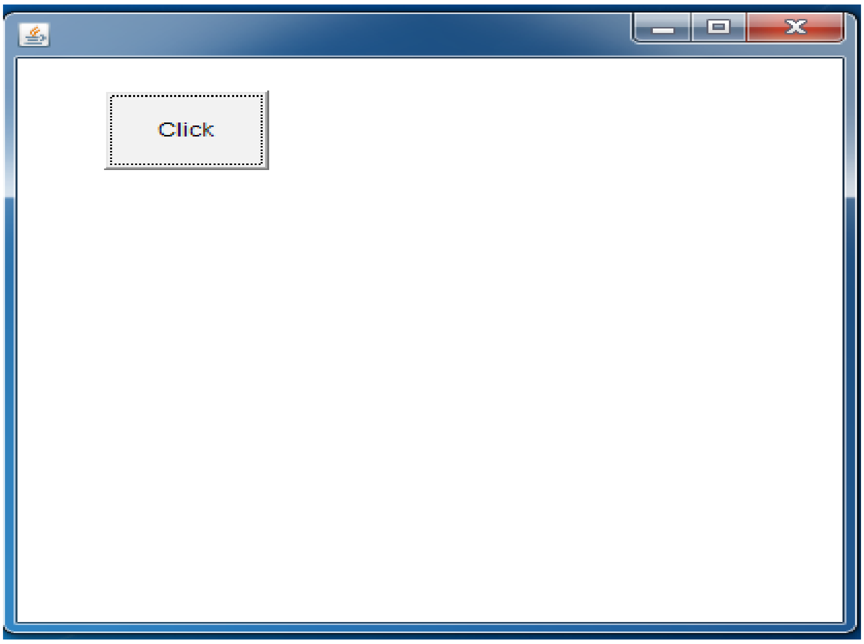
Output:



Example program using by creating the object of Frame class (association)

```
import java.awt.*;
class First2{
First2(){
Frame f=new Frame();
Button b=new Button(“click me”);
b.setBounds(30,50,80,30);
f.add(b);
f.setSize(300,300);
f.setLayout(null);
f.setVisible(true);
}
public static void main(String args[]){
First2 f=new First2();
}}
```

Output:



Java Swing

Swing was developed to provide a more sophisticated set of GUI components than the earlier Abstract Window Toolkit (AWT). Swing provides a look and feel that emulates the look and feel of several platforms, and also supports a pluggable look and feel that allows applications to have a look and feel unrelated to the underlying platform. It has more powerful and flexible components than AWT.

In addition to familiar components such as buttons, check boxes and labels, Swing provides several advanced components such as tabbed panel, scroll panes, trees, tables, and lists.

Unlike AWT components, Swing components are not implemented by platform-specific code. Instead, they are written entirely in Java and therefore are platform-independent. The term “lightweight” is used to describe such an element.

Java Swing is a part of Java Foundation Classes (JFC) that is *used to create window-based applications*. It is built on the top of AWT (Abstract Windowing Toolkit) API and entirely written in java.

Unlike AWT, Java Swing provides platform-independent and lightweight components.

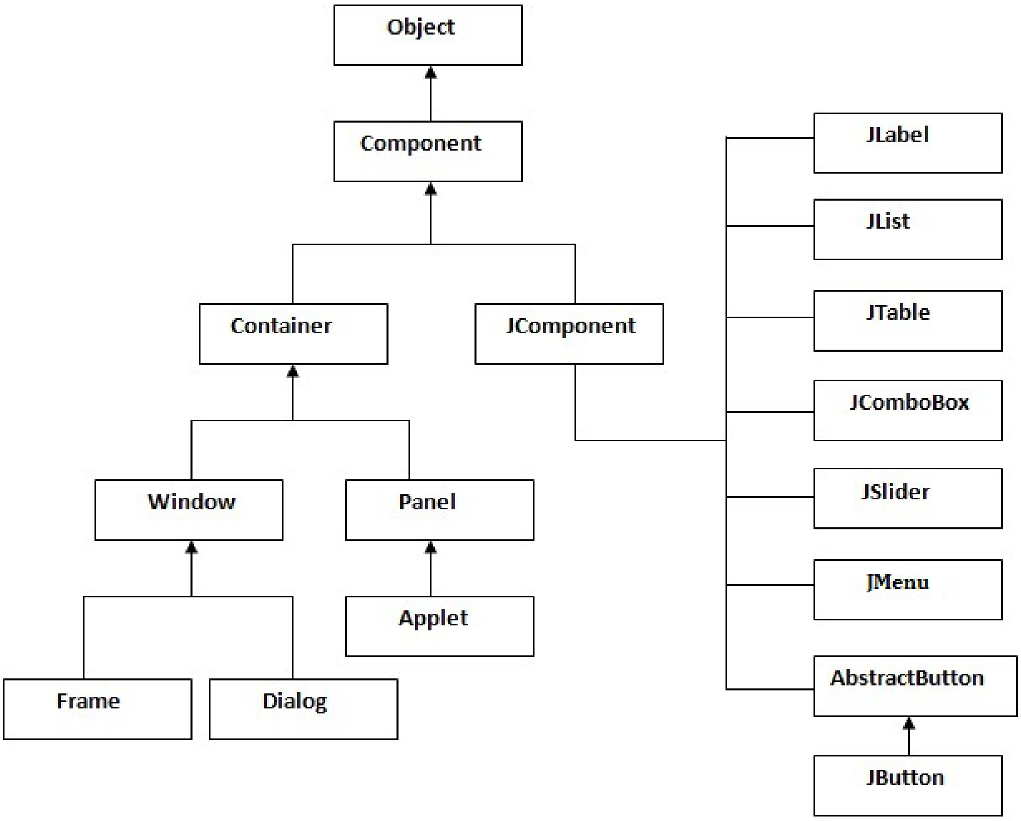
The javax.swing package provides classes for java swing API such as JButton, JTextField, JTextArea, JRadioButton, JCheckbox, JMenu, JColorChooser etc.

Swing Features

- ***Light Weight*** - Swing component are independent of native Operating System's API as Swing API controls are rendered mostly using pure JAVA code instead of underlying operating system calls.
- ***Rich controls*** - Swing provides a rich set of advanced controls like Tree, TabbedPane, slider, colourpicker, table controls
- ***Highly Customizable*** - Swing controls can be customized in very easy way as visual appearance is independent of internal representation.
- ***Pluggable look-and-feel***- SWING based GUI Application look and feel can be changed at run time based on available values.

Hierarchy of Java Swing classes

The hierarchy of java swing API is given below.



The following program is an example for Java Swing.

```
import javax.swing.*;
public class FirstSwingExample {
public static void main(String[] args) {
JFrame f=new JFrame();//creating instance of JFrame
Button b=new JButton(“click”);//creating instance of JButton
b.setBounds(130,100,100, 40);//x axis, y axis, width, height
 f.add(b);//adding button in JFrame
 f.setSize(400,500);//400 width and 500 height
f.setLayout(null);//using no layout managers
f.setVisible(true);//making the frame visible
}
}
```

Output: