INTRODUCTION TO OOP AND JAVA FUNDAMENTALS

Object Oriented Programming - Abstraction – objects and classes - Encapsulation- Inheritance- Polymorphism- OOP in Java – Characteristics of Java – The Java Environment **-** Java Source File - Structure Compilation. Fundamental Programming Structures in Java – Defining classes in Java – constructors, methods - access specifiers - static members - Comments, Data Types, Variables, Operators, Control Flow, Arrays, Packages - JavaDoc comments.

OBJECT-ORIENTED PROGRAMMING

Object-Oriented Programming (OOP) is a programming language model organized around objects rather than actions and data. An object-oriented program can be characterized as data controlling access to code. Concepts of OOPS

- Object
- Class
- Inheritance
- Polymorphism
- Abstraction
- Encapsulation

OBJECT

Object means a real word entity such as pen, chair, table etc. Any entity that has state and behavior is known as an object. Object can be defined as an instance of a class. An object contains an address and takes up some space in memory. Objects can communicate without knowing details of each other's data or code, the only necessary thing is that the type of message accepted and type of response returned by the objects.
An object has three characteristics:

- state: represents data (value) of an object.
- behavior: represents the behavior (functionality) of an object such as deposit, withdraw etc.
- identity: Object identity is typically implemented via a unique ID. The value of the ID is not visible to the external user. But, it is used internally by the JVM to identify each object uniquely.
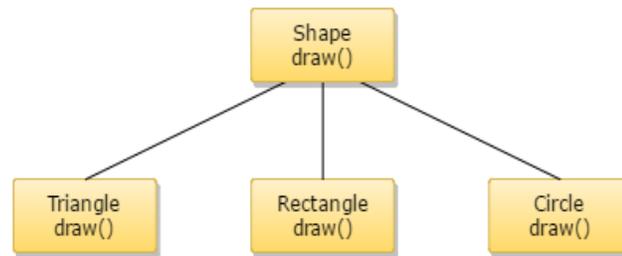
CLASS

Collection of objects is called class. It is a logical entity. A class can also be defined as a blueprint from which you can create an individual object. A class consists of Data members and methods.The primary purpose of a class is to hold data/information. The member functions determine the behavior of the class, i.e. provide a definition for supporting various operations on data held in the form of an object.Class doesn't store any space.

INHERITANCE

Inheritance can be defined as the procedure or mechanism of acquiring all the properties and behavior of one class to another, i.e., acquiring the properties and behavior of child class from the parent class. When one object acquires all the properties and behaviours of another object, it is known as inheritance. It provides code reusability and establishes relationships between different classes. A class which inherits the properties is known as Child Class(sub-class or derived class) whereas a class whose properties are inherited is known as Parent class(super-class or base class). Types of inheritance in java: single, multilevel and hierarchical inheritance. Multiple and hybrid inheritance is supported through interface only.

POLYMORPHISM

When one task is performed by different ways i.e. known as polymorphism. For example: to convince the customer differently, to draw something e.g. shape or rectangle etc.

Polymorphism is classified into two ways:

Method Overloading(Compile time Polymorphism)

Method Overloading is a feature that allows a class to have two or more methods having the same name but the arguments passed to the methods are different. 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 Overriding(Run time Polymorphism)

If subclass (child class) has the same method as declared in the parent class, it is known as method overriding in java.In other words, If subclass provides the specific implementation of the method that has been provided by one of its parent class, it is known as method overriding.

ABSTRACTION

Abstraction is a process of hiding the implementation details and showing only functionality to the user. For example: phone call, we don't know the internal processing.In java, we use abstract class and interface to achieve abstraction.

ENCAPSULATION

Encapsulation in java is a process of wrapping code and data together into a single unit, for example capsule i.e. mixed of several medicines.A java class is the example of encapsulation.
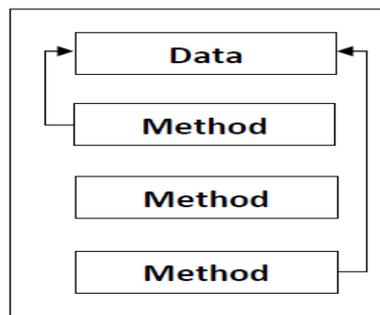
DIFFERENCE BETWEEN PROCEDURE-ORIENTED AND OBJECT-ORIENTED PROGRAMMING

Procedure-Oriented Programming	Object-Oriented Programming
In POP, program is divided into small parts called **functions**	In OOP, program is divided into parts called **objects**.
In POP,Importance is not given to **data** but to	In OOP, Importance is given to the data rather

functions as well as **sequence** of actions to be done.	than procedures or functions because it works as a **real world**.
POP follows **Top Down approach**.	OOP follows **Bottom Up approach**.
POP does not have any access specifier.	OOP has access specifiers named Public, Private, Protected, etc.
In POP, Data can move freely from function to function in the system.	In OOP, objects can move and communicate with each other through member functions.
To add new data and function in POP is not so easy.	OOP provides an easy way to add new data and function.
In POP, Most function uses Global data for sharing that can be accessed freely from function to function in the system.	In OOP, data can not move easily from function to function,it can be kept public or private so we can control the access of data.
POP does not have any proper way for hiding data so it is **less secure**.	OOP provides Data Hiding so provides **more security**.
In POP, Overloading is not possible.	In OOP, overloading is possible in the form of Function Overloading and Operator Overloading.
Example of POP are : C, VB, FORTRAN, Pascal.	Example of OOP are : C++, JAVA, VB.NET, C#.NET.

FEATURES OF JAVA

The main objective of Java programming language creation was to make it portable, simple and secure programming language. Apart from this, there are also some awesome features which play important role in the popularity of this language. The features of Java are also known as java *buzzwords*.

A list of most important features of Java language are given below.

Simple

Java is very easy to learn and its syntax is simple, clean and easy to understand. According to Sun, Java language is a simple programming language because:

- Java syntax is based on C++ (so easier for programmers to learn it after C++).
- Java has removed many confusing and rarely-used features e.g. explicit pointers, operator overloading etc.
- There is no need to remove unreferenced objects because there is Automatic Garbage Collection in java.

Object-oriented

Java is object-oriented programming language. Everything in Java is an object. Object-oriented means we organize our software as a combination of different types of objects that incorporates both data and behaviour.

Object-oriented programming (OOPs) is a methodology that simplifies software development and maintenance by providing some rules.

Basic concepts of OOPs are:

1. Object
2. Class
3. Inheritance
4. Polymorphism
5. Abstraction
6. Encapsulation

Platform Independent

Java is platform independent because it is different from other languages like C, C++ etc. which are compiled into platform specific machines while Java is a write once, run anywhere language. A platform is the hardware or software environment in which a program runs.

There are two types of platforms software-based and hardware-based. Java provides software-based platform.
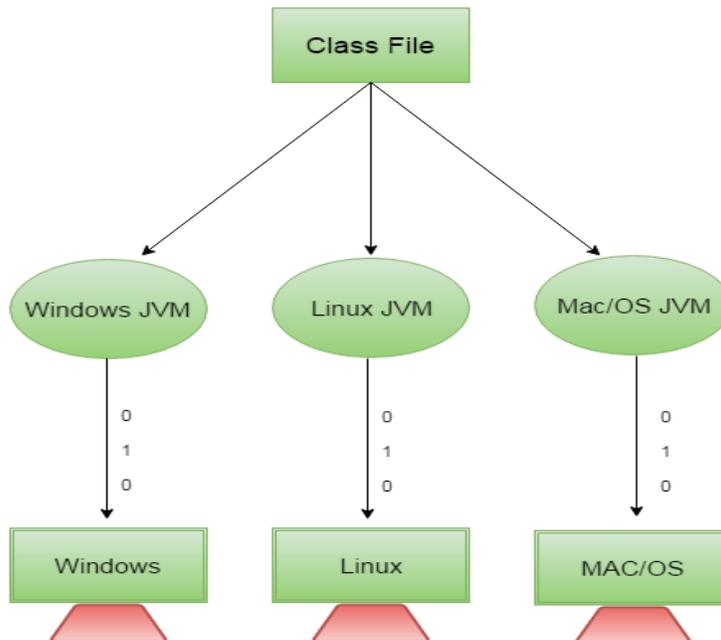
The Java platform differs from most other platforms in the sense that it is a software-based platform that runs on the top of other hardware-based platforms. It has two components:
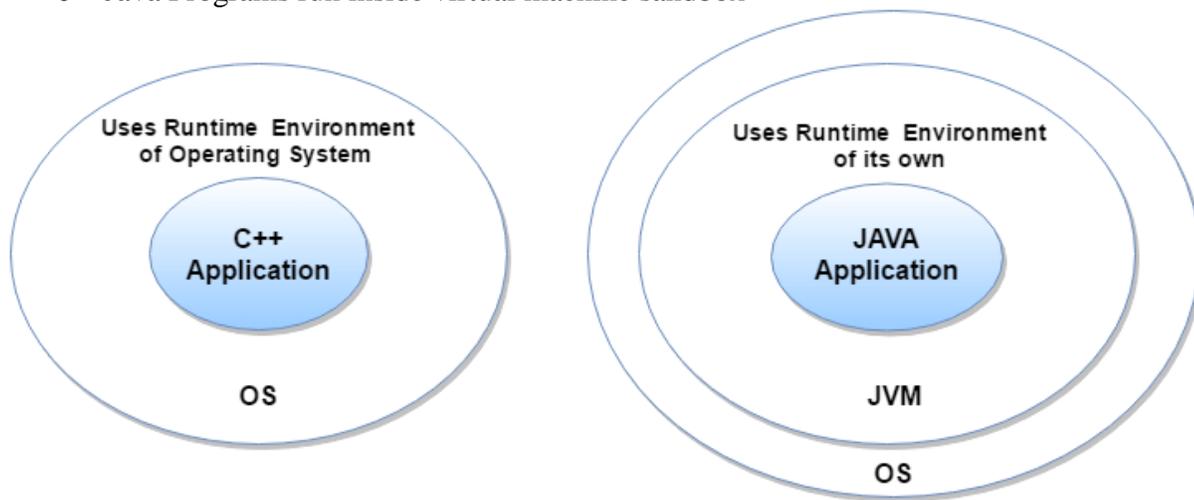
1. Runtime Environment
2. API(Application Programming Interface)

Java code can be run on multiple platforms e.g. Windows, Linux, Sun Solaris, Mac/OS etc. Java code is compiled by the compiler and converted into bytecode. This bytecode is a platform-independent code because it can be run on multiple platforms i.e. Write Once and Run Anywhere(WORA).

Secured

Java is best known for its security. With Java, we can develop virus-free systems. Java is secured because:

o   No explicit pointer
o   Java Programs run inside virtual machine sandbox

- **Classloader**: Classloader in Java is a part of the Java Runtime Environment(JRE) which is used to dynamically load Java classes into the Java Virtual Machine. It adds security by separating the package for the classes of the local file system from those that are imported from network sources.
- **Bytecode Verifier**: It checks the code fragments for illegal code that can violate access right to objects.
- **Security Manager**: It determines what resources a class can access such as reading and writing to the local disk.

These security are provided by java language. Some security can also be provided by application developer through SSL, JAAS, Cryptography etc.

Robust

- Robust simply means strong. Java is robust because:
- It uses strong memory management.
- There are lack of pointers that avoids security problems.
- There is automatic garbage collection in java which runs on the Java Virtual Machine to get rid of objects which are not being used by a Java application anymore.
- There is exception handling and type checking mechanism in java. All these points makes java robust.

Architecture-neutral

Java is architecture neutral because there is no implementation dependent features e.g. size of primitive types is fixed.

In C programming, int data type occupies 2 bytes of memory for 32-bit architecture and 4 bytes of memory for 64-bit architecture. But in java, it occupies 4 bytes of memory for both 32 and 64 bit architectures.

Portable

Java is portable because it facilitates you to carry the java bytecode to any platform. It doesn't require any type of implementation.

High-performance

Java is faster than other traditional interpreted programming languages because Java bytecode is "close" to native code. It is still a little bit slower than a compiled language (e.g. C++). Java is an interpreted language that is why it is slower than compiled languages e.g. C, C++ etc.

Distributed

Java is distributed because it facilitates users to create distributed applications in java. RMI and EJB are used for creating distributed applications. This feature of Java makes us able to access files by calling the methods from any machine on the internet.

Multi-threaded

A thread is like a separate program, executing concurrently. We can write Java programs that deal with many tasks at once by defining multiple threads. The main advantage of multi-threading is that it doesn't occupy memory for each thread. It shares a common memory area. Threads are important for multi-media, Web applications etc.

Dynamic

Java is a dynamic language. It supports dynamic loading of classes. It means classes are loaded on demand. It also supports functions from its native languages i.e. C and C++.

Java supports dynamic compilation and automatic memory management (garbage collection).

GARBAGE COLLECTION

Objects are dynamically allocated by using the **new** operator, dynamically allocated objects must be manually released by use of a **delete** operator. Java takes a different approach; it handles deallocation automatically this is called garbage collection. When no references to an object exist, that object is assumed to be no longer needed, and the memory occupied by the object can be reclaimed. Garbage collection only occurs sporadically (if at all) during the execution of your program. It will not occur simply because one or more objects exist that are no longer used.

THE JAVA ENVIRONMENT

JRE

JRE is an acronym for Java Runtime Environment. It is also written as Java RTE. The Java Runtime Environment is a set of software tools which are used for developing java applications. It is used to provide runtime environment. It is the implementation of JVM. It physically exists. It contains set of libraries + other files that JVM uses at runtime.

Implementation of JVMs are also actively released by other companies besides Sun Micro Systems.
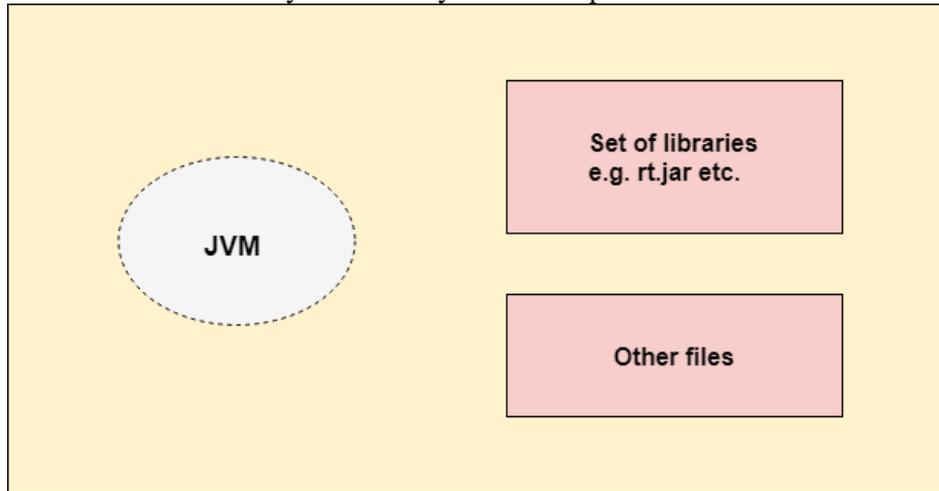

JRE

JDK

JDK is an acronym for Java Development Kit. The Java Development Kit (JDK) is a software development environment which is used to develop java applications and applets. It physically exists. It contains JRE + development tools.

JDK is an implementation of any one of the below given Java Platforms released by Oracle corporation:

- Standard Edition Java Platform
- Enterprise Edition Java Platform
- Micro Edition Java Platform

The JDK contains a private Java Virtual Machine (JVM) and a few other resources such as an interpreter/loader (Java), a compiler (javac), an archiver (jar), a documentation generator (Javadoc) etc. to complete the development of a Java Application.
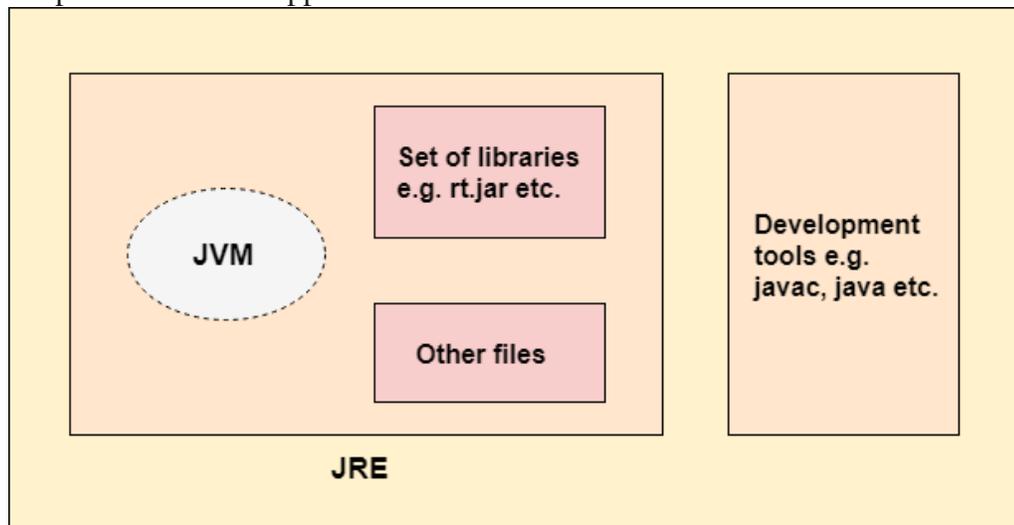
JDK

JVM (Java Virtual Machine)

JVM (Java Virtual Machine) is an abstract machine. It is a specification that provides runtime environment in which java bytecode can be executed.

JVMs are available for many hardware and software platforms (i.e. JVM is platform dependent).
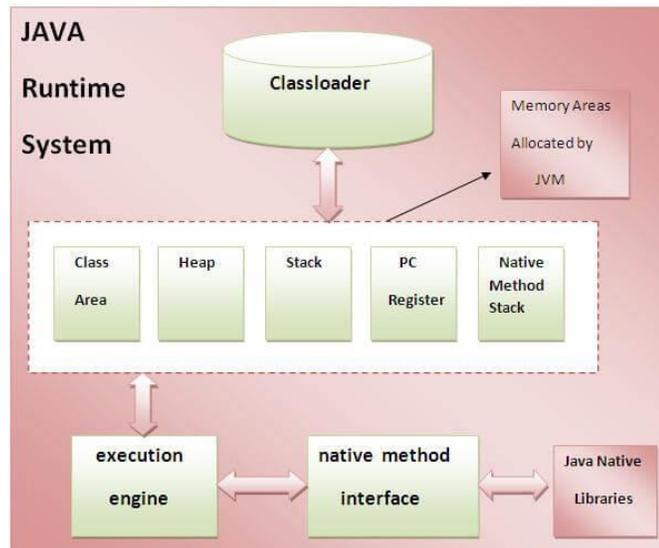
The JVM performs following operation:

- Loads code
- Verifies code
- Executes code
- Provides runtime environment

JVM provides definitions for the:

- Memory area
- Class file format
- Register set
- Garbage-collected heap
- Fatal error reporting etc.

Internal Architecture of JVM

1.Classloader

Classloader is a subsystem of JVM that is used to load class files.

2.Class(Method) Area

Class(Method) Area stores per-class structures such as the runtime constant pool, field and method data, the code for methods.

3.Heap

It is the runtime data area in which objects are allocated.

4.Stack

Java Stack stores frames. It holds local variables and partial results, and plays a part in method invocation and return.

Each thread has a private JVM stack, created at the same time as thread.

A new frame is created each time a method is invoked. A frame is destroyed when its method invocation completes.

5.Program Counter Register

PC (program counter) register contains the address of the Java virtual machine instruction currently being executed.

6. Native Method Stack

It contains all the native methods used in the application.

7. Execution Engine

Contains a virtual processor, Interpreter to read bytecode stream then execute the instructions and Just-In-Time(JIT) compiler is used to improve the performance. JIT compiles parts of the byte code that have similar functionality at the same time, and hence reduces the amount of time needed for compilation. Here, the term "compiler" refers to a translator from the instruction set of a Java virtual machine (JVM) to the instruction set of a specific CPU.

STRUCTURE OF JAVA PROGRAM

A first Simple Java Program

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

To compile:

javac Simple.java

To execute:

java Simple

class keyword is used to declare a class in java.

public keyword is an access modifier which represents visibility, it means it is visible to all.

static is a keyword, if we declare any method as static, it is known as static method. The core advantage of static method is that there is no need to create object to invoke the static method. The main method is executed by the JVM, so it doesn't require to create object to invoke the main method. So it saves memory.

void is the return type of the method, it means it doesn't return any value.

main represents the starting point of the program.

String[] args is used for command line argument.

System.out.println() is used print statement.

A program is written in JAVA, the javac compiles it. The result of the JAVA compiler is the .class file or the bytecode and not the machine native code (unlike C compiler).

The bytecode generated is a non-executable code and needs an interpreter to execute on a machine. This interpreter is the JVM and thus the Bytecode is executed by the JVM.

And finally program runs to give the desired output.
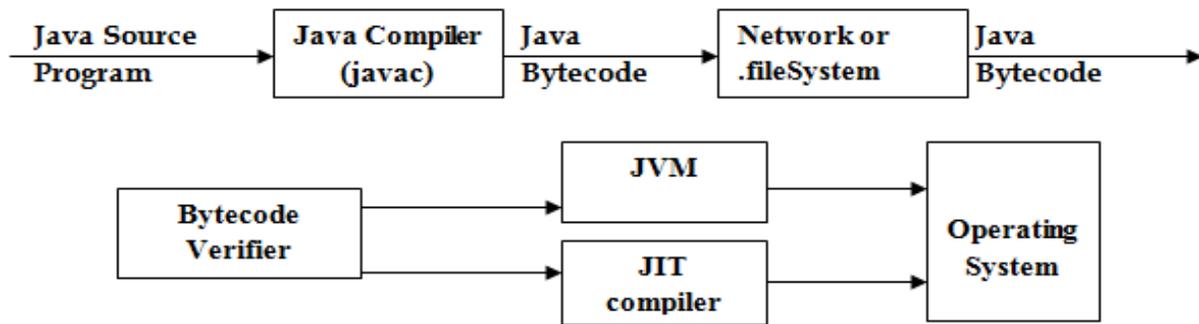
DEFINING CLASSES IN JAVA

The class is at the core of Java .A class is a *template* for an object, and an object is an *instance* of a class. A class is declared by use of the **class** keyword

Syntax:

class *classname* {
type instance-variable1;
type instance-variable2;
// ...
type instance-variableN;
type methodname1(parameter-list) {
// body of method
}
...
type methodnameN(parameter-list) {
// body of method
}

The data, or variables, defined within a **class** are called *instance variables*. The code is contained within *methods*. The methods and variables defined within a class are called *members* of the class. In most classes, the instance variables are acted upon and accessed by the methods defined for that class.

Variables defined within a class are called instance variables because each instance of the class (that is, each object of the class) contains its own copy of these variables. Thus, the data for one object is separate and unique from the data for another.

A Simple Class

class called **Box** that
defines three instance variables: **width**, **height**, and **depth**.

class Box {
double width;
double height;
double depth;
}

The new data type is called **Box**. This name is used to declare objects of type **Box**. The class declaration only creates a template. It does not create an actual object.

To create a Box object

Box mybox = new Box(); // create a Box object called mybox

mybox will be an instance of **Box**.

9

Each time you create an instance of a class, you are creating an object that contains its own copy of each instance variable defined by the class. To access these variables, you will use the *dot* (.) operator. The dot operator links the name of the object with the name of an instance variable.

Example1:

```
/* A program that uses the Box class.
Call this file BoxDemo.java
*/
class Box {
double width;
double height;
double depth;
}
// This class declares an object of type Box.
class BoxDemo {
public static void main(String args[]) {
Box mybox = new Box();
double vol;
// assign values to mybox's instance variables
mybox.width = 10;
mybox.height = 20;
mybox.depth = 15;
// compute volume of box
vol = mybox.width * mybox.height * mybox.depth;
System.out.println("Volume is " + vol);
}
}
```

Output:

Volume is 3000.0

Example2:

```
// This program declares two Box objects.
class Box {
double width;
double height;
double depth;
}
class BoxDemo2 {
public static void main(String args[]) {
Box mybox1 = new Box();
Box mybox2 = new Box();
double vol;
// assign values to mybox1's instance variables
mybox1.width = 10;
mybox1.height = 20;
mybox1.depth = 15;
/* assign different values to mybox2's
instance variables */
mybox2.width = 3;
mybox2.height = 6;
```

```
mybox2.depth = 9;
// compute volume of first box
vol = mybox1.width * mybox1.height * mybox1.depth;
System.out.println("Volume is " + vol);
// compute volume of second box
vol = mybox2.width * mybox2.height * mybox2.depth;
System.out.println("Volume is " + vol);
}
}
```

Output:
Volume is 3000.0
Volume is 162.0

Declaring Objects
First, declare a variable of the class type. This variable does not define an object.Instead, it is simply a variable that can *refer* to an object.
Second, you must acquire an actual, physical copy of the object and assign it to that variable. This is done using the **new** operator. The **new** operator dynamically allocates (that is, allocates at run time) memory for an object and returns a reference to it. This reference is then stored in the variable. Thus, in Java, all class objects must be dynamically allocated.
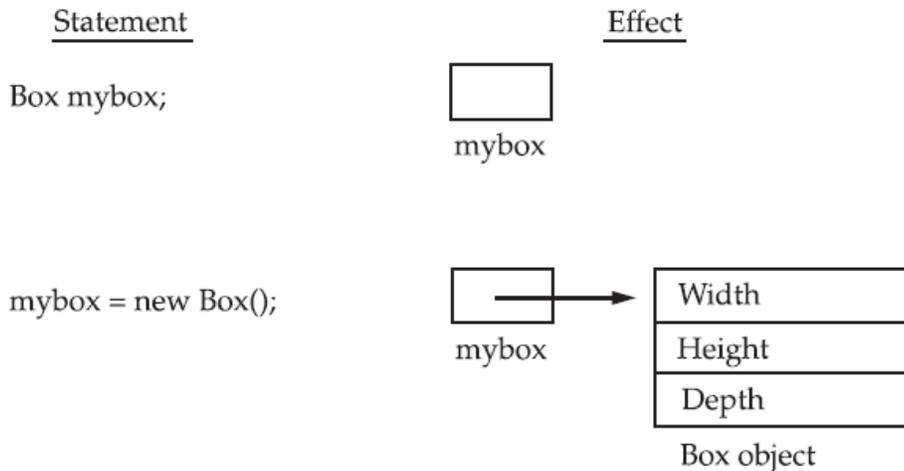
Syntax:
```
Box mybox = new Box();
Box mybox; // declare reference to object
mybox = new Box(); // allocate a Box object
```

The first line declares **mybox** as a reference to an object of type **Box**. At this point, **mybox** does not yet refer to an actual object. The next line allocates an object and assigns a reference to it to **mybox**. After the second line executes, we can use **mybox** as if it were a **Box** object. But in reality, **mybox** simply holds, in essence, the memory address of the actual **Box** object.

Assigning Object Reference Variables
**Syntax:**
```
Box b1 = new Box();
Box b2 = b1;
```
b2 is being assigned a reference to a copy of the object referred to by **b1**. **b1** and **b2** will both refer to the *same* object. The assignment of **b1** to **b2** did not allocate any memory or copy any part of the original

object. It simply makes **b2** refer to the same object as does **b1**. Thus, any changes made to the object through **b2** will affect the object to which **b1** is referring, since they are the same object.
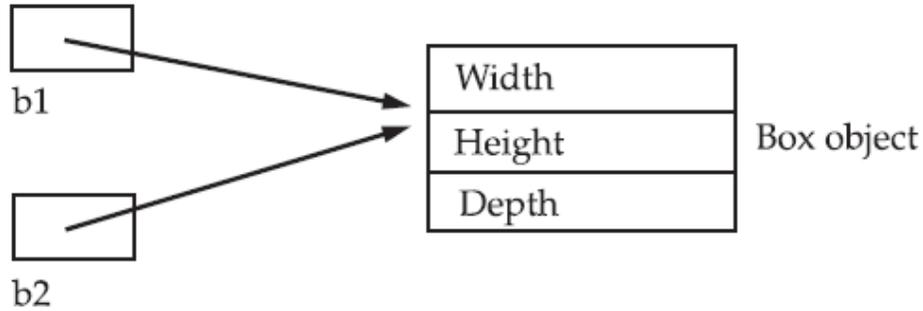
CONSTRUCTORS

- Constructors are special member functions whose task is to initialize the objects of its class.
- It is a special member function, it has the same as the class name.
- Java constructors are invoked when their objects are created. It is named such because, it constructs the value, that is provides data for the object and are used to initialize objects.
- Every class has a constructor when we don't explicitly declare a constructor for any java class the compiler creates a default constructor for that class which does not have any return type.
- The constructor in Java cannot be abstract, static, final or synchronized and these modifiers are not allowed for the constructor.

There are two types of constructors:
1. Default constructor (no-arg constructor)
2. Parameterized constructor

Default constructor (no-arg constructor)
A constructor having no parameter is known as default constructor and no-arg constructor.
Example:
```
/* Here, Box uses a constructor to initialize the
dimensions of a box.
*/
class Box {
double width;
double height;
double depth;
// This is the constructor for Box.
Box() {
System.out.println("Constructing Box");
width = 10;
height = 10;
depth = 10;
}
// compute and return volume
double volume() {
return width * height * depth;
}
}
```

```java
class BoxDemo6 {
public static void main(String args[]) {
// declare, allocate, and initialize Box objects
Box mybox1 = new Box();
Box mybox2 = new Box();
double vol;
// get volume of first box
vol = mybox1.volume();
System.out.println("Volume is " + vol);
// get volume of second box
vol = mybox2.volume();
System.out.println("Volume is " + vol);
}
}
```

Output:
Constructing Box
Constructing Box
Volume is 1000.0
Volume is 1000.0
new Box( ) is calling the **Box( )** constructor. When the constructor for a class is not explicitly defined , then Java creates a default constructor for the class. The default constructor automatically initializes all instance variables to their default values, which are zero, **null**, and **false**, for numeric types, reference types, and **boolean**, respectively.

Parameterized Constructors
A constructor which has a specific number of parameters is called parameterized constructor. Parameterized constructor is used to provide different values to the distinct objects.
Example:
```java
/* Here, Box uses a parameterized constructor to
initialize the dimensions of a box.
*/
class Box {
double width;
double height;
double depth;
// This is the constructor for Box.
Box(double w, double h, double d) {
width = w;
height = h;
depth = d;
}
// compute and return volume
double volume() {
return width * height * depth;
}
}
class BoxDemo7 {
public static void main(String args[]) {
```

```java
// declare, allocate, and initialize Box objects
Box mybox1 = new Box(10, 20, 15);
Box mybox2 = new Box(3, 6, 9);
double vol;
// get volume of first box
vol = mybox1.volume();
System.out.println("Volume is " + vol);
// get volume of second box
vol = mybox2.volume();
System.out.println("Volume is " + vol);
}
}
```

Output:
Volume is 3000.0
Volume is 162.0
Box mybox1 = new Box(10, 20, 15);
The values 10, 20, and 15 are passed to the **Box( )** constructor when **new** creates the object. Thus, **mybox1**'s copy of **width**, **height**, and **depth** will contain the values 10, 20, and 15 respectively.

Overloading Constructors
Example:
```java
/* Here, Box defines three constructors to initialize
the dimensions of a box various ways.
*/
class Box {
double width;
double height;
double depth;
// constructor used when all dimensions specified
Box(double w, double h, double d) {
width = w;
height = h;
depth = d;
}
// constructor used when no dimensions specified
Box() {
width = -1; // use -1 to indicate
height = -1; // an uninitialized
depth = -1; // box
}
// constructor used when cube is created
Box(double len) {
width = height = depth = len;
}
// compute and return volume
double volume() {
return width * height * depth;
}
```

14

```
}
class OverloadCons
{
public static void main(String args[])
{
// create boxes using the various constructors
Box mybox1 = new Box(10, 20, 15);
Box mybox2 = new Box();
Box mycube = new Box(7);
double vol;
// get volume of first box
vol = mybox1.volume();
System.out.println("Volume of mybox1 is " + vol);
// get volume of second box
vol = mybox2.volume();
System.out.println("Volume of mybox2 is " + vol);
// get volume of cube
vol = mycube.volume();
System.out.println("Volume of mycube is " + vol);
}
}
```

Output:
Volume of mybox1 is 3000.0
Volume of mybox2 is -1.0
Volume of mycube is 343.0

METHODS
Syntax:
type name(*parameter-list*) {
// body of method
}

- *type* specifies the type of data returned by the method. This can be any valid type, including class types that you create.
- If the method does not return a value, its return type must be **void**.
- The name of the method is specified by *name*.
- The *parameter-list* is a sequence of type and identifier pairs separated by commas. Parameters are essentially variables that receive the value of the arguments passed to the method when it is called. If the method has no parameters, then the parameter list will be empty.
- Methods that have a return type other than **void** return a value to the calling routine using the following form of the **return** statement:

Syntax:
return *value*;
Example:
```
// This program includes a method inside the box class.
class Box {
double width;
double height;
double depth;
// display volume of a box
```

15

```
void volume() {
System.out.print("Volume is ");
System.out.println(width * height * depth);
}}
class BoxDemo3 {
public static void main(String args[]) {
Box mybox1 = new Box();
Box mybox2 = new Box();
// assign values to mybox1's instance variables
mybox1.width = 10;
mybox1.height = 20;
mybox1.depth = 15;
/* assign different values to mybox2's
instance variables */
mybox2.width = 3;
mybox2.height = 6;
mybox2.depth = 9;
// display volume of first box
mybox1.volume();
// display volume of second box
mybox2.volume();
}
}
```
Output:
Volume is 3000.0
Volume is 162.0
The first line here invokes the **volume( )** method on **mybox1**. That is, it calls **volume( )** relative to the **mybox1** object, using the object's name followed by the dot operator. Thus, the call to **mybox1.volume( )** displays the volume of the box defined by **mybox1**, and the call to **mybox2.volume( )** displays the volume of the box defined by **mybox2**. Each time **volume( )** is invoked, it displays the volume for the specified box.

Returning a Value
Example:
```
// Now, volume() returns the volume of a box.
class Box {
double width;
double height;
double depth;
// compute and return volume
double volume() {
return width * height * depth;
}
}
class BoxDemo4 {
public static void main(String args[]) {
Box mybox1 = new Box();
Box mybox2 = new Box();
double vol;
// assign values to mybox1's instance variables
```

```java
mybox1.width = 10;
mybox1.height = 20;
mybox1.depth = 15;
/* assign different values to mybox2's
instance variables */
mybox2.width = 3;
mybox2.height = 6;
mybox2.depth = 9;
// get volume of first box
vol = mybox1.volume();
System.out.println("Volume is " + vol);
// get volume of second box
vol = mybox2.volume();
System.out.println("Volume is " + vol);
}
}
```
Output:
Volume is 3000
Volume is 162

when **volume( )** is called, it is put on the right side of an assignment statement. On the left is a variable, in this case **vol**, that will receive the value returned by **volume( )**.
Syntax:
```java
vol = mybox1.volume();
```
executes, the value of **mybox1.volume( )** is 3,000 and this value then is stored in **vol**.

There are two important things to understand about returning values:
• The type of data returned by a method must be compatible with the return type specified by the method.
• The variable receiving the value returned by a method (such as **vol**, in this case) must also be compatible with the return type specified for the method.

Adding a Method That Takes Parameters
Example:
```java
// This program uses a parameterized method.
class Box {
double width;
double height;
double depth;
// compute and return volume
double volume() {
return width * height * depth;
}
// sets dimensions of box
void setDim(double w, double h, double d) {
width = w;
height = h;
depth = d;
}
}
```

```
class BoxDemo5 {
public static void main(String args[]) {
Box mybox1 = new Box();
Box mybox2 = new Box();
double vol;
// initialize each box
mybox1.setDim(10, 20, 15);
mybox2.setDim(3, 6, 9);
// get volume of first box
vol = mybox1.volume();
System.out.println("Volume is " + vol);
// get volume of second box
vol = mybox2.volume();
System.out.println("Volume is " + vol);
}
}
```

Output:

Volume is 3000
Volume is 162

The this Keyword

this keyword is used to to refer to the object that invoked it. this can be used inside any method to refer to the *current* object. That is, this is always a reference to the object on which the method was invoked. this() can be used to invoke current class constructor.

Syntax:

```
Box(double w, double h, double d) {
this.width = w;
this.height = h;
this.depth = d;
}
```

Example:

```
class Student
{
 int id;
 String name;
 student(int id, String name)
 {
  this.id = id;
  this.name = name;

}
void display()
{
        System.out.println(id+" "+name);
 }
 public static void main(String args[])
{
  Student stud1 = new Student(01,"Tarun");
  Student stud2 = new Student(02,"Barun");
```

```java
   stud1.display();
   stud2.display();
  }
}
```

Output:
01 Tarun
02 Barun

Overloading Methods
When two or more methods within the same class that have the same name, but their parameter declarations are different. The methods are said to be overloaded, and the process is referred to as method overloading. Method overloading is one of the ways that Java supports polymorphism.
There are two ways to overload the method in java
1. By changing number of arguments
2. By changing the data type

Example:
```java
// Demonstrate method overloading.
class OverloadDemo {
void test() {
System.out.println("No parameters");
}
// Overload test for one integer parameter.
void test(int a) {
System.out.println("a: " + a);
}
// Overload test for two integer parameters.
void test(int a, int b) {
System.out.println("a and b: " + a + " " + b);
}
// Overload test for a double parameter
double test(double a) {
System.out.println("double a: " + a);
return a*a;
}
}
class Overload {
public static void main(String args[]) {
OverloadDemo ob = new OverloadDemo();
double result;
// call all versions of test()
ob.test();
ob.test(10);
ob.test(10, 20);
result = ob.test(123.25);
System.out.println("Result of ob.test(123.25): " + result);
}
}
```

Output:
No parameters

a: 10
a and b: 10 20
double a: 123.25
Result of ob.test(123.25): 15190.5625

Method Overriding
When a method in a subclass has the same name and type signature as a method in its superclass, then the method in the subclass is said to override the method in the superclass. When an overridden method is called from within its subclass, it will always refer to the version of that method defined by the subclass. The version of the method defined by the superclass will be hidden.
Example:
```
// Method overriding.
class A {
int i, j;
A(int a, int b) {
i = a;
j = b;
}
// display i and j
void show() {
System.out.println("i and j: " + i + " " + j);
}
}
class B extends A {
int k;
B(int a, int b, int c) {
super(a, b);
k = c;
}
// display k – this overrides show() in A
void show() {
System.out.println("k: " + k);
}
}
class Override {
public static void main(String args[]) {
B subOb = new B(1, 2, 3);
subOb.show(); // this calls show() in B
}
}
```
Output:
k: 3

When **show( )** is invoked on an object of type **B**, the version of **show( )** defined within **B** is used. That is, the version of **show( )** inside **B** overrides the version declared in **A**. If you wish to access the superclass version of an overridden method, you can do so by using **super**. For example, in this version of **B**, the superclass version of **show( )** is invoked within the subclass' version. This allows all instance variables to be displayed.
```
class B extends A {
int k;
B(int a, int b, int c) {
```

```
super(a, b);
k = c;
}
void show() {
super.show(); // this calls A's show()
System.out.println("k: " + k);
}
}
```

If you substitute this version of **A** into the previous program, you will see the following

Output:
i and j: 1 2
k: 3
Here, **super.show( )** calls the superclass version of **show( )**.

ACCESS PROTECTION

The access modifiers in java specifies accessibility (scope) of a data member, method, constructor or class.
There are 4 types of java access modifiers:

1. private

2. default

3. protected

4. public

1) Private Access Modifier
The private access modifier is accessible only within class.
Simple example of private access modifier
In this example, we have created two classes A and Simple. A class contains private data member and private method. We are accessing these private members from outside the class, so there is compile time error.

```
class A{
private int data=40;
private void msg(){System.out.println("Hello java");}
}
public class Simple{
 public static void main(String args[]){
   A obj=new A();
   System.out.println(obj.data);//Compile Time Error
   obj.msg();//Compile Time Error
   }
}
```

Role of Private Constructor
If you make any class constructor private, you cannot create the instance of that class from outside the class.
For example:

```
class A{
private A(){}//private constructor
void msg(){System.out.println("Hello java");}
}
public class Simple{
 public static void main(String args[]){
```

```
  A obj=new A();//Compile Time Error
 }
}
```

If you make any class constructor private, you cannot create the instance of that class from outside the class. For example:

```
class A{
private A(){}//private constructor
void msg(){System.out.println("Hello java");}
}
public class Simple{
 public static void main(String args[]){
  A obj=new A();//Compile Time Error
 }
}
```

2) Default Access Modifier
If you don't use any modifier, it is treated as default bydefault. The default modifier is accessible only within package.

Example:
In this example, we have created two packages pack and mypack. We are accessing the A class from outside its package, since A class is not public, so it cannot be accessed from outside the package.

```
//save by A.java
package pack;
class A{
  void msg(){System.out.println("Hello");}
}
//save by B.java
package mypack;
import pack.*;
class B{
 public static void main(String args[]){
  A obj = new A();//Compile Time Error
  obj.msg();//Compile Time Error
 }
}
```

In the above example, the scope of class A and its method msg() is default so it cannot be accessed from outside the package.

3) Protected Access Modifier
The protected access modifier is accessible within package and outside the package but through inheritance only.

The protected access modifier can be applied on the data member, method and constructor. It can't be applied on the class.

Example:

In this example, we have created the two packages pack and mypack. The A class of pack package is public, so can be accessed from outside the package. But msg method of this package is declared as protected, so it can be accessed from outside the class only through inheritance.

```
//save by A.java
package pack;
public class A{
protected void msg(){System.out.println("Hello");}
}
//save by B.java
package mypack;
import pack.*;

class B extends A{
  public static void main(String args[]){
  B obj = new B();
  obj.msg();
  }
}
```

Output:

Hello

4) Public Access Modifier

The public access modifier is accessible everywhere. It has the widest scope among all other modifiers.

Example:

```
//save by A.java
package pack;
public class A{
public void msg(){System.out.println("Hello");}
}
//save by B.java
package mypack;
import pack.*;
class B{
  public static void main(String args[]){
  A obj = new A();
  obj.msg();
  }
}
```

Output:

Hello

Access Modifier	**Within Class**	**Within Package**	**Outside Package**	**Outside Package**

23

			By Subclass Only	
Private	Y	N	N	N
Default	Y	Y	N	N
Protected	Y	Y	Y	N
Public	Y	Y	Y	Y

Java access modifiers with method overriding
If you are overriding any method, overridden method (i.e. declared in subclass) must not be more restrictive.
```
class A{
protected void msg(){System.out.println("Hello java");}
}
public class Simple extends A{
void msg(){System.out.println("Hello java");}//C.T.Error
 public static void main(String args[]){
  Simple obj=new Simple();
  obj.msg();
  }
}
```
The default modifier is more restrictive than protected. That is why there is compile time error.

STATIC MEMBERS
Static is a non-access modifier in Java which is applicable for the following:
1. blocks
2. variables
3. methods
4. nested classes

Static blocks
If you need to do computation in order to initialize your **static variables**, you can declare a static block that gets executed exactly once, when the class is first loaded.

Example:
```
// Java program to demonstrate use of static blocks
class Test
{
   // static variable
   static int a = 10;
   static int b;

   // static block
   static {
      System.out.println("Static block initialized.");
      b = a * 4;
   }

   public static void main(String[] args)
   {
     System.out.println("from main");
     System.out.println("Value of a : "+a);
     System.out.println("Value of b : "+b);
```

```
    }
}
```
Output:
Static block initialized.
from main
Value of a : 10
Value of b : 40

Static variables

When a variable is declared as static, then a single copy of variable is created and shared among all objects at class level. Static variables are, essentially, global variables. All instances of the class share the same static variable.

Important points for static variables :-
- We can create static variables at class-level only.
- static block and static variables are executed in order they are present in a program.

Example:
```
 // Demonstrate static variables, methods, and blocks.
class UseStatic {
static int a = 3;
static int b;
static void meth(int x) {
System.out.println("x = " + x);
System.out.println("a = " + a);
System.out.println("b = " + b);
}
static {
System.out.println("Static block initialized.");
b = a * 4;
}
public static void main(String args[]) {
meth(42);
}
}
```
Output:
Static block initialized.
x = 42
a = 3
b = 12

Static methods

When a method is declared with *static* keyword, it is known as static method. When a member is declared **static**, it can be accessed before any objects of its class are created, and without reference to any object. The most common example of a static method is *main( )* method. Methods declared as static have several restrictions:
- They can only directly call other static methods.
- They can only directly access static data.
- They cannot refer to **this** or **super** in any way.

Syntax:
classname.method( )

Example:
//Inside main( ), the static method callme( ) and the static variable b are accessed through their class name
//StaticDemo.

```java
class StaticDemo {
static int a = 42;
static int b = 99;
static void callme() {
System.out.println("a = " + a);
}
}
class StaticByName {
public static void main(String args[]) {
StaticDemo.callme();
System.out.println("b = " + StaticDemo.b);
}
}
```

Output:
a = 42
b = 99

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.
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.
Syntax:
//This is single line comment
Example:

```java
public class CommentExample1
{
 public static void main(String[] args)
 {
   int i=10;//Here, i is a variable
   System.out.println(i);
 }
}
```

Output:
10

2) Java Multi Line Comment
The multi line comment is used to comment multiple lines of code.
Syntax:
/*

This
is
multi line
comment
*/
Example:
```
public class CommentExample2
{
  public static void main(String[] args)
  {
   /* Let's declare and
    print variable in java. */
    int i=10;
    System.out.println(i);
  }
}
```
Output:
10

3) Java Documentation Comment
The documentation comment is used to create documentation API. To create documentation API, you need to use **javadoc tool.**
Syntax:
```
/**
This
is
documentation
comment
*/
```

Example:
```
/** The Calculator class provides methods to get addition and subtraction of given 2 numbers.*/
public class Calculator
{
/** The add() method returns addition of given numbers.*/
       public static int add(int a, int b)
       {
       return a+b;
       }
/** The sub() method returns subtraction of given numbers.*/
       public static int sub(int a, int b)
       {
       return a-b;
       }
}
```
 This type of comment is used to produce an HTML file that documents your program. The documentation comment begins with a /** and ends with a */.

DATATYPES IN JAVA

Data types specify the different sizes and values that can be stored in the variable. There are two types of data types in Java:

1. **Primitive data types:** The primitive data types include Integer, Character, Boolean, and Floating Point.
2. **Non-primitive data types:** The non-primitive data types include Classes, Interfaces, and Arrays.
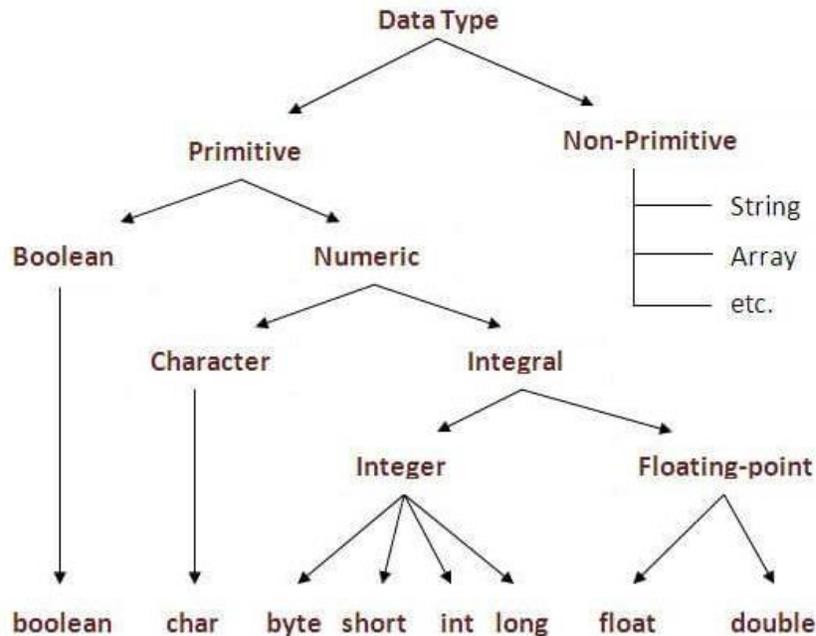
Java defines eight primitive types of data: byte, short, int, long, char, float, double, and boolean. These can be put in four groups:

• Integers This group includes byte, short, int, and long, which are for whole-valued signed numbers.

• Floating-point numbers This group includes float and double, which represent numbers with fractional precision.

• Characters This group includes char, which represents symbols in a character set, like letters and numbers.

• Boolean This group includes boolean, which is a special type for representing true/false values.

Example :

```
// Compute distance light travels using long variables.
class Light {
 public static void main(String args[]) {
 int lightspeed;
 long days;
 long seconds;
 long distance;
 // approximate speed of light in miles per second
 lightspeed = 186000;
 days = 1000; // specify number of days here
 seconds = days * 24 * 60 * 60; // convert to seconds
 distance = lightspeed * seconds; // compute distance
 System.out.print("In " + days);
 System.out.print(" days light will travel about ");
 System.out.println(distance + " miles.");
 }
}
```

Output:
In 1000 days light will travel about 16070400000000 miles.
Clearly, the result could not have been held in an int variable.

VARIABLES

A variable is a container which holds the value and that can be changed durig the execution of the program. A variable is assigned with a datatype. Variable is a name of memory location. All the variables must be declared before they can be used. There are three types of variables in java: local variable, instance variable and static variable.
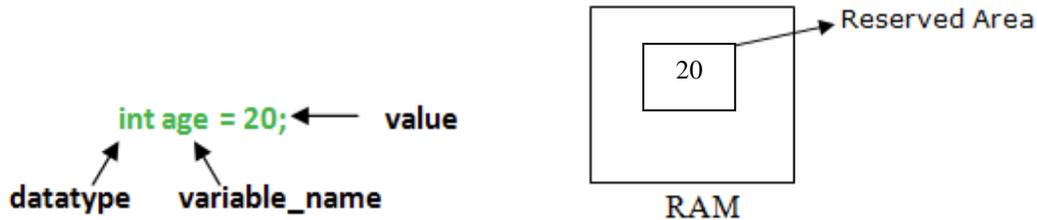
1) Local Variable
A variable defined within a block or method or constructor is called local variable.
- These variable are created when the block in entered or the function is called and destroyed after exiting from the block or when the call returns from the function.
- The scope of these variables exists only within the block in which the variable is declared. i.e. we can access these variable only within that block.

Example:
```
import java.io.*;
public class StudentDetails
{
   public void StudentAge()
   {   //local variable age
      int age = 0;
      age = age + 5;
      System.out.println("Student age is : " + age);
   }
   public static void main(String args[])
   {
      StudentDetails obj = new StudentDetails();
      obj.StudentAge();
   }
}
```
Output:
Student age is : 5

2) Instance Variable
Instance variables are non-static variables and are declared in a class outside any method, constructor or block.
- As instance variables are declared in a class, these variables are created when an object of the class is created and destroyed when the object is destroyed.

- Unlike local variables, we may use access specifiers for instance variables. If we do not specify any access specifier then the default access specifier will be used.

Example:
```
import java.io.*;
class Marks{
    int m1;
    int m2;
}
class MarksDemo
{
    public static void main(String args[])
    {   //first object
        Marks obj1 = new Marks();
        obj1.m1 = 50;
        obj1.m2 = 80;
        //second object
        Marks obj2 = new Marks();
        obj2.m1 = 80;
        obj2.m2 = 60;
        //displaying marks for first object
        System.out.println("Marks for first object:");
        System.out.println(obj1.m1);
        System.out.println(obj1.m2);
        //displaying marks for second object
        System.out.println("Marks for second object:");
        System.out.println(obj2.m1);
        System.out.println(obj2.m2);
    }}
```
Output:
Marks for first object:
50
80
Marks for second object:
80
60

3) Static variable

Static variables are also known as Class variables.
- These variables are declared similarly as instance variables, the difference is that static variables are declared using the static keyword within a class outside any method constructor or block.
- Unlike instance variables, we can only have one copy of a static variable per class irrespective of how many objects we create.
- Static variables are created at start of program execution and destroyed automatically when execution ends.

Example:
```
import java.io.*;
class Emp {

    // static variable salary
    public static double salary;
```

```
   public static String name = "Vijaya";
}
public class EmpDemo
{
   public static void main(String args[]) {

      //accessing static variable without object
      Emp.salary = 1000;
      System.out.println(Emp.name + "'s average salary:" + Emp.salary);
   }
}
```
Output:
Vijaya's average salary:10000.0

Difference between Instance variable and Static variable

INSTANCE VARIABLE	STATIC VARIABLE
Each object will have its **own copy** of instance variable	We can only have **one copy** of a static variable per class irrespective of how many objects we create.
Changes made in an instance variable using one object will **not be reflected** in other objects as each object has its own copy of instance variable	In case of static changes **will be reflected** in other objects as static variables are common to all object of a class.
We can access instance variables **through object references**	Static Variables can be accessed **directly using class name.**

Class Sample
{
int a;
}
	Class Sample
	{
	static int a;
	}

OPERATORS IN JAVA

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

- Arithmetic Operators
- Increment and Decrement
- Bitwise Operators
- Relational Operators
- Boolean Operators
- Assignment Operator
- Ternary Operator

Arithmetic Operators

Arithmetic operators are used to manipulate mathematical expressions

Operator Result

Operator	Result
+	Addition (also unary plus)
−	Subtraction (also unary minus)
*	Multiplication
/	Division
%	Modulus
++	Increment
+=	Addition assignment
− =	Subtraction assignment
*=	Multiplication assignment
/=	Division assignment
%=	Modulus assignment
− −	Decrement

Example:

```java
// Demonstrate the basic arithmetic operators.
class BasicMath
{
public static void main(String args[])
{
// arithmetic using integers
System.out.println("Integer Arithmetic");
int a = 1 + 1;
int b = a * 3;
int c = b / 4;
int d = c - a;
int e = -d;
System.out.println("a = " + a);
System.out.println("b = " + b);
System.out.println("c = " + c);
System.out.println("d = " + d);
System.out.println("e = " + e);
// arithmetic using doubles
System.out.println("\nFloating Point Arithmetic");
double da = 1 + 1;
double db = da * 3;
double dc = db / 4;
double dd = dc - a;
double de = -dd;
System.out.println("da = " + da);
System.out.println("db = " + db);
System.out.println("dc = " + dc);
System.out.println("dd = " + dd);
System.out.println("de = " + de);
}}
```

Output:

Integer Arithmetic
a = 2
b = 6
c = 1

d = -1
e = 1
Floating Point Arithmetic
da = 2.0
db = 6
dc = 1.5
dd = -0.5
de = 0.5

Modulus Operator

The modulus operator, **%**, returns the remainder of a division operation. It can be applied to floating-point types as well as integer types.

Example:

```
// Demonstrate the % operator.
class Modulus {
public static void main(String args[]) {
int x = 42;
double y = 42.25;
System.out.println("x mod 10 = " + x % 10);
System.out.println("y mod 10 = " + y % 10);
}
}
```

Output:
x mod 10 = 2
y mod 10 = 2.25

Arithmetic Compound Assignment Operators

Java provides special operators that can be used to combine an arithmetic operation with an assignment.

a = a + 4;

In Java, you can rewrite this statement as shown here:

a += 4;

Syntax:

var op= expression;

Example:

```
// Demonstrate several assignment operators.
class OpEquals
{
public static void main(String args[])
{
int a = 1;
int b = 2;
int c = 3;
a += 5;
b *= 4;
c += a * b;
```

```
c %= 6;
System.out.println("a = " + a);
System.out.println("b = " + b);
System.out.println("c = " + c);
}
}
```
Output:
a = 6
b = 8
c = 3

Increment and Decrement Operators

The ++ and the − − are Java's increment and decrement operators. The increment operator increases its operand by one. The decrement operator decreases its operand by one.

Example:
```
// Demonstrate ++.
class IncDec
{
public static void main(String args[])
{
int a = 1;
int b = 2;
int c;
int d;
c = ++b;
d = a++;
c++;
System.out.println("a = " + a);
System.out.println("b = " + b);
System.out.println("c = " + c);
System.out.println("d = " + d);
}
}
```
Output:
a = 2
b = 3
c = 4
d = 1

Bitwise Operators

Java defines several *bitwise operators* that can be applied to the integer types: **long**, **int**, **short**, **char**, and **byte**. These operators act upon the individual bits of their operands.

Operator	Result
~	Bitwise unary NOT
&	Bitwise AND
|	Bitwise OR
^	Bitwise exclusive OR
>>	Shift right
>>>	Shift right zero fill
<<	Shift left
&=	Bitwise AND assignment
|=	Bitwise OR assignment
^=	Bitwise exclusive OR assignment
>>=	Shift right assignment
>>>=	Shift right zero fill assignment
<<=	Shift left assignment

Bitwise Logical Operators

The bitwise logical operators are **&**, **|**, **^**, and **~**. the bitwise operators are applied to each individual bit within each operand.

A	B	A | B	A & B	A ^ B	~A
0	0	0	0	0	1
1	0	1	0	1	0
0	1	1	0	1	1
1	1	1	1	0	0

Example:

```
// Demonstrate the bitwise logical operators.
class BitLogic
{
public static void main(String args[])
{
String binary[] = {"0000", "0001", "0010", "0011", "0100", "0101", "0110", "0111",
"1000", "1001", "1010", "1011", "1100", "1101", "1110", "1111"};
int a = 3; // 0 + 2 + 1 or 0011 in binary
int b = 6; // 4 + 2 + 0 or 0110 in binary
int c = a | b;
int d = a & b;
int e = a ^ b;
int f = (~a & b)|(a & ~b);
int g = ~a & 0x0f;
System.out.println(" a = " + binary[a]);
System.out.println(" b = " + binary[b]);
System.out.println(" a|b = " + binary[c]);
System.out.println(" a&b = " + binary[d]);
System.out.println(" a^b = " + binary[e]);
```

```
System.out.println("~a&b|a&~b = " + binary[f]);
System.out.println(" ~a = " + binary[g]);
}
}
```
Output:
a = 0011
b = 0110
a|b = 0111
a&b = 0010
a^b = 0101
~a&b|a&~b = 0101
~a = 1100

Left Shift Operator
The Java left shift operator << is used to shift all of the bits in a value to the left side of a specified number of times.
Example:
```
class OperatorExample
{
public static void main(String args[])
{
System.out.println(10<<2);//10*2^2=10*4=40
System.out.println(10<<3);//10*2^3=10*8=80
System.out.println(20<<2);//20*2^2=20*4=80
System.out.println(15<<4);//15*2^4=15*16=240
}
}
```
Output:
40
80
80
240

Right Shift Operator
The Java right shift operator >> is used to move left operands value to right by the number of bits specified by the right operand.
Example:
```
class OperatorExample
{
public static void main(String args[])
{
System.out.println(10>>2);//10/2^2=10/4=2
System.out.println(20>>2);//20/2^2=20/4=5
System.out.println(20>>3);//20/2^3=20/8=2
}
}
```
Output:
2
5

2

Relational Operators

The *relational operators* determine the relationship that one operand has to the other.Specifically, they determine equality and ordering. The outcome of these operations is a **boolean** value.

Boolean Operators

The Boolean logical operators shown here operate only on **boolean** operands. All of the binary logical operators combine two **boolean** values to form a resultant **boolean** value.

Operator	Result
&	Logical AND
|	Logical OR
^	Logical XOR (exclusive OR)
||	Short-circuit OR
&&	Short-circuit AND
!	Logical unary NOT
&=	AND assignment
|=	OR assignment
^=	XOR assignment
==	Equal to
!=	Not equal to
?:	Ternary if-then-else

Example:

```
// Demonstrate the boolean logical operators.
class BoolLogic
{
public static void main(String args[])
{
boolean a = true;
boolean b = false;
boolean c = a | b;
boolean d= a & b;
boolean e = a ^ b;
boolean f = (!a & b) | (a & !b);
boolean g = !a;
System.out.println(" a = " + a);
System.out.println(" b = " + b);
System.out.println(" a|b = " + c);
System.out.println(" a&b = " + d);
System.out.println(" a^b = " + e);
System.out.println("!a&b|a&!b = " + f);
System.out.println(" !a = " + g);
}
}
```

Output:

```
a = true
b = false
a|b = true
a&b = false
```

a^b = true
!a&b|a&!b = true
!a=false
In the output, the string representation of a Java **boolean** value is one of the literal values **true** or **false**. Java AND Operator Example: Logical && and Bitwise &
The logical && operator doesn't check second condition if first condition is false. It checks second condition only if first one is true.
The bitwise & operator always checks both conditions whether first condition is true or false.
Example:
class OperatorExample
{
public static void main(String args[])
{
int a=10;
int b=5;
int c=20;
System.out.println(a<b&&a<c);//false && true = false
System.out.println(a<b&a<c);//false & true = false
}
}
Output:
false
false

Assignment Operator
The *assignment operator* is the single equal sign, =.
Syntax:
var = expression;
Here, the type of *var* must be compatible with the type of *expression*.
int x, y, z;
x = y = z = 100; // set x, y, and z to 100
This fragment sets the variables **x**, **y**, and **z** to 100 using a single statement.

Ternary Operator
Ternary operator in java is used as one liner replacement for if-then-else statement and used a lot in java programming. it is the only conditional operator which takes three operands.
Syntax:
expression1 ? expression2 : expression3
Example:
class OperatorExample
{
public static void main(String args[])
{
int a=2;
int b=5;
int min=(a<b)?a:b;
System.out.println(min);
}
}

Output:
2

CONTROL STATEMENTS
Selection Statements in Java
A programming language uses control statements to control the flow of execution of program based on certain conditions.
Java's Selection statements:
- if
- if-else
- nested-if
- if-else-if
- switch-case
- jump – break, continue, return

if Statement
if statement is the most simple decision making statement. It is used to decide whether a certain statement or block of statements will be executed or not that is if a certain condition is true then a block of statement is executed otherwise not.
Syntax:
if(condition)
{
        //statements to execute if
        //condition is true
}
Condition after evaluation will be either true or false. If the value is true then it will execute the block of statements under it. If there are no curly braces '{' and '}' after **if(condition)** then by default if statement will consider the immediate one statement to be inside its block.
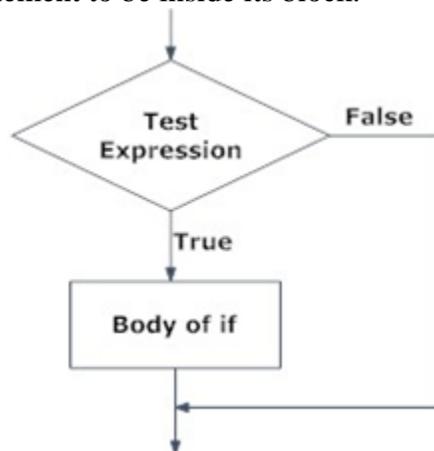
Example:
class IfSample
{
public static void main(String args[])
{
int x, y;
x = 10;
y = 20;

```java
if(x < y)
System.out.println("x is less than y");
x = x * 2;
if(x == y)
System.out.println("x now equal to y");
x = x * 2;
if(x > y)
System.out.println("x now greater than y");
// this won't display anything
if(x == y)
System.out.println("you won't see this");
}
}
```
Output:
x is less than y
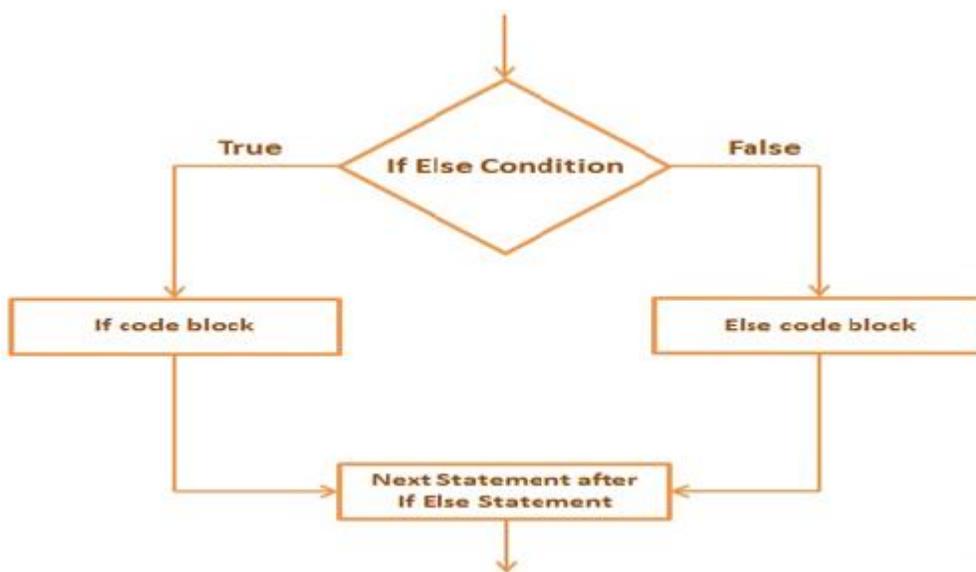x now equal to y
x now greater than y

if-else Statement
The Java if-else statement also tests the condition. It executes the *if* block if condition is true else if it is false the else block is executed.
Syntax:.
```java
If(condition)
{
        //Executes this block if
        //condition is true
}
else
{
        //Executes this block if
        //condition is false
}
```


Example:

```java
public class IfElseExample
{
public static void main(String[] args)
  {
    int number=13;
    if(number%2==0){
       System.out.println("even number");
   }else
   {
       System.out.println("odd number");
  } } }
```
Output:
odd number

Nested if Statement
Nested if-else statements, is that using one if or else if statement inside another if or else if statement(s).
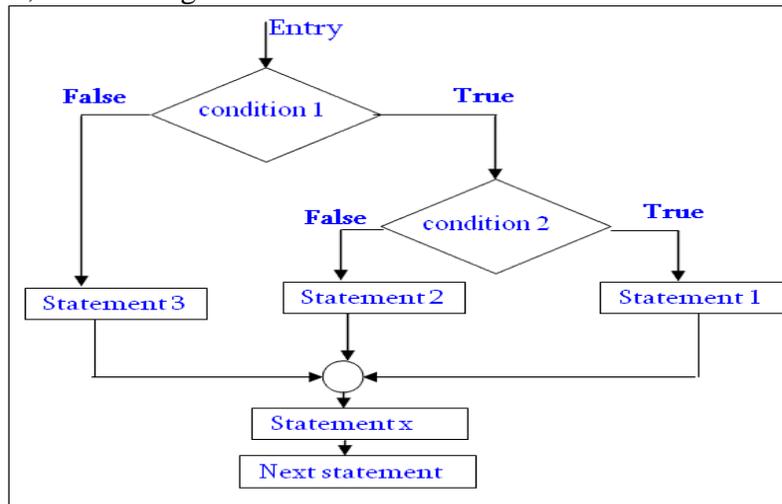
Example:
```java
// Java program to illustrate nested-if statement
class NestedIfDemo
{
   public static void main(String args[])
   {
     int i = 10;

     if (i == 10)
     {
        if (i < 15)
           System.out.println("i is smaller than 15");
        if (i < 12)
           System.out.println("i is smaller than 12 too");
        else
           System.out.println("i is greater than 15");
```
41

```
        }
    }
}
```
Output:
i is smaller than 15
i is smaller than 12 too

if-else-if ladder statement
The *if* statements are executed from the top down. The conditions controlling the *if* is true, the statement associated with that if is executed, and the rest of the ladder is bypassed. If none of the conditions is true, then the final else statement will be executed.
Syntax:
if(condition)
statement;
else if(condition)
statement;
else if(condition)
statement;
.
.
.
else
statement;
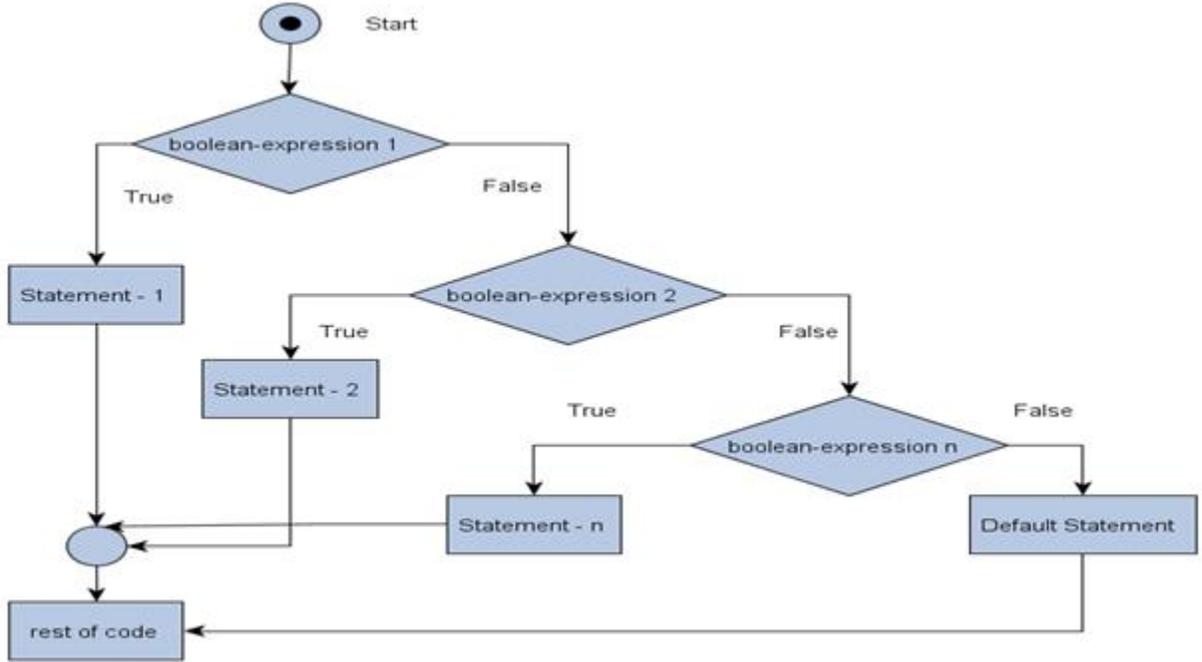
Example:
```
public class IfElseIfExample {
public static void main(String[] args) {
    int marks=65;
    if(marks<50){
        System.out.println("fail");
    }
    else if(marks>=50 && marks<60){
```

```
        System.out.println("D grade");
    }
    else if(marks>=60 && marks<70){
        System.out.println("C grade");
    }
    else if(marks>=70 && marks<80){
        System.out.println("B grade");
    }
    else if(marks>=80 && marks<90){
        System.out.println("A grade");
    }else if(marks>=90 && marks<100){
        System.out.println("A+ grade");
    }else{
        System.out.println("Invalid!");
    }
 }
}
```
Output:
C grade

Switch Statements
The **switch** statement is Java's multiway branch statement. It provides an easy way to dispatch execution to different parts of your code based on the value of an expression.
Syntax:
```
switch (expression) {
case value1:
// statement sequence
break;
case value2:
// statement sequence
break;
.
.
case valueN :
// statement sequence
break;
default:
// default statement sequence
}
```
Example:
```
// A simple example of the switch.
class SampleSwitch {
public static void main(String args[]) {
for(int i=0; i<6; i++)
switch(i) {
case 0:
System.out.println("i is zero.");
break;
case 1:
```

43

```
System.out.println("i is one.");
break;
case 2:
System.out.println("i is two.");
break;
case 3:
System.out.println("i is three.");
break;
default:
System.out.println("i is greater than 3.");
}}}
```
Output:
i is zero.
i is one.
i is two.
i is three.
i is greater than 3.
i is greater than 3.
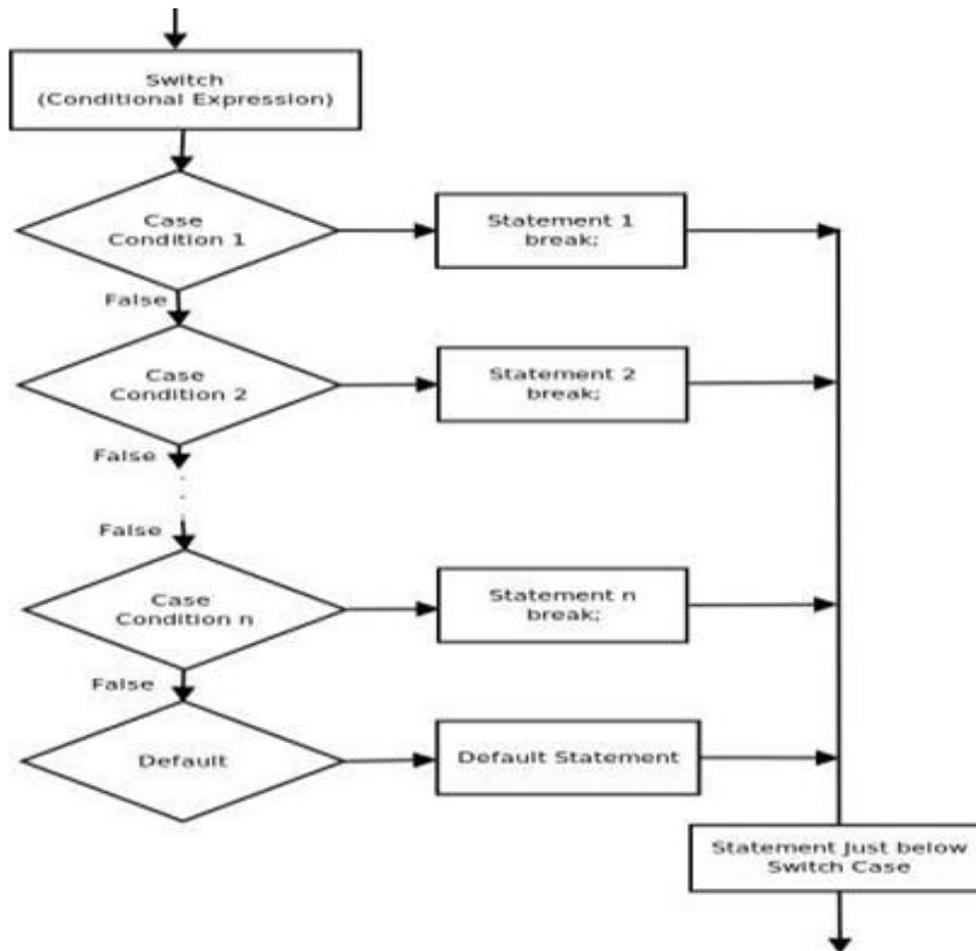
ITERATIVE STATEMENTS
In programming languages, loops are used to execute a set of instructions/functions repeatedly when some conditions become true. There are three types of loops in java.

- while loop
- do-while loop
- For loop

while loop

A while loop is a control flow statement that allows code to be executed repeatedly based on a given Boolean condition. The while loop can be thought of as a repeating if statement.

Syntax:

while(*condition*) {
// body of loop
}
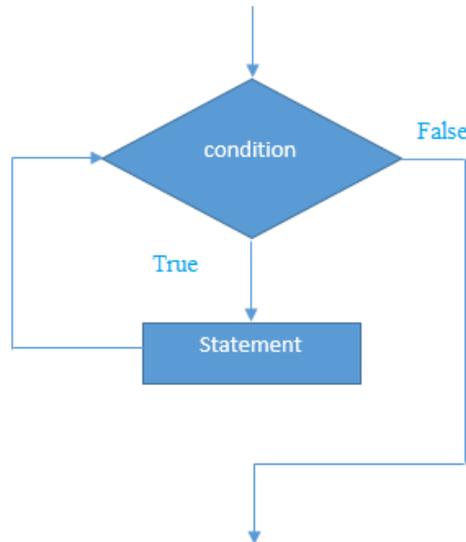
- While loop starts with the checking of condition. If it evaluated to true, then the loop body statements are executed otherwise first statement following the loop is executed. It is called as Entry controlled loop.
- Normally the statements contain an update value for the variable being processed for the next iteration.
- When the condition becomes false, the loop terminates which marks the end of its life cycle.

Example:

```
// Demonstrate the while loop.
class While {
public static void main(String args[]) {
int n = 5;
while(n > 0) {
System.out.println("tick " + n);
n--;
}
}
}
```

Output:

tick 5
tick 4
tick 3
tick 2
tick 1

do-while loop:
do while loop checks for condition after executing the statements, and therefore it is called as Exit Controlled Loop.
Syntax:
do {
// body of loop
} while (condition);
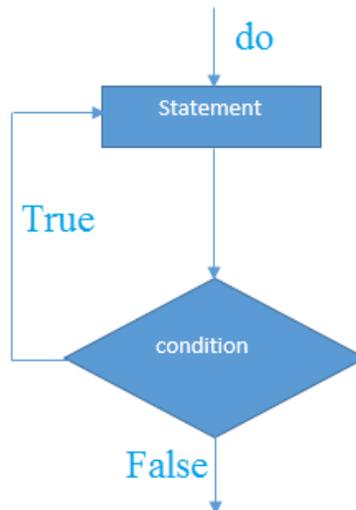
- do while loop starts with the execution of the statement(s). There is no checking of any condition for the first time.
- After the execution of the statements, and update of the variable value, the condition is checked for true or false value. If it is evaluated to true, next iteration of loop starts.
- When the condition becomes false, the loop terminates which marks the end of its life cycle.
- It is important to note that the do-while loop will execute its statements atleast once before any condition is checked, and therefore is an example of exit control loop.

Example
```
public class DoWhileExample {
public static void main(String[] args) {
    int i=1;
    do{
        System.out.println(i);
    i++;
    }while(i<=5);
}
}
```
Output:
1
2
3
4
5

for loop

for loop provides a concise way of writing the loop structure. A for statement consumes the initialization, condition and increment/decrement in one line.

Syntax

```
for(initialization; condition; iteration) {
// body
}
```
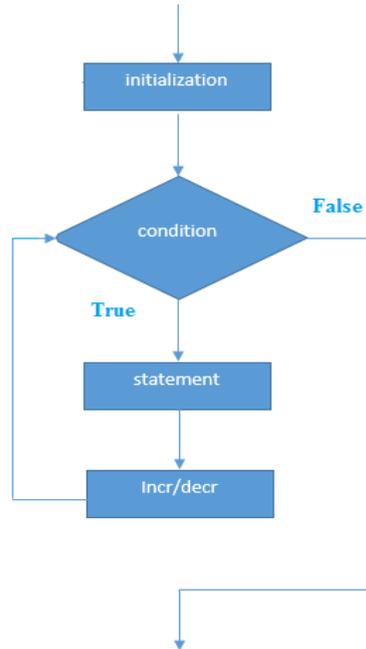

- **Initialization condition:** Here, we initialize the variable in use. It marks the start of a for loop. An already declared variable can be used or a variable can be declared, local to loop only.
- **Testing Condition:** It is used for testing the exit condition for a loop. It must return a boolean value. It is also an **Entry Control Loop** as the condition is checked prior to the execution of the loop statements.
- **Statement execution:** Once the condition is evaluated to true, the statements in the loop body are executed.
- **Increment/ Decrement:** It is used for updating the variable for next iteration.
- **Loop termination:**When the condition becomes false, the loop terminates marking the end of its life cycle.

Example

```
public class ForExample {
public static void main(String[] args) {
   for(int i=1;i<=5;i++){
     System.out.println(i);
   }
} }
```

Output:

1
2
3
4
5

for-each Loop

The for-each loop is used to traverse array or collection in java. It is easier to use than simple for loop because we don't need to increment value and use subscript notation. It works on elements basis not index. It returns element one by one in the defined variable.

Syntax:

for(*type itr-var* : *collection*) *statement-block*

Example:
```
// Use a for-each style for loop.
class ForEach {
public static void main(String args[]) {
int nums[] = { 1, 2, 3, 4, 5, 6, 7, 8, 9, 10 };
int sum = 0;
// use for-each style for to display and sum the values
for(int x : nums) {
System.out.println("Value is: " + x);
sum += x;
}
System.out.println("Summation: " + sum);
}
}
```

Output:
```
Value is: 1
Value is: 2
Value is: 3
Value is: 4
Value is: 5
Value is: 6
Value is: 7
Value is: 8
Value is: 9
Value is: 10
Summation: 55
```

Nested Loops

Java allows loops to be nested. That is, one loop may be inside another.

Example:
```
// Loops may be nested.
class Nested {
public static void main(String args[]) {
int i, j;
for(i=0; i<10; i++) {
for(j=i; j<10; j++)
System.out.print(".");
System.out.println();
}}
}
```

Output:

....
....
....
....
......
.....
....
...
..
.

JUMP STATEMENTS
Java Break Statement
- When a break statement is encountered inside a loop, the loop is immediately terminated and the program control resumes at the next statement following the loop.
- The Java *break* is used to break loop or switch statement. It breaks the current flow of the program at specified condition. In case of inner loop, it breaks only inner loop.

Example:
```
// Using break to exit a loop.
class BreakLoop {
public static void main(String args[]) {
for(int i=0; i<100; i++) {
if(i == 10) break; // terminate loop if i is 10
System.out.println("i: " + i);
}
System.out.println("Loop complete.");
}
}
```
Output:
```
i: 0
i: 1
i: 2
i: 3
i: 4
i: 5
i: 6
i: 7
```

i: 8
i: 9
Loop complete.

Java Continue Statement
- The continue statement is used in loop control structure when you need to immediately jump to the next iteration of the loop. It can be used with for loop or while loop.
- The Java *continue statement* is used to continue loop. It continues the current flow of the program and skips the remaining code at specified condition. In case of inner loop, it continues only inner loop.

Example:
```
// Demonstrate continue.
class Continue {
public static void main(String args[]) {
for(int i=0; i<10; i++) {
System.out.print(i + " ");
if (i%2 == 0) continue;
System.out.println("");
}
}
}
```
This code uses the **%** operator to check if **i** is even. If it is, the loop continues without
printing a newline.
Output:
```
0 1
2 3
4 5
6 7
8 9
```

Return
The last control statement is return. The return statement is used to explicitly return from a method. That is,
it causes program control to transfer back to the caller of the method.
Example:
```
// Demonstrate return.
class Return {
public static void main(String args[]) {
boolean t = true;
System.out.println("Before the return.");
if(t) return; // return to caller
System.out.println("This won't execute.");
}
}
```
**Output:**
Before the return.

ARRAYS
- Array is a collection of similar type of elements that have contiguous memory location.

- In Java all arrays are dynamically allocated.
- Since arrays are objects in Java, we can find their length using member length.
- A Java array variable can also be declared like other variables with [] after the data type.
- The variables in the array are ordered and each have an index beginning from 0.
- Java array can be also be used as a static field, a local variable or a method parameter.
- The **size** of an array must be specified by an int value and not long or short.
- The direct superclass of an array type is Object.
- Every array type implements the interfaces Cloneable and java.io.Serializable.
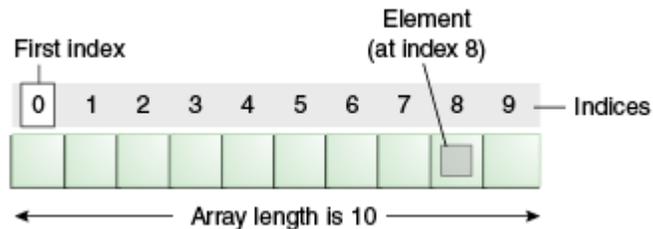
Advantage of Java Array
- **Code Optimization:** It makes the code optimized, we can retrieve or sort the data easily.
- **Random access:** We can get any data located at any index position.

Disadvantage of Java Array
- **Size Limit:** We can store only fixed size of elements in the array. It doesn't grow its size at runtime. To solve this problem, collection framework is used in java.

Types of Array in java
1. One- Dimensional Array
2. Multidimensional Array

One-Dimensional Arrays
An array is a group of like-typed variables that are referred to by a common name. An array declaration has two components: the type and the name. *type* declares the element type of the array. The element type determines the data type of each element that comprises the array. We can also create an array of other primitive data types like char, float, double..etc or user defined data type(objects of a class).Thus, the element type for the array determines what type of data the array will hold.
Syntax:
type var-name[ ];
Instantiation of an Array in java
array-var = new type [size];
Example:
class Testarray{
public static void main(String args[]){
int a[]=new int[5];//declaration and instantiation
a[0]=10;//initialization
a[1]=20;
a[2]=70;
a[3]=40;
a[4]=50;

```
//printing array
for(int i=0;i<a.length;i++)//length is the property of array
System.out.println(a[i]);
}}
```
Output:
10
20
70
40
50

Declaration, Instantiation and Initialization of Java Array
Example:
```
class Testarray1{
public static void main(String args[]){
int a[]={33,3,4,5};//declaration, instantiation and initialization
//printing array
for(int i=0;i<a.length;i++)//length is the property of array
System.out.println(a[i]);
}}
```
Output:
33
3
4
5

Passing Array to method in java
We can pass the java array to method so that we can reuse the same logic on any array.
Example:
```
class Testarray2{
static void min(int arr[]){
int min=arr[0];
for(int i=1;i<arr.length;i++)
 if(min>arr[i])
  min=arr[i];
 System.out.println(min);
}
public static void main(String args[]){
int a[]={33,3,4,5};
min(a);//passing array to method
}}
```
Output:
3

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. A multidimensional array is created by appending one set of square brackets ([]) per dimension.

Syntax:
type var-name[ ][ ]=new type[row-size][col-size];
Example:
// Demonstrate a two-dimensional array.
class TwoDArray {
public static void main(String args[]) {
int twoD[][]= new int[4][5];
int i, j, k = 0;
for(i=0; i<4; i++)
for(j=0; j<5; j++) {
twoD[i][j] = k;
k++;
}
for(i=0; i<4; i++) {
for(j=0; j<5; j++)
System.out.print(twoD[i][j] + " ");
System.out.println();
}
}
}
Output:
0 1 2 3 4
5 6 7 8 9
10 11 12 13 14
15 16 17 18 19
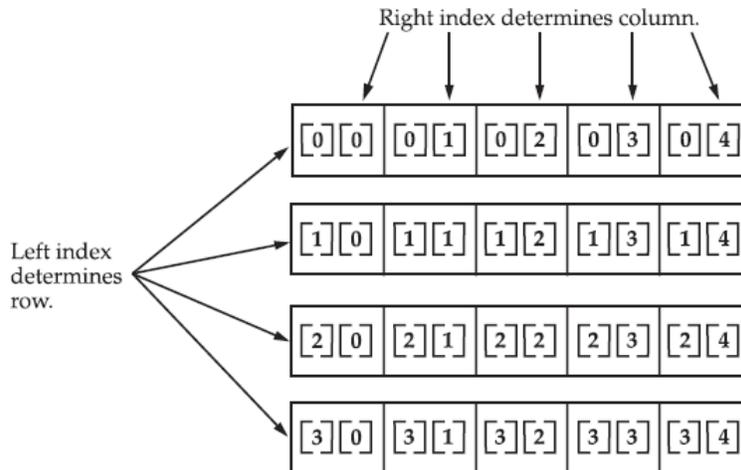
Given: int twoD [ ] [ ] = new int [4] [5] ;

When you allocate memory for a multidimensional array, you need only specify the memory for the first (leftmost) dimension. You can allocate the remaining dimensions separately. For example, this following code allocates memory for the first dimension of **twoD** when it is declared. It allocates the second dimension manually.
Syntax:
int twoD[][] = new int[4][];
twoD[0] = new int[5];
twoD[1] = new int[5];
twoD[2] = new int[5];

53

```
twoD[3] = new int[5];
```
Example:
```
// Manually allocate differing size second dimensions.
class TwoDAgain {
public static void main(String args[]) {
int twoD[][] = new int[4][];
twoD[0] = new int[1];
twoD[1] = new int[2];
twoD[2] = new int[3];
twoD[3] = new int[4];
int i, j, k = 0;
for(i=0; i<4; i++)
for(j=0; j<i+1; j++) {
twoD[i][j] = k;
k++;
}
for(i=0; i<4; i++) {
for(j=0; j<i+1; j++)
System.out.print(twoD[i][j] + " ");
System.out.println();
}
}
}
```
**Output:**
```
0
1 2
3 4 5
6 7 8 9
```
The array created by this program looks like this:
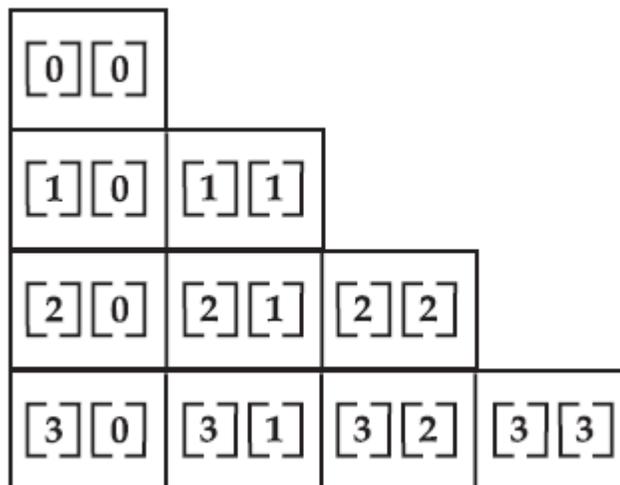
PACKAGES

A java package is a group of similar types of classes, interfaces and sub-packages. Package in java can be categorized in two form, built-in package and user-defined package. There are many built-in packages such as java, lang, awt, javax, swing, net, io, util, sql etc.

Advantage of Java Package

1) Java package is used to categorize the classes and interfaces so that they can be easily maintained.
2) Java package provides access protection.
3) Java package removes naming collision.

Defining a Package

      To create a package include a **package** command as the first statement in a Java source file. Any classes declared within that file will belong to the specified package. The **package** statement defines a name space in which classes are stored. If **package** statement is omitted, the class names are put into the default package, which has no name.

Syntax:

package <fully qualified package name>;

package *pkg*;

Here, *pkg* is the name of the package. For example, the following statement creates a package called MyPackage.

package MyPackage;

Java uses file system directories to store packages. For example, the **.class** files for any classes you declare to be part of **MyPackage** must be stored in a directory called **MyPackage**.

It is possible to create a hierarchy of packages. The general form of a multileveled package statement is shown here:

package pkg1[.pkg2[.pkg3]];
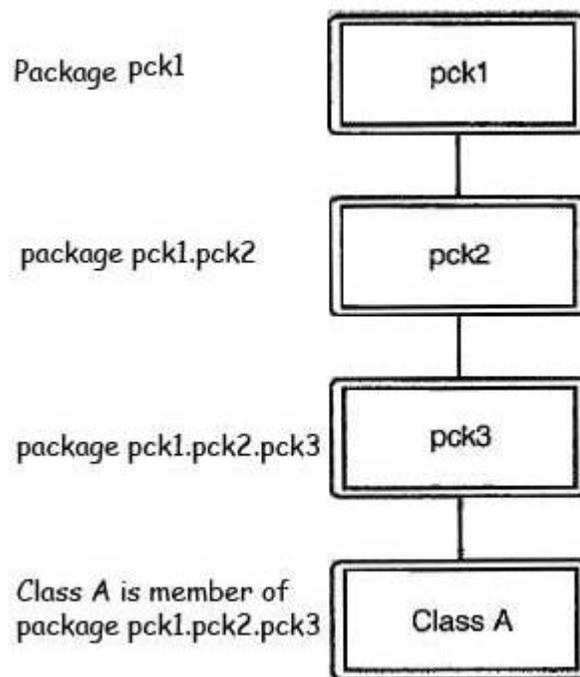
A package hierarchy must be reflected in the file system of your Java development system. For example, a package declared as

package java.awt.image;

needs to be stored in **java\awt\image** in a Windows environment. We cannot rename a package without renaming the directory in which the classes are stored.

Finding Packages and CLASSPATH

First, by default, the Java run-time system uses the current working directory as its starting point. Thus, if your package is in a subdirectory of the current directory, it will be found. Second, you can specify a directory path or paths by setting the **CLASSPATH** environmental variable. Third, you can use the **-classpath** option with **java** and **javac** to specify the path to your classes.

consider the following package specification:
package MyPack
        In order for a program to find **MyPack**, one of three things must be true. Either the program can be executed from a directory immediately above **MyPack**, or the **CLASSPATH** must be set to include the path to **MyPack**, or the **-classpath** option must specify the path to **MyPack** when the program is run via **java**. When the second two options are used, the class path *must not* include **MyPack**, itself. It must simply specify the *path to* **MyPack**. For example, in a Windows environment, if the path to **MyPack** is
C:\MyPrograms\Java\MyPack
then the class path to **MyPack** is
C:\MyPrograms\Java
Example:

```
// A simple package
package MyPack;
class Balance {
String name;
double bal;
Balance(String n, double b) {
name = n;
bal = b;
}
void show() {
if(bal<0)
System.out.print("--> ");
System.out.println(name + ": $" + bal);
}
}
class AccountBalance {
public static void main(String args[]) {
Balance current[] = new Balance[3];
current[0] = new Balance("K. J. Fielding", 123.23);
current[1] = new Balance("Will Tell", 157.02);
current[2] = new Balance("Tom Jackson", -12.33);
for(int i=0; i<3; i++) current[i].show();
}
}
```

Call this file **AccountBalance.java** and put it in a directory called **MyPack**.
Next, compile the file. Make sure that the resulting **.class** file is also in the **MyPack** directory. Then, try executing the **AccountBalance** class, using the following command line:
java MyPack.AccountBalance
Remember, you will need to be in the directory above **MyPack** when you execute this command. (Alternatively, you can use one of the other two options described in the preceding section to specify the path **MyPack**.)
As explained, **AccountBalance** is now part of the package **MyPack**. This means that it cannot be executed by itself. That is, you cannot use this command line:

java AccountBalance
AccountBalance must be qualified with its package name.

Example:
```
package pck1;
class Student
{
                private int rollno;
                private String name;
                private String address;
                public Student(int rno, String sname, String sadd)
            {

                rollno = rno;
                name = sname;
                address = sadd;

            }
                public void showDetails()
            {
                System.out.println("Roll No :: " + rollno);
                System.out.println("Name :: " + name);
                System.out.println("Address :: " + address);
            }
}
            public class DemoPackage
        {
                public static void main(String ar[])
            {
                Student st[]=new Student[2];
                st[0] = new Student (1001,"Alice", "New York");
                st[1] = new Student(1002,"BOb","Washington");
                st[0].showDetails();
                st[1].showDetails();
            }
        }
```
There are two ways to create package directory as follows:
1. Create the folder or directory at your choice location with the same name as package name. After compilation of copy *.class* (byte code file) file into this folder.
2. Compile the file with following syntax.
javac -d <target location of package> sourceFile.java

The above syntax will create the package given in the *sourceFile* at the *<target location of pacakge>* if it is not yet created. If package already exist then only the *.class* (byte code file) will be stored to the package given in *sourceFile.*

Steps to compile the given example code:
Compile the code with the command on the command prompt.
    javac -d DemoPackage.java
1. The command will create the package at the current location with the name pck1, and contains the file DemoPackage.class and Student.class

2. To run write the command given below
   java pckl.DemoPackage

Note: The DemoPackate.class is now stored in pck1 package. So that we've to use *fully qualified type name* to run or access it.

Output:
 Roll No :: 1001
Name :: Alice
Address :: New York
Roll No :: 1002
Name :: Bob
Address :: Washington

UNIT II	INHERITANCE AND INTERFACES

Inheritance – Super classes- sub classes –Protected members – constructors in sub classes- the Object class – abstract classes and methods- final methods and classes – Interfaces – defining an interface, implementing interface, differences between classes and interfaces and extending interfaces - Object cloning -inner classes, Array Lists – Strings

INHERITANCE

Inheritance can be defined as the procedure or mechanism of acquiring all the properties and behaviors of one class to another, i.e., acquiring the properties and behavior of child class from the parent class.

When one object acquires all the properties and behaviours of another object, it is known as inheritance. Inheritance represents the **IS-A relationship**, also known as *parent-child* relationship.
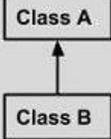
Uses of inheritance in java

- For Method Overriding (so runtime polymorphism can be achieved).
- For Code Reusability.

Types of inheritance in java: single, multilevel and hierarchical inheritance. Multiple and hybrid inheritance is supported through interface only.

Syntax:

class subClass extends superClass
{
  //methods and fields
}

Single Inheritance	Class A → Class B	public class A {
		....
		}
		public class B **extends** A {
		....
		}
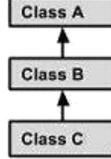
Multi Level Inheritance	Class A → Class B → Class C	public class A { ....} public class B **extends** A {...................} public class C **extends** B {..................... }
Hierarchical Inheritance	Class A → Class B, Class C	public class A { ....} public class B **extends** A {...................} public class C **extends** A {..................... }
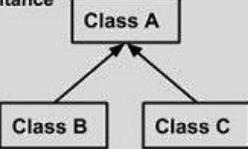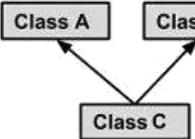
Multiple Inheritance	Class A, Class B → Class C	public class A { ....}
		
		public class B {...................}
		
		public class C **extends** A,B {
		....
		} // Java does not support mutiple Inheritance

Terms used in **Inheritence**
- **Class:** A class is a group of objects which have common properties. It is a template or blueprint from which objects are created.
- **Sub Class/Child Class:** Subclass is a class which inherits the other class. It is also called a derived class, extended class, or child class.
- **Super Class/Parent Class:** Superclass is the class from where a subclass inherits the features. It is also called a base class or a parent class.
- **Reusability:** As the name specifies, reusability is a mechanism which facilitates you to reuse the fields and methods of the existing class when you create a new class. You can use the same fields and methods already defined in previous class.

<u>**SINGLE INHERITANCE**</u>
In Single Inheritance one class extends another class (one class only).
Example:

```java
public class ClassA
{
    public void dispA()
    {
        System.out.println("disp() method of ClassA");
    }
}
public class ClassB extends ClassA
{
    public void dispB()
    {
        System.out.println("disp() method of ClassB");
    }
    public static void main(String args[])
    {
        //Assigning ClassB object to ClassB reference
        ClassB b = new ClassB();
        //call dispA() method of ClassA
        b.dispA();
        //call dispB() method of ClassB
        b.dispB();
    }
}
```

Output :

disp() method of ClassA
disp() method of ClassB

MULTILEVEL INHERITANCE

In Multilevel Inheritance, one class can inherit from a derived class. Hence, the derived class becomes the base class for the new class.

Example:

```java
public class ClassA
{
   public void dispA()
   {
      System.out.println("disp() method of ClassA");
   }
}
public class ClassB extends ClassA
{
   public void dispB()
   {
      System.out.println("disp() method of ClassB");
   }
}
public class ClassC extends ClassB
{
   public void dispC()
   {
      System.out.println("disp() method of ClassC");
   }
   public static void main(String args[])
   {
      //Assigning ClassC object to ClassC reference
      ClassC c = new ClassC();
      //call dispA() method of ClassA
      c.dispA();
      //call dispB() method of ClassB
      c.dispB();
      //call dispC() method of ClassC
      c.dispC();
   }
}
```

Output :

disp() method of ClassA
disp() method of ClassB
disp() method of ClassC

HIERARCHICAL INHERITANCE

In Hierarchical Inheritance, one class is inherited by many sub classes.

Example:

```java
public class ClassA
{
   public void dispA()
   {
      System.out.println("disp() method of ClassA");
   }
}
public class ClassB extends ClassA
{
   public void dispB()
   {
      System.out.println("disp() method of ClassB");
   }
}
public class ClassC extends ClassA
{
   public void dispC()
   {
      System.out.println("disp() method of ClassC");
   }
}
public class ClassD extends ClassA
{
   public void dispD()
   {
      System.out.println("disp() method of ClassD");
   }
}
public class HierarchicalInheritanceTest
{
   public static void main(String args[])
   {
      //Assigning ClassB object to ClassB reference
      ClassB b = new ClassB();
      //call dispB() method of ClassB
      b.dispB();
      //call dispA() method of ClassA
      b.dispA();
       //Assigning ClassC object to ClassC reference
      ClassC c = new ClassC();
      //call dispC() method of ClassC
      c.dispC();
```

```
        //call dispA() method of ClassA
        c.dispA();

        //Assigning ClassD object to ClassD reference
        ClassD d = new ClassD();
        //call dispD() method of ClassD
        d.dispD();
        //call dispA() method of ClassA
        d.dispA();
    }
}
```
Output :

disp() method of ClassB
disp() method of ClassA
disp() method of ClassC
disp() method of ClassA
disp() method of ClassD
disp() method of ClassA

Hybrid Inheritance is the combination of both Single and Multiple Inheritance. Again Hybrid inheritance is also not directly supported in Java only through interface we can achieve this. Flow diagram of the Hybrid inheritance will look like below. As you can ClassA will be acting as the Parent class for ClassB & ClassC and ClassB & ClassC will be acting as Parent for ClassD.

Multiple Inheritance is nothing but one class extending more than one class. Multiple Inheritance is basically not supported by many Object Oriented Programming languages such as Java, Small Talk, C# etc.. (C++ Supports Multiple Inheritance). As the Child class has to manage the dependency of more than one Parent class. But you can achieve multiple inheritance in Java using Interfaces.

"super" KEYWORD
Usage of super keyword

1. super() invokes the constructor of the parent class.
2. super.variable_name refers to the variable in the parent class.
3. super.method_name refers to the method of the parent class.

1. super() invokes the constructor of the parent class

    super() will invoke the constructor of the parent class. Even when you don't add **super()** keyword the compiler will add one and will invoke the Parent Class constructor.
Example:
class ParentClass
{
    ParentClass()

```
                {
                        System.out.println("Parent Class default Constructor");
                }
}
public class SubClass extends ParentClass
{
        SubClass()
        {
                System.out.println("Child Class default Constructor");
        }
        public static void main(String args[])
        {
                SubClass s = new SubClass();
        }
}
```

Output:
Parent Class default Constructor
Child Class default Constructor

Even when we add explicitly also it behaves the same way as it did before.
```
class ParentClass
{
        public ParentClass()
        {
                System.out.println("Parent Class default Constructor");
        }
}
public class SubClass extends ParentClass
{
        SubClass()
        {
         super();
                System.out.println("Child Class default Constructor");
        }
        public static void main(String args[])
        {
                SubClass s = new SubClass();
        }
}
```
Output:
Parent Class default Constructor
Child Class default Constructor

You can also call the parameterized constructor of the Parent Class. For example, **super(10)** will call parameterized constructor of the Parent class.

```java
class ParentClass
{
        ParentClass()
        {
                System.out.println("Parent Class default Constructor called");
        }
         ParentClass(int val)
        {
                System.out.println("Parent Class parameterized Constructor, value: "+val);
        }
}
public class SubClass extends ParentClass
{
        SubClass()
        {
                super();//Has to be the first statement in the constructor
                System.out.println("Child Class default Constructor called");
        }
        SubClass(int val)
        {
                super(10);
                System.out.println("Child Class parameterized Constructor, value: "+val);
        }
        public static void main(String args[])
        {
                //Calling default constructor
                SubClass s = new SubClass();
                //Calling parameterized constructor
                SubClass s1 = new SubClass(10);
        }
}
```

Output

Parent Class default Constructor called
Child Class default Constructor called

Parent Class parameterized Constructor, value: 10
Child Class parameterized Constructor, value: 10

2. *super.variable_name refers to the variable in the parent class*

When we have the same variable in both parent and subclass
class ParentClass
{

```
            int val=999;
}
public class SubClass extends ParentClass
{
        int val=123;

        void disp()
        {
                System.out.println("Value is : "+val);
        }

        public static void main(String args[])
        {
                SubClass s = new SubClass();
                s.disp();
        }
}
```

Output

Value is : 123

This will call only the **val** of the sub class only. Without **super** keyword, you cannot call the **val** which is present in the Parent Class.

```
class ParentClass
{
        int val=999;
}
public class SubClass extends ParentClass
{
        int val=123;

        void disp()
        {
                System.out.println("Value is : "+super.val);
        }

        public static void main(String args[])
        {
                SubClass s = new SubClass();
                s.disp();
        }
}
```

Output

Value is : 999

3. super.method_nae refers to the method of the parent class

When you override the Parent Class method in the Child Class without super keywords support you will not be able to call the Parent Class method. Let's look into the below example

```java
class ParentClass
{
        void disp()
        {
                System.out.println("Parent Class method");
        }
}
public class SubClass extends ParentClass
{

        void disp()
        {
                System.out.println("Child Class method");
        }

        void show()
        {
                disp();
        }
        public static void main(String args[])
        {
                SubClass s = new SubClass();
                s.show();
        }
}
```

Output:

Child Class method

Here we have overridden the **Parent Class disp()** method in the SubClass and hence **SubClass disp()** method is called. If we want to call the **Parent Class disp()** method also means then we have to use the super keyword for it.

```java
class ParentClass
{
         void disp()
        {
                System.out.println("Parent Class method");
        }
}
public class SubClass extends ParentClass
{
```

```
            void disp()
            {
                     System.out.println("Child Class method");
            }

            void show()
            {
                     //Calling SubClass disp() method
                     disp();
                     //Calling ParentClass disp()
                     method super.disp();
            }
            public static void main(String args[])
            {
                     SubClass s = new SubClass();
                     s.show();
            }
}
```
Output

Child Class method
Parent Class method

When there is no method overriding then by default **Parent Class disp()** method will be called.
```
class ParentClass
{
            public void disp()
            {
                     System.out.println("Parent Class method");
            }
}
public class SubClass extends ParentClass
{
            public void show()
            {
                     disp();
            }
            public static void main(String args[])
            {
                     SubClass s = new SubClass();
                     s.show(); }}
```

Output:
Parent Class method

The Object Class

There is one special class, **Object**, defined by Java. All other classes are subclasses of **Object**. That is, **Object** is a superclass of all other classes. This means that a reference variable of type **Object** can refer to an object of any other class. Also, since arrays are implemented as

classes, a variable of type **Object** can also refer to any array. **Object** defines the following methods, which means that they are available in every object.

Method	Purpose
Object clone( )	Creates a new object that is the same as the object being cloned.
boolean equals(Object *object*)	Determines whether one object is equal to another.
void finalize( )	Called before an unused object is recycled.
Class<?> getClass( )	Obtains the class of an object at run time.
int hashCode( )	Returns the hash code associated with the invoking object.
void notify( )	Resumes execution of a thread waiting on the invoking object.
void notifyAll( )	Resumes execution of all threads waiting on the invoking object.
String toString( )	Returns a string that describes the object.
void wait( ) void wait(long *milliseconds*) void wait(long *milliseconds*, int *nanoseconds*)	Waits on another thread of execution.

The methods **getClass( )**, **notify( )**, **notifyAll( )**, and **wait( )** are declared as **final**. You may override the others. These methods are described elsewhere in this book. However, notice two methods now: **equals( )** and **toString( )**. The **equals( )** method compares two objects. It returns **true** if the objects are equal, and **false** otherwise. The precise definition of equality can vary, depending on the type of objects being compared. The **toString( )** method returns a string that contains a description of the object on which it is called. Also, this method is automatically called when an object is output using **println( )**. Many classes override this method. Doing so allows them to tailor a description specifically for the types of objects that they create.

ABSTRACT CLASS

A class that is declared with abstract keyword, is known as abstract class in java. It can have abstract and non-abstract methods (method with body).Abstraction is a process of hiding the implementation details and showing only functionality to the user. Abstraction lets you focus on what the object does instead of how it does it. It needs to be extended and its method implemented. It cannot be instantiated.

Example abstract class:

abstract class A{}
abstract method:

A method that is declared as abstract and does not have implementation is known as abstract method.

abstract void printStatus();//no body and abstract

In this example, Shape is the abstract class, its implementation is provided by the Rectangle and Circle classes.

If you create the instance of Rectangle class, draw() method of Rectangle class will be invoked.

Example1:

File: TestAbstraction1.java

```
abstract class Shape{
abstract void draw();
}
//In real scenario, implementation is provided by others i.e. unknown by end
user class Rectangle extends Shape{
void draw(){System.out.println("drawing rectangle");}
}
class Circle1 extends Shape{
void draw(){System.out.println("drawing circle");}
}
//In real scenario, method is called by programmer or user
class TestAbstraction1{
public static void main(String args[]){
Shape s=new Circle1();//In real scenario, object is provided through method e.g. getShape() method
s.draw();
}
}
```

Output:

drawing circle

Abstract class having constructor, data member, methods

An abstract class can have data member, abstract method, method body, constructor and even main() method.

Example2:

File: TestAbstraction2.java

```
//example of abstract class that have method body
 abstract class Bike{
   Bike(){System.out.println("bike is created");}
   abstract void run();
   void changeGear(){System.out.println("gear changed");}
 }
 class Honda extends Bike{
 void run(){System.out.println("running safely..");}
 }
```

```
 class TestAbstraction2{
 public static void main(String args[]){
  Bike obj = new Honda();
  obj.run();
  obj.changeGear();
 }
}
```

Output:

bike is created
  running safely..
  gear changed

The abstract class can also be used to provide some implementation of the interface. In such case, the end user may not be forced to override all the methods of the interface.

Example3:

```
interface A{
void a();
void b();
void c();
void d();
}
abstract class B implements A{
public void c(){System.out.println("I am c");}
}
class M extends B{
public void a(){System.out.println("I am a");}
public void b(){System.out.println("I am b");}
public void d(){System.out.println("I am d");}
}
 class Test5{
public static void main(String args[]){
A a=new M();
a.a();
a.b();
a.c();
a.d();
}}
```
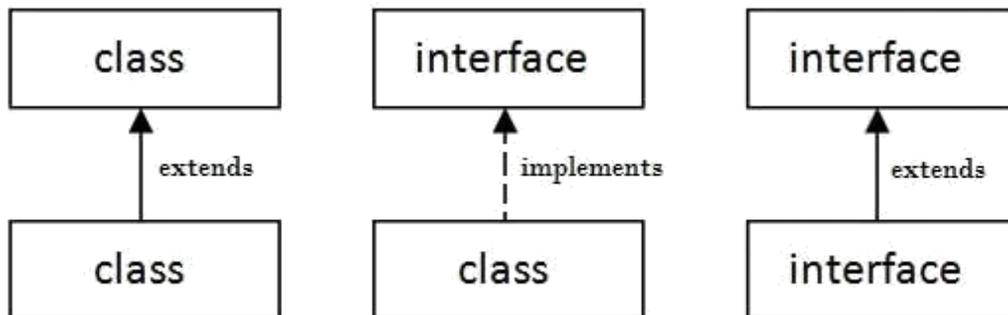
Output:

I am a
I am b
I am c
I am d

Example: 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");}
}
class TestInterface1{
public static void main(String args[]){

Drawable d=new Circle();//In real scenario, object is provided by method e.g. getDrawable()
d.draw();
}
}
Output:
drawing circle

Multiple inheritance in Java by interface
If a class implements multiple interfaces, or an interface extends multiple interfaces i.e. known as multiple inheritance.
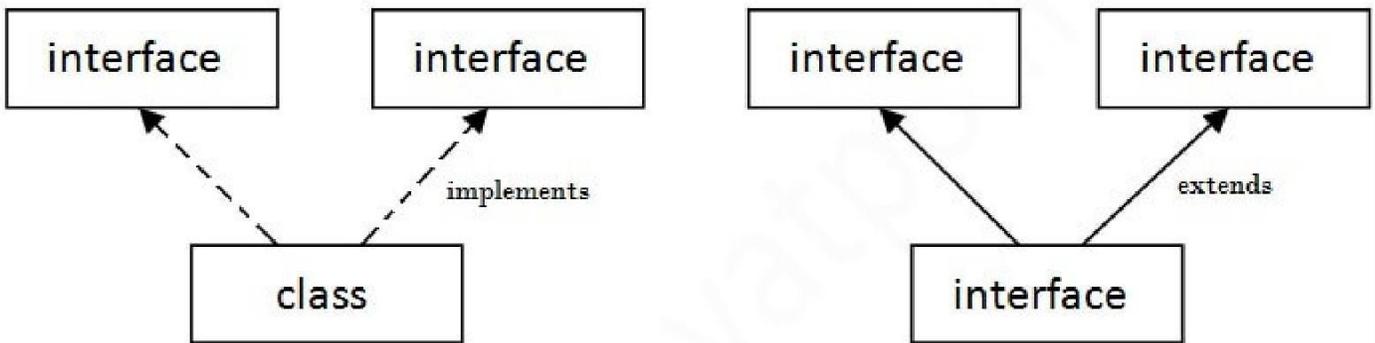
Example: interface
Printable{ void
print();
}
interface Showable{
void show();
}
class A7 implements Printable,Showable{
public void print(){System.out.println("Hello");}

```
public void show(){System.out.println("Welcome");}

public static void main(String args[]){
A7 obj = new A7();
obj.print();
obj.show();
 }
}
```
Output:
Hello
Welcome
Interface inheritance

A class implements interface but one interface extends another interface .
Example:
```
interface Printable{
void print();
}
interface Showable extends Printable{
void show();
}
class TestInterface4 implements Showable{
public void print(){System.out.println("Hello");}

public void show(){System.out.println("Welcome");}
public static void main(String args[]){ TestInterface4
obj = new TestInterface4(); obj.print();

obj.show();
 }}
```
Output:
Hello
Welcome

Nested Interface in Java
An interface can have another interface i.e. known as nested interface.
```
interface printable{
      void print();
      interface MessagePrintable{
      void msg();
      }
 }
```

Key points to remember about interfaces:

1) We can't instantiate an interface in java. That means we cannot create the object of an interface

2) Interface provides full abstraction as none of its methods have body. On the other hand abstract class provides partial abstraction as it can have abstract and concrete(methods with body) methods both.

3) "implements" keyword is used by classes to implement an interface.

4) While providing implementation in class of any method of an interface, it needs to be mentioned as public.

5) Class that implements any interface must implement all the methods of that interface, else the class should be declared abstract.

6) Interface cannot be declared as private, protected or transient.

7) All the interface methods are by default abstract and public.

8) Variables declared in interface are public, static and final by default.

interface Try
{
  int a=10; public
  int a=10;

  public static final int a=10;
  final int a=10;
  static int a=0;
}

All of the above statements are identical.

9) Interface variables must be initialized at the time of declaration otherwise compiler will throw an error.

interface Try
{
    int x;//Compile-time error
}

Above code will throw a compile time error as the value of the variable x is not initialized at the time of declaration.

10) Inside any implementation class, you cannot change the variables declared in interface because by default, they are public, static and final. Here we are implementing the interface

"Try" which has a variable x. When we tried to set the value for variable x we got compilation error as the variable x is public static **final** by default and final variables can not be re-initialized.

class Sample implements Try
{
  public static void main(String args[])
  {
    x=20; //compile time error
  }
}

11) An interface can extend any interface but cannot implement it. Class implements interface and interface extends interface.

12) A class can implement any number of interfaces.

13) If there are two or more same methods in two interfaces and a class implements both interfaces, implementation of the method once is enough.

```
interface A
{
  public void aaa();
}
interface B
{
  public void aaa();
}
class Central implements A,B
{
  public void aaa()
  {
     //Any Code here
  }
  public static void main(String args[])
  {
     //Statements
  }
}
```

14) A class cannot implement two interfaces that have methods with same name but different return type.

```
interface A
{
  public void aaa();
}
interface B
{
  public int aaa();
}
class Central implements A,B
{
  public void aaa() // error
  {
  }
  public int aaa() // error
  {
  }
  public static void main(String args[])
  {
```

```
   }
}
```

15) Variable names conflicts can be resolved by interface

name. interface A

```
{
   int x=10;
}
interface B
{
   int x=100;
}
class Hello implements A,B
{
   public static void Main(String args[])
   {
        System.out.println(x);
     System.out.println(A.x);
     System.out.println(B.x);
   }
}
```

Advantages of interface in java:

- Without bothering about the implementation part, we can achieve the security of implementation
- In java, **multiple inheritance** is not allowed, however you can use interface to make use of it as you can implement more than one interface.

DIFFERENCE BETWEEN ABSTRACT CLASS AND INTERFACE

ABSTRACT CLASS	INTERFACE
1) Abstract class can **have abstract and non-abstract** methods.	Interface can have **only abstract** methods. Sinc Java 8, it can have **default and static methods** also.
2) Abstract class **doesn't support multiple inheritance**.	Interface **supports multiple inheritance**.
3) Abstract class **can have final, non-final, static and non-static variables**.	Interface has **only static and final variables**.
4) Abstract class **can provide the implementation of interface**.	Interface **can't provide the implementation of abstract class**.
5) The **abstract keyword** is used to declare abstract class.	The **interface keyword** is used to declare interface.

6) An **abstract class** can extend another Java class and implement multiple Java interfaces.	An **interface** can extend another Java interface only.
7) An **abstract class** can be extended using keyword extends.	An **interface class** can be implemented using keyword implements
8) A Java **abstract class** can have class members like private, protected, etc.	Members of a Java interface are public by default.

9)**Example:**
public abstract class Shape{
public abstract void draw();
}
	Example:
	public interface Drawable{
	void draw();
	}

FINAL KEYWORD
Final keyword can be used along with **variables, methods** and **classes**.
1) final variable
2) final method
3) final class

1. Java final variable

A **final variable** is a variable whose value **cannot** be changed at anytime once assigned, it remains as a constant forever.
Example:

```
public class Travel
{
   final int SPEED=60;
   void increaseSpeed(){
     SPEED=70;
   }
   public static void main(String args[])
   {
     Travel t=new Travel();
     t.increaseSpeed();
   }
}
```
Output :

Exception in thread "main" java.lang.Error: Unresolved compilation problem:
        The final field Travel.SPEED cannot be assigned

The above code will give you Compile time error, as we are trying to change the value of a final variable **'SPEED'**.

2. Java final method

When you declare a method as **final**, then it is called as **final method**. A **final method cannot be overridden**.

```
package com.javainterviewpoint;

class Parent
{
   public final void disp()
   {
     System.out.println("disp() method of parent class");
   }
}
public class Child extends Parent
{
   public void disp()
   {
     System.out.println("disp() method of child class");
   }
   public static void main(String args[])
   {
     Child c = new Child();
     c.disp();
   }
}
```

Output : We will get the below error as we are overriding the **disp()** method of the **Parent** class.
Exception in thread "main" java.lang.VerifyError: class com.javainterviewpoint.Child overrides final method disp.()

```
        at java.lang.ClassLoader.defineClass1(Native Method)
        at java.lang.ClassLoader.defineClass(Unknown Source)
        at java.security.SecureClassLoader.defineClass(Unknown Source)
        at java.net.URLClassLoader.defineClass(Unknown Source)
        at java.net.URLClassLoader.access$100(Unknown Source)
        at java.net.URLClassLoader$1.run(Unknown Source)
        at java.net.URLClassLoader$1.run(Unknown Source)
```

3. Java final class

A final class cannot be extended(cannot be subclassed), lets take a look into the below example

```
package com.javainterviewpoint;

final class Parent
{
}
public class Child extends Parent
{
```

```
    public static void main(String args[])
    {
       Child c = new Child();
    }
}
```
Output :
We will get the compile time error like *"The type Child cannot subclass the final class Parent"*
Exception in thread "main" java.lang.Error: Unresolved compilation problem

OBJECT CLONING
The **object cloning** is a way to create exact copy of an object. The clone() method of Object class is used to clone an object.

The **java.lang.Cloneable interface** must be implemented by the class whose object clone we want to create. If we don't implement Cloneable interface, clone() method generates **CloneNotSupportedException**.
The **clone() method** is defined in the Object class.

Syntax of the clone() method:
*protected Object clone() **throws** CloneNotSupportedException*

The **clone() method** saves the extra processing task for creating the exact copy of an object. If we perform it by using the new keyword, it will take a lot of processing time to be performed that is why we use object cloning.
Advantage of Object cloning

*   You don't need to write lengthy and repetitive codes. Just use an abstract class with a 4- or 5-line long clone() method.
*   It is the easiest and most efficient way for copying objects, especially if we are applying it to an already developed or an old project. Just define a parent class, implement Cloneable in it, provide the definition of the clone() method and the task will be done.
*   Clone() is the fastest way to copy array.

Disadvantage of Object cloning

*   To use the Object.clone() method, we have to change a lot of syntaxes to our code, like implementing a Cloneable interface, defining the clone() method and handling CloneNotSupportedException, and finally, calling Object.clone() etc.
*   We have to implement cloneable interface while it doesn?t have any methods in it. We just have to use it to tell the JVM that we can perform clone() on our object.
*   Object.clone() is protected, so we have to provide our own clone() and indirectly call Object.clone() from it.
*   Object.clone() doesn?t invoke any constructor so we don?t have any control over object construction.
*   If you want to write a clone method in a child class then all of its superclasses should define the clone() method in them or inherit it from another parent class. Otherwise, the super.clone() chain will fail.

- Object.clone() supports only shallow copying but we will need to override it if we need deep cloning.

Example of clone() method (Object cloning)

```
class Student implements Cloneable{
int rollno;
String name;
Student(int rollno,String
name){ this.rollno=rollno;
this.name=name;
}
public Object clone()throws CloneNotSupportedException{
return super.clone();
}
public static void main(String args[]){
try{
Student s1=new Student(101,"amit");
Student s2=(Student)s1.clone();
System.out.println(s1.rollno+" "+s1.name);
System.out.println(s2.rollno+" "+s2.name);
}
catch(CloneNotSupportedException c){}
}
}
```

Output:
101 amit
101 amit

INNER CLASSES

Inner class means one class which is a member of another class. There are basically four types of inner classes in java.
1) Nested Inner class
2) Method Local inner classes
3) Anonymous inner classes
4) Static nested classes

Nested Inner class

Nested Inner class can access any private instance variable of outer class. Like any other instance variable, we can have access modifier private, protected, public and default modifier. Like class, interface can also be nested and can have access specifiers.

Example:
```
class Outer {
  // Simple nested inner class
  class Inner {
```

```
     public void show() {
         System.out.println("In a nested class method");
     }
   }
}
class Main {
   public static void main(String[] args) {
      Outer.Inner in = new Outer().new Inner();
      in.show();
   }
}
```
Output:
In a nested class method

Method Local inner classes
Inner class can be declared within a method of an outer class. In the following example, Inner is an inner class in outerMethod().
Example:
```
class Outer {
   void outerMethod() {
      System.out.println("inside outerMethod");
      // Inner class is local to outerMethod()
      class Inner {
         void innerMethod() {
            System.out.println("inside innerMethod");
         }
      }
      Inner y = new Inner();
      y.innerMethod();
   }
}
class MethodDemo {
   public static void main(String[] args) {
      Outer x = new Outer();
      x.outerMethod();
   }
}
```
Output:
Inside outerMethod
Inside innerMethod

Static nested classes
Static nested classes are not technically an inner class. They are like a static member of outer

class.
Example:
```
class Outer {
  private static void outerMethod() {
   System.out.println("inside outerMethod");
  }
  // A static inner class
  static class Inner {
   public static void main(String[] args) {
     System.out.println("inside inner class Method");
     outerMethod();
   }
  }
}
```
Output:
inside inner class Method
inside outerMethod

<u>Anonymous inner classes</u>
Anonymous inner classes are declared without any name at all. They are created in two ways.
<u>a) As subclass of specified type</u>
```
class Demo {
  void show() {
    System.out.println("i am in show method of super class");
  }
}
class Flavor1Demo {
  // An anonymous class with Demo as base class
  static Demo d = new Demo() {
    void show() {
      super.show();
      System.out.println("i am in Flavor1Demo class");
    }
  };
    public static void main(String[] args){
    d.show();
  }
}
```
Output:
i am in show method of super class
i am in Flavor1Demo class

In the above code, we have two class Demo and Flavor1Demo. Here demo act as super class and anonymous class acts as a subclass, both classes have a method show(). In anonymous

class show() method is overridden.

b) As implementer of the specified interface
Example:
class Flavor2Demo {
   // An anonymous class that implements Hello interface
   static Hello h = new Hello() {
      public void show() {
         System.out.println("i am in anonymous class");
      }
   };
   public static void main(String[] args) {
   h.show();
   }
}
 interface Hello {
   void show(); }
Output:
i am in anonymous class
        In above code we create an object of anonymous inner class but this anonymous inner class is an implementer of the interface Hello. Any anonymous inner class can implement only one interface at one time. It can either extend a class or implement interface at a time.

STRINGS IN JAVA

In java, string is basically an object that represents sequence of char values**. Java String** provides a lot of concepts that can be performed on a string such as compare, concat, equals, split, length, replace, compareTo, intern, substring etc.

In java, **string objects are immutable**. Immutable simply means unmodifiable or unchangeable.
        String s="javatpoint";
There are two ways to create String object:
   1. By string literal
   2. By new keyword
1) String Literal

Java String literal is created by using double quotes. For Example:
String s="welcome";
2) By new keyword
        String s=new String("Welcome");

	String methods:	
1.	char charAt(int index)	returns char value for the particular index
2.	int length()	returns string length
3.	static String format(String format, Object... args)	returns formatted string
4.	static String format(Locale l, String format, Object... args)	returns formatted string with given locale
5.	String substring(int beginIndex)	returns substring for given begin index
6.	String substring(int beginIndex, int endIndex)	returns substring for given begin index and end index
7.	boolean contains(CharSequence s)	returns true or false after matching the sequence of char value
8.	static String join(CharSequence delimiter, CharSequence... elements)	returns a joined string
9.	static String join(CharSequence delimiter, Iterable<? extends CharSequence> elements)	returns a joined string
10.	boolean equals(Object another)	checks the equality of string with object
11.	boolean isEmpty()	checks if string is empty
12.	String concat(String str)	concatinates specified string
13.	String replace(char old, char new)	replaces all occurrences of specified char value
14.	String replace(CharSequence old, CharSequence new)	replaces all occurrences of specified CharSequence
15.	static String equalsIgnoreCase(String another)	compares another string. It doesn't check case.
16.	String[] split(String regex)	returns splitted string matching regex
17.	String[] split(String regex, int limit)	returns splitted string matching regex and limit
18.	String intern()	returns interned string
19.	int indexOf(int ch)	returns specified char value index
20.	int indexOf(int ch, int fromIndex)	returns specified char value index starting with given index
21.	int indexOf(String substring)	returns specified substring index
22.	int indexOf(String substring, int fromIndex)	returns specified substring index starting with given index
23.	String toLowerCase()	returns string in lowercase.
24.	String toLowerCase(Locale l)	returns string in lowercase using specified locale.
25.	String toUpperCase()	returns string in uppercase.
26.	String toUpperCase(Locale l)	returns string in uppercase using specified locale.
27.	String trim()	removes beginning and ending spaces of this string.
28.	static String valueOf(int value)	converts given type into string. It is overloaded.

Example:

```
public classstringmethod
{
   public static void main(String[] args)
   {
       String string1 = new String("hello");
       String string2 = new String("hello");
       if (string1 == string2)
       {
          System.out.println("string1= "+string1+" string2= "+string2+" are equal");
       }
       else
       {
          System.out.println("string1= "+string1+" string2= "+string2+" are Unequal");
       }
       System.out.println("string1 and string2 is=
       "+string1.equals(string2)); String a="information";
       System.out.println("Uppercase of String a is= "+a.toUpperCase());
       String b="technology";
       System.out.println("Concatenation of object a and b is= "+a.concat(b));
       System.out.println("After concatenation Object a is= "+a.toString());
       System.out.println("\"Joseph\'s\" is the greatest\\ college in chennai");
       System.out.println("Length of Object a is= "+a.length());
       System.out.println("The third character of Object a is= "+a.charAt(2));
       StringBuffer n=new StringBuffer("Technology");
       StringBuffer m=new StringBuffer("Information");
       System.out.println("Reverse of Object n is= "+n.reverse());
       n= new StringBuffer("Technology");
       System.out.println("Concatenation of Object m and n is= "+m.append(n));
       System.out.println("After concatenation of Object m is= "+m);
   }
}
```

Output:

string1= hello string2= hello are Unequal

string1 and string2 is= true

Uppercase of String a is= INFORMATION

Concatenation of object a and b is= informationtechnology

After concatenation Object a is= information

"Joseph's" is the greatest\ college in chennai

Length of Object a is= 11

The third character of Object a is= f

Reverse of Object n is= ygolonhceT

Concatenation of Object m and n is= InformationTechnology

After concatenation of Object m is= InformationTechnology

Java ArrayList class

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 Syntax:

public class ArrayList<E> **extends** AbstractList<E> **implements** List<E>, RandomAccess, Clone able, 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.

Java ArrayList Example: Book
Example:
```java
import java.util.*;
class Book {
int id;
String name,author,publisher;
int quantity;
public Book(int id, String name, String author, String publisher, int quantity) {
   this.id = id;
   this.name = name;
   this.author = author;
   this.publisher = publisher;
   this.quantity = quantity;
}
}
  public class ArrayListExample {
  public static void main(String[] args) {
   //Creating list of Books
   List<Book> list=new ArrayList<Book>();
   //Creating Books
   Book b1=new Book(101,"Let us C","Yashwant Kanetkar","BPB",8);
   Book b2=new Book(102,"Data Communications & Networking","Forouzan","Mc Graw Hill",4);
   Book b3=new Book(103,"Operating System","Galvin","Wiley",6);
   //Adding Books to list
   list.add(b1);
   list.add(b2);
   list.add(b3);
   //Traversing list
   for(Book b:list){
      System.out.println(b.id+" "+b.name+" "+b.author+" "+b.publisher+" "+b.quantity);
   }
  }
}
```
Output:
101 Let us C Yashwant Kanetkar BPB 8

102 Data Communications & Networking Forouzan Mc Graw Hill 4
103 Operating System Galvin Wiley 6

Unit – III

Exception Handling and I/O

Exceptions-exception hierarchy-throwing and catching exceptions-built-in exceptions, creating own exceptions, Stack Trace Elements. Input /Output Basics-Streams-Byte streams and character streams-Reading and Writing Console-Reading and Writing Files Templates

Difference between error and exception

Errors indicate serious problems and abnormal conditions that most applicationsshould not try to handle. Error defines problems that are not expected to be caught under normal circumstances by our program. For example memory error, hardware error, JVM error etc.

Exceptions are conditions within the code. A developer can handle such conditionsand take necessary corrective actions. Few examples

- DivideByZero exception
- NullPointerException
- ArithmeticException
- ArrayIndexOutOfBoundsException

- An exception (or exceptional event) is a problem that arises during the execution of a program.
- When an Exception occurs the normal flow of the program is disrupted and the program/Application terminates abnormally, which is not recommended, therefore, these exceptions are to be handled.
- If an exception is raised, which has not been handled by programmer then program execution can get terminated and system prints a non user friendly error message.

Ex: Exception in thread "main"

java.lang.ArithmeticException: / by zero at

ExceptionDemo.main(ExceptionDemo.java:5)

Where,   ExceptionDemo : The class name

main : The method name

ExceptionDemo.java : The filename

java:5 : Line number

An exception can occur for many different reasons. Following are some scenarios where an exception occurs.

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

Exception Hierarchy

All exception classes are subtypes of the java.lang.Exception class. The exception class is a subclass of the Throwable class.
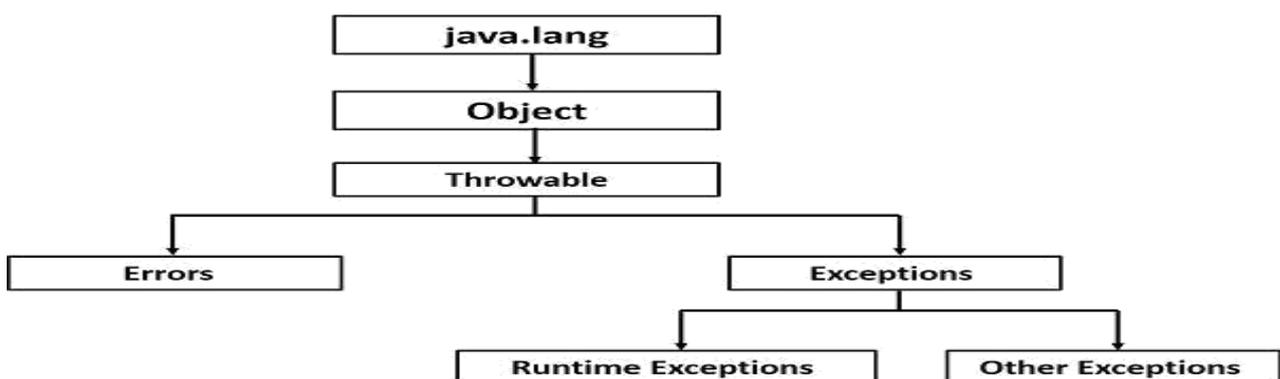
Key words used in Exception handling

There are 5 keywords used in java exception handling.

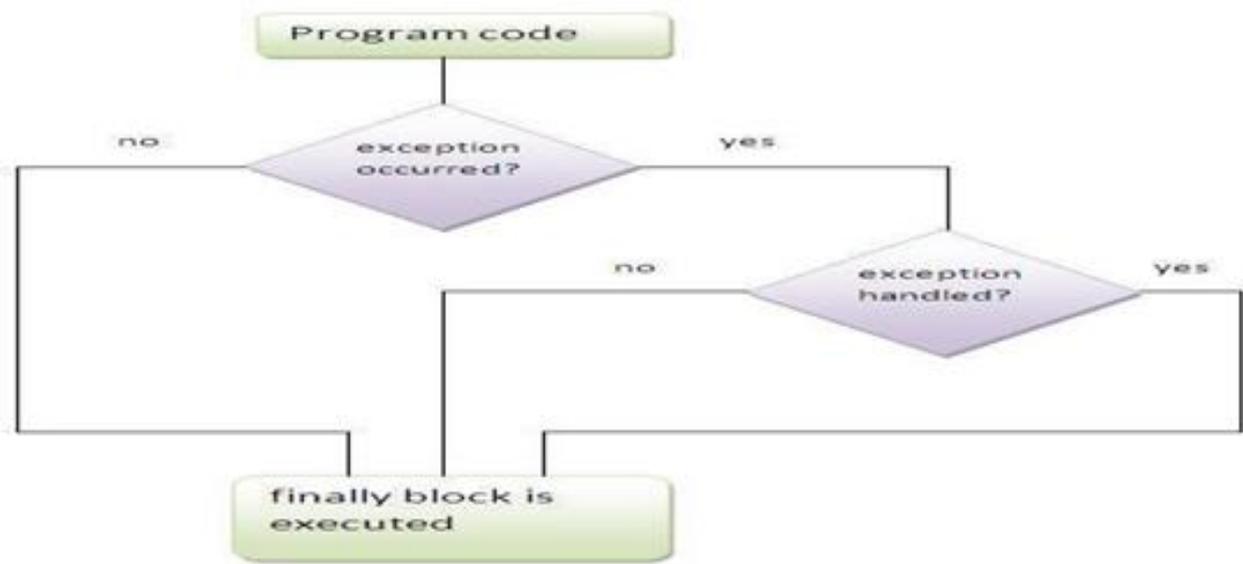|	
1. try	A try/catch block is placed around the code that might generate an exception. Code within a try/catch block is referred to as protected code.
2. catch	A catch statement involves declaring the type of exception we are trying to catch.
3. finally	A finally block of code always executes, irrespective of occurrence of an Exception.
4. throw	It is used to execute important code such as closing connection, stream etc. throw is used to invoke an exception explicitly.
5. throws	throws is used to postpone the handling of a checked exception.

Syntax :	//Example-predefined Exection - for
try	//ArrayindexoutofBounds Exception
{	public class ExcepTest
//Protected code	{
}	public static void main(String args[])
	{ int a[] = new int[2];
catch(ExceptionType1 e1)	**try**
	{ System.out.println("Access element three :" +
{	a[3]);
//Catch block	}
}	**catch**(**ArrayIndexOutOfBoundsException** e)
catch(ExceptionType2 e2)	{ System.out.println("Exception thrown :" + e);
{	}
//Catch block	**finally**
}	{ a[0] = 6;

catch(ExceptionType3 e3)
 	
{
 	
  //Catch block
 	
}
finally
 	
{
 	
//The finally block always executes.
 	
}
	System.out.println("First element value: " + a[0]);
	
	  System.out.println("The finally statement is executed");
	
	  }
	
	  }
	
	}
	Output
	Exception thrown :java.lang.ArrayIndexOutOfBoundsException:3
	First element value: 6
	The finally statement is executed
	Note : *here array size is 2 but we are trying to access 3rdelement.*

Uncaught Exceptions

This small program includes an expression that intentionally causes a divide-by-zero error: class Exc0 { public static void main(String args[]) { int d = 0; int a =

42 / d; } } When the Java run-time system detects the attempt to divide by zero, it constructs a new exception object and then throws this exception. This causes the execution of Exc0 to stop, because once an exception has been thrown, it must be caught by an exception handler and dealt with immediately

Any exception that is not caught by your program will ultimately be processed by the default handler. The default handler displays a string describing the exception, prints a stack trace from the point at which the exception occurred, and terminates the program. Here is the exception generated when this example is executed:

java.lang.ArithmeticException: / by zero at Exc0.main(Exc0.java:4)

Stack Trace:

Stack Trace is a list of method calls from the point when the application was started to the point where the exception was thrown. The most recent method calls are at the top. A stacktrace is a very helpful debugging tool. It is a list of the method calls that the application was in the middle of when an Exception was thrown. This is very useful because it doesn't only show you where the error happened, but also how the program ended up in that place of the code.

Using try and Catch

To guard against and handle a run-time error, simply enclose the code that you want to monitor inside a try block. Immediately following the try block, include a catch clause that specifies the exception type that you wish to catch. A try and its catch statement form a unit. The the following program includes a try block and a catch clause that processes the ArithmeticException generated by the division-by-zero error:

```
class Exc2 {
public static void main(String args[]) {
int d, a;
try { // monitor a block of code.
d = 0;
a = 42 / d;
System.out.println("This will not be printed.");
} catch (ArithmeticException e) { // catch divide-by-zero error
System.out.println("Division by zero.");
}
System.out.println("After catch statement.");
}
}
```

This program generates the following output:

Division by zero.

After catch statement.

The call to println( ) inside the try block is never executed. Once an exception is thrown, program control transfers out of the try block into the catch block.

Multiple catch Clauses

In some cases, more than one exception could be raised by a single piece of code. To handle this type of situation, you can specify two or more catch clauses, each catching a 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:

```
// Demonstrate multiple catch statements.
```

```java
class MultiCatch {
public static void main(String args[]) {
try {
int a = args.length;
System.out.println("a = " + a);
int b = 42 / a;
int c[] = { 1 };
c[42] = 99;
} 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 running it both ways:

```
C:\>java MultiCatch
a = 0
Divide by 0: java.lang.ArithmeticException: / by zero
After try/catch blocks.
C:\>java MultiCatch TestArg
a = 1
Array index oob: java.lang.ArrayIndexOutOfBoundsException:42
After try/catch blocks.
```

Nested try Statements

The try statement can be nested. That is, a try statement can be inside the block of another try. Each time a try statement is entered, the context of that exception is pushed on the stack. If an inner try statement does not have a catch handler for a particular exception, the stack is unwound and the next try statement's catch handlers are inspected for a match. This continues until one of the catch statements succeeds, or until all of the nested try statements are exhausted. If no catch statement matches, then the Java run-time system will handle the exception.

```java
// An example of nested try statements.
class NestTry {
public static void main(String args[]) {
try {
int a = args.length;
/* If no command-line args are present,
the following statement will generate
a divide-by-zero exception. */
int b = 42 / a;
System.out.println("a = " + a);
try { // nested try block
/* If one command-line arg is used,
then a divide-by-zero exception
will be generated by the following code. */
if(a==1) a = a/(a-a); // division by zero
/* If two command-line args are used,
then generate an out-of-bounds exception. */
if(a==2) {
int c[] = { 1 };
```

```
            c[42] = 99; // generate an out-of-bounds exception
        }
    } catch(ArrayIndexOutOfBoundsException e) {
System.out.println("Array index out-of-bounds: " + e);
    }
    } catch(ArithmeticException e) {
System.out.println("Divide by 0: " + e);
        }
    }
}
```

```
C:\>java NestTry
Divide by 0: java.lang.ArithmeticException: / by zero
C:\>java NestTry One
a = 1
Divide by 0: java.lang.ArithmeticException: / by zero
C:\>java NestTry One Two
a = 2
Array index out-of-bounds:
java.lang.ArrayIndexOutOfBoundsException:42
```

throw

it is possible for your program to throw an exception explicitly, using the throw statement. The general form of throw is shown here:

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 you can obtain a Throwable object: using a parameter in a catch clause, or creating one with the new operator.

The flow of execution stops immediately after the throw statement; any subsequent statements 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

```java
// Demonstrate throw.
class ThrowDemo {
static void demoproc() {
try {
throw new NullPointerException("demo");
} catch(NullPointerException e) {
System.out.println("Caught inside demoproc.");
throw e; // rethrow the exception
}
}
public static void main(String args[]) {
try {
demoproc();
} catch(NullPointerException e) {
System.out.println("Recaught: " + e);
}
}
}
```

Here is the resulting output:

Caught inside demoproc.

Recaught: java.lang.NullPointerException: demo

Throws

If a method is capable of causing an exception that it does not handle, it must specify this behaviour so that callers of the method can guard themselves against that exception. You do this by including a throws clause in the method's declaration. A throws clause lists the types of exceptions that a method might throw. This is necessary for all exceptions, except those of type Error or RuntimeException, or any of their subclasses. All other exceptions that a method can throw must be declared in the throws clause. This is the general form of a method declaration that includes a throws clause:

type method-name(parameter-list) throws exception-list

{

// body of method

}

Here, exception-list is a comma-separated list of the exceptions that a method can throw.

class ThrowsDemo {

static void throwOne() throws IllegalAccessException {

System.out.println("Inside throwOne.");

throw new IllegalAccessException("demo");

}

public static void main(String args[]) {

try {

throwOne();

} catch (IllegalAccessException e) {

System.out.println("Caught " + e);

}

}

}

Here is the output generated by running this example program:

inside throwOne

caught java.lang.IllegalAccessException: demo

finally

The finally keyword is designed to address this contingency. finally creates a block of code that will be executed after a try/catch block has completed and before the code following the try/catch block. The finally block will execute whether or not an exception is thrown. If an exception is thrown, the finally block will execute even if no catch statement matches the exception. Any time a method is about to return to the caller from inside a try/catch block, via an uncaught exception or an explicit return statement, the finally clause is also executed just before the method returns. This can be useful for closing file handles and freeing up any other resources that might have been allocated at the beginning of a method with the intent of disposing of them before returning. The finally clause is optional.

```
// Demonstrate finally.
class FinallyDemo {
// Through an exception out of the method.
static void procA() {
try {
System.out.println("inside procA");
throw new RuntimeException("demo");
} finally {
```

```java
System.out.println("procA's finally");

}

}

// Return from within a try block.

static void procB() {

try {

System.out.println("inside procB");

return;

} finally {

System.out.println("procB's finally");

}

}

// Execute a try block normally.

static void procC() {

try {

System.out.println("inside procC");

} finally {

System.out.println("procC's finally");

}

}

public static void main(String args[]) {

try {

procA();

} catch (Exception e) {

System.out.println("Exception caught");

}

procB();

procC();

}
```

}

Here is the output generated by the preceding program:

inside procA

procA's finally

Exception caught

inside procB

procB's finally

inside procC

procC's finally

Categories of Exceptions

Checked exceptions −A checked exception is an exception that occurs at the compiletime, these are also called as compile time exceptions. These exceptions cannot simply be ignored at the time of compilation, the programmer should take care of (handle) these exceptions.

Unchecked exceptions − An unchecked exception is an exception that occurs at thetime of execution. These are 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.

Common scenarios where exceptions may occur:

There are given some scenarios where unchecked exceptions can occur. They are as follows:

1) Scenario where ArithmeticException occurs

If we divide any number by zero, there occurs an ArithmeticException.

int a=50/0;//ArithmeticException

2) Scenario where NullPointerException occurs

If we have null value in any variable, performing any operation by the variable occurs an NullPointerException.

String s=null;

System.out.println(s.length());//NullPointerException

3) Scenario where ArrayIndexOutOfBoundsException occurs

If you are inserting any value in the wrong index, it would result ArrayIndexOutOfBoundsException as shown below:

int a[]=new int[5];

a[10]=50; //ArrayIndexOutOfBoundsException

Java's Built-in Exceptions

User-defined Exceptions

All exceptions must be a child of **Throwable.**

If we want to write a checked exception that is automatically enforced by the Handle ,we need to extend the Exception class.

User defined exception needs to inherit (extends) Exception class in order to act as an exception.

throw keyword is used to throw such exceptions.

class MyOwnException extends Exception

```
{                         public
    MyOwnException(String
    msg) { super(msg);
    }
}
class EmployeeTest
{
```

```
    static   void   employeeAge(int   age)   throws
     MyOwnException {
        if(age < 0)
          throw new MyOwnException("Age can't be less
      than zero"); else
          System.out.println("Input is valid!!");
     }
public static void main(String[] args)
{
     try { employeeAge(-2);
        }
     catch (MyOwnException e)
     {
         e.printStackTrace();
       }
}
}
```

Advantages of Exception Handling

Exception handling allows us to control the normal flow of the program by using exception handling in program.

It throws an exception whenever a calling method encounters an error providing that the calling method takes care of that error.

It also gives us the scope of organizing and differentiating between different error types using a separate block of codes. This is done with the help of try-catch blocks.

IO IN JAVA

Java I/O (Input and Output) is used to process the input and produce the output based on the input. Java uses the concept of stream to make I/O operation fast. The java.io package contains all the classes required for input and output operations.

Stream

A stream can be defined as a sequence of data. there are two kinds of Streams

- **InputStream:** The InputStream is used to read data from a source.
- **OutputStream:** the OutputStream is used for writing data to a destination.

Byte Streams

Java byte streams are used to perform input and output of 8-bit bytes FileInputStream , FileOutputStream.

Character Streams

Java Character streams are used to perform input and output for 16-bit unicode. FileReader , FileWriter

Standard Streams

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

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

Classification of Stream Classes:
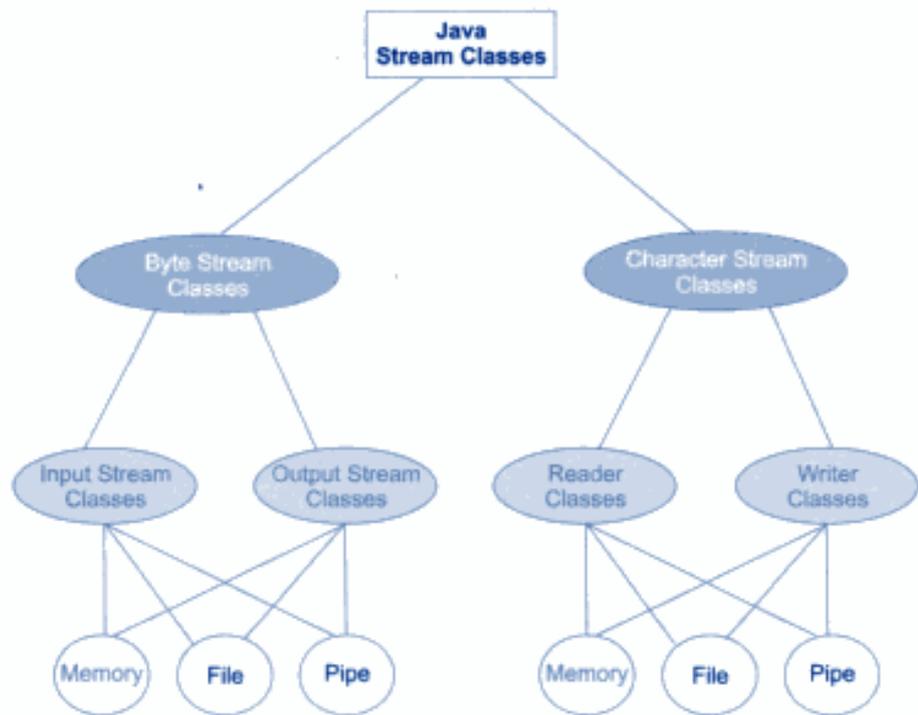
Byte Stream Classes:

ByteStream classes have been designed to provide functional features for creating and manipulating streams and files for reading and writing bytes. Since the streams are unidirectional, they can transmit bytes in only one direction and therefore, Java provides two kinds of byte stream classes: InputStream class and OutputStream class.

Input Stream Classes

Input stream classes that are used to read 8-bit bytes include a super class known as InputStream and number of subclasses for supporting various input-related functions**.**
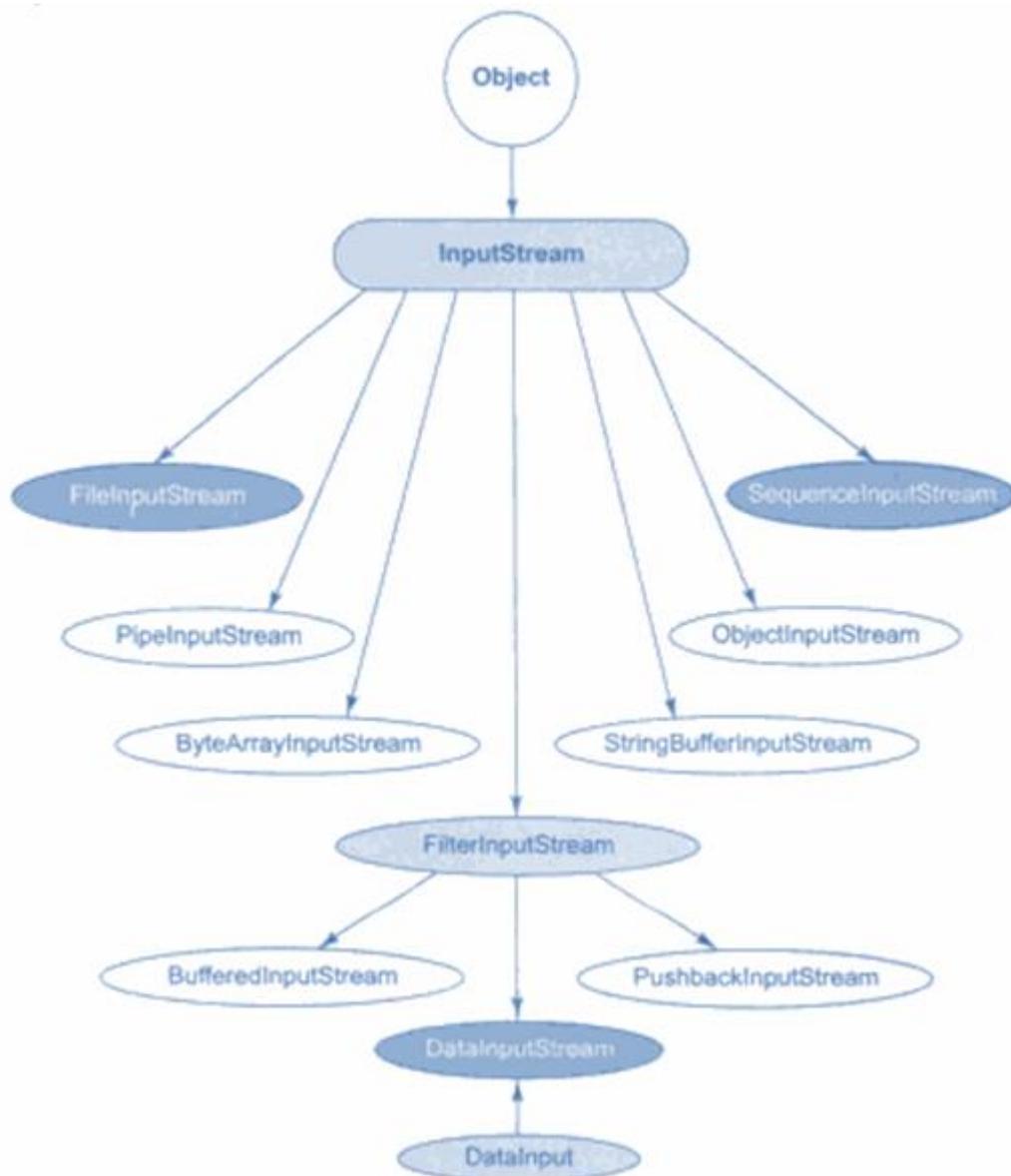
Hierarchy of Input Stream Classes

The super class InputStream is an abstract class, so we cannot create object for the class. InputStream class defines the methods to perform the following functions:-

- Reading Bytes
- Closing Streams
- Marking position in Streams
- Skipping ahead in streams

- Finding the number of bytes in stream.

The following are the InputStream methods:

Method	Description
1. read( )	Reads a byte from the input stream
2. read (byte b[ ])	Reads an array of bytes into b
3. read (byte b[ ], int n, int m)	Reads m bytes into b starting from nth byte.
4. available( )	Gives number of bytes available in the input
5. skip(n)	Skips over n bytes from the input stream
6. reset( )	Goes back to the beginning of the stream
7. close( )	Closes the input stream

The DataInput interface contains the following methods

- readShort( )
- readInt( )
- readLong( )
- readFloat( )
- readUTF( )

- readDouble( )
- readLine( )
- readChar( )
- readBollean( )

OutputStream Class

The super class InputStream is an abstract class, so we cannot create object for the class. InputStream class defines the methods to perform the following functions:

- Writing Bytes
- Closing Streams
- Flushing Streams

Hierarchy of OutputStream Classes

OutputStream Methods

Method	Description
1. write( )	Writes a byte to the output stream
2. write(byte[ ] b)	Writes all bytes in the array b to the output stream
3. write(byte b[ ], int n, int m)	Writes m bytes from array b starting from nth byte
4. close( )	Closes the output stream
5. flush( )	Flushes the output stream

Character Stream Vs Byte Stream in Java

I/O Stream

A stream is a method to sequentially access a file. I/O Stream means an input source or output destination representing different types of sources e.g. disk files.The java.io package provides classes that allow you to convert between Unicode character streams and byte streams of non-Unicode text.

Stream: A sequence of data.

Input Stream: reads data from source.

Output Stream: writes data to destination.

Character Stream

In Java, characters are stored using Unicode conventions (Refer this for details). Character stream automatically allows us to read/write data character by character. For example FileReader and FileWriter are character streams used to read from source andwrite to destination.

```
Non-Unicode bytes
       |
       v
+------------------+
|   InputStream    |
+------------------+
       |
       v
Unicode character Streams
       |
       v
+------------------+
|   OutputStream   |
+------------------+
       |
       v
Non-Unicode bytes
```
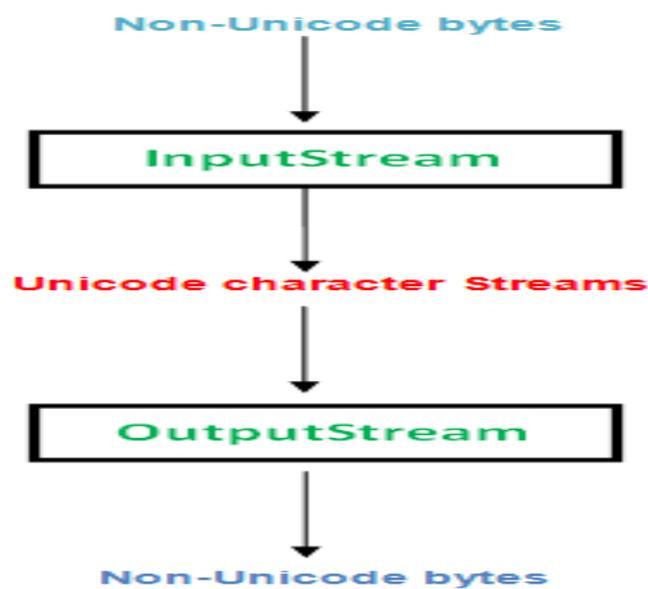
```java
// Java Program illustrating that we can read a file in
// a human readable format using FileReader
importjava.io.*;   // Accessing FileReader, FileWriter, IOException
publicclassGfG
{
    publicstaticvoidmain(String[] args) throwsIOException
    {
        FileReader sourceStream = null;
        try
        {
            sourceStream = newFileReader("test.txt");
```

```
            // Reading sourcefile and writing content to

            // target file character by character.

            inttemp;

            while((temp = sourceStream.read()) != -1)

                System.out.println((char)temp);

        }

        finally

        {

            // Closing stream as no longer in use

            if(sourceStream != null)

                sourceStream.close();

        }

    }

}
```

Reading and Writing Files:

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

TheInputStream is used to read data from a source and the OutputStream is used for writing data to a destination.The two important streams are **FileInputStream** and **FileOutputStream**

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

FileInputStream

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

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

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

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

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

InputStream f = new FileInputStream(f);

Once you have *InputStream* object in hand, then there is a list of helper methods which can be used to read to stream or to do other operations on the stream.

Example:

```java
import
java.io.*;
class C{
public    static    void    main(String    args[])throws
Exception{           FileInputStream           fin=new
FileInputStream("C.java");           FileOutputStream
fout=new FileOutputStream("M.java"); int i=0;
while((i=fin.read())!=-
1){ fout.write((byte)i);
}
fin.close();
}
}
```

Byte Stream

Byte streams process data byte by byte (8 bits). For example FileInputStream is used to read from source and FileOutputStream to write to the destination.

// Java Program illustrating the Byte Stream to copy

// contents of one file to another file.

```java
importjava.io.*;
publicclassBStream
{
   Public static void main(String[] args) throws IOException
   {
      FileInputStream sourceStream = null;
      FileOutputStream targetStream = null;
       try
      {
         sourceStream = newFileInputStream("sorcefile.txt");
         targetStream = newFileOutputStream ("targetfile.txt");


         // Reading source file and writing content to target
         // file byte by byte
         inttemp;
         while((temp = sourceStream.read()) != -1)
            targetStream.write((byte)temp);
      }
      finally
      {
         if(sourceStream != null)
            sourceStream.close();
         if(targetStream != null)
            targetStream.close();
      }
   }
}
```

Final Keyword In Java – Final variable, Method and Class

final keyword can be used along with variables, methods and classes.

1) final variable

2) final method

3) final class

1) final variable

final variables are nothing but constants. We cannot change the value of a final variable once it is initialized. Lets have a look at the below code:

```
classDemo{
finalint MAX_VALUE=99;
void myMethod(){
    MAX_VALUE=101;
}
Public static void main(String args[]){
Demo obj=newDemo();
    obj.myMethod();
}
}
```

Exception in thread "main" java.lang.Error: Unresolved compilation problem:
        The final field Demo.MAX_VALUE cannot be assigned
        at beginnersbook.com.Demo.myMethod(Details.java:6)
        at beginnersbook.com.Demo.main(Details.java:10)

We got a compilation error in the above program because we tried to change the value of a final variable "MAX_VALUE".

2) final method

A final method cannot be overridden. Which means even though a sub class can call the final method of parent class without any issues but it cannot override it.

Example:

```
class XYZ{
```

```
finalvoid demo(){

System.out.println("XYZ Class Method");

}

}

class ABC extends XYZ{

void demo(){

System.out.println("ABC Class Method");

}

public static void main(String args[]){

    ABC obj=new ABC();

    obj.demo();

}

}
```

The above program would throw a compilation error, however we can use the parent class final method in sub class without any issues. Lets have a look at this code: This program would run fine as we are not overriding the final method. That shows that final methods are inherited but they are not eligible for overriding.

```
class XYZ{

finalvoid demo(){

System.out.println("XYZ Class Method");

}

}

class ABC extends XYZ{

public static void main(String args[]){

    ABC obj=new ABC();

    obj.demo();

}

}
```

Output:

XYZ ClassMethod

3) final class

We cannot extend a final class. Consider the below example:

finalclass XYZ{

}

class ABC extends XYZ{

void demo(){

System.out.println("My Method");

}

Public static void main(String args[]){

    ABC obj=new ABC();

    obj.demo();

}

}

Output:

The type ABC cannot subclass the final class XYZ

UNIT IV
MULTITHREADING AND GENERIC PROGRAMMING

Differences between multithreading and multitasking , thread life cycle, creating threads, creating threads, synchronizing threads, Inter-thread communication, daemon threads, thread groups. Generic Programming - Generic classes-generic methods-Bounded Types-Restrictions and Limitations

Thread:

A thread is a single sequential (separate) flow of control within program. Sometimes, it is called an execution context or light weight process.

Multithreading

Multithreading is a conceptual programming concept where a program (process) is divided into two or more subprograms (process), which can be implemented at the same time in parallel. A multithreaded program contains two or more parts that can run concurrently. Each part of such a program is called a thread, and each thread defines a separate path of execution.

Multitasking

Executing several tasks simultaneously is called multi-tasking.

There are 2 types of multi-tasking

1. Process-based multitasking

2. Thread-based multi-tasking

1. Process-based multi-tasking

Executing various jobs together where each job is a separate independent operation is called process-based multi-tasking.

2. Thread-based multi-tasking

Executing several tasks simultaneously where each task is a separate independent part of the same program is called Thread-based multitasking and each independent part is called Thread. It is best suitable for the programmatic

level. The main goal of multi-tasking is to make or do a better performance of the system by reducing response time

Multithreading vs Multitasking	
Multithreading is to execute multiple threads in a process concurrently.	Multitasking is to run multiple processes on a computer concurrently.
Execution	
In Multithreading, the CPU switches between multiple threads in the same process.	In Multitasking, the CPU switches between multiple processes to complete the execution.
Resource Sharing	
In Multithreading, resources are shared among multiple threads in a process.	In Multitasking, resources are shared among multiple processes.
Complexity	
Multithreading is light-weight and easy to create.	Multitasking is heavy-weight and harder to create.

Life Cycle of Thread

A thread can be in any of the five following states

1.Newborn State:

When a thread object is created a new thread is born and said to be in Newborn state.

2.Runnable State:

If a thread is in this state it means that the thread is ready for execution and waiting for the availability of the processor. If all threads in queue are of same priority then they are given time slots for execution in round robin fashion

3.Running State:

It means that the processor has given its time to the thread for execution. A thread keeps running until the following conditions occurs

(a) Thread give up its control on its own and it can happen in the following situations

> i.A thread gets suspended using suspend() method which can only be revived with resume() method

> ii.A thread is made to sleep for a specified period of time using sleep(time) method, where time in milliseconds

> iii.A thread is made to wait for some event to occur using wait () method. In this case a thread can be scheduled to run again using notify () method.
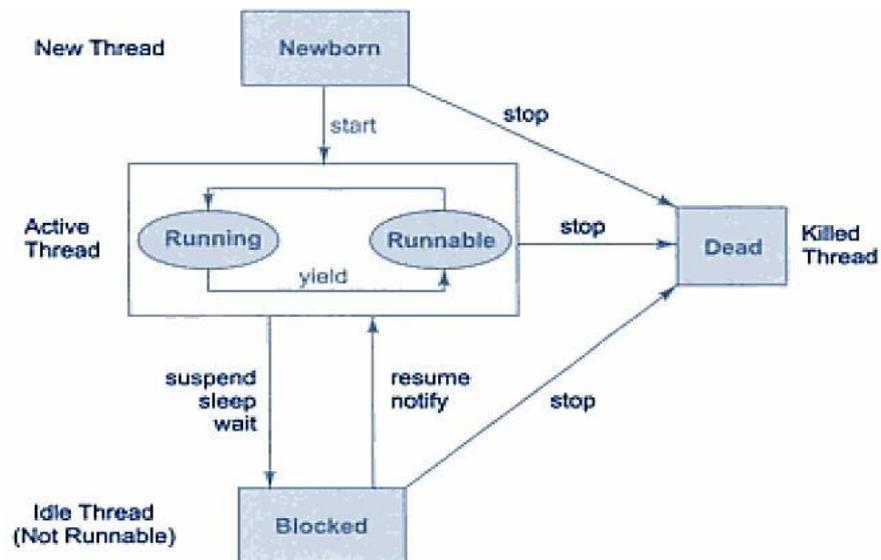
(b) A thread is pre-empted by a higher priority thread

4. Blocked State:

If a thread is prevented from entering into runnable state and subsequently running state, then a thread is said to be in Blocked state.

5. Dead State:

A runnable thread enters the Dead or terminated state when it completes its task or otherwise

The Main Thread

When we run any java program, the program begins to execute its code starting from the main method. Therefore, the JVM creates a thread to start executing the code present in main method. This thread is called as main thread. Although the main thread is automatically created, you can control it by obtaining a reference to it by calling currentThread() method.

Two important things to know about main thread are,

- It is the thread from which other threads will be produced.
- main thread must be always the last thread to finish execution.

```java
class MainThread
{
 public static void main(String[] args)
 {
  Thread t1=Thread.currentThread();
  t.setName("MainThread");
  System.out.println("Name of thread is "+t1);
 }
}
```

Output:

Name of thread is Thread[MainThread,5,main]

Creation Of Thread

Thread Can Be Implemented In Two Ways

1) **Implementing Runnable Interface**
2) **Extending Thread Class**

1.Create Thread by Implementing Runnable

The easiest way to create a thread is to create a class that implements the Runnable interface. To implement Runnable, a class need only implement a single method called run()

Example:

```java
public class ThreadSample implements Runnable
{
        public void run()
        {
                try
                {
                        for (int i = 5; i > 0; i--)
                        {
                                System.out.println("Child Thread" + i);
                                Thread.sleep(1000);
                        }
                }
                catch (InterruptedException e)
                {
                        System.out.println("Child interrupted");
                }
```

```java
            System.out.println("Exiting Child Thread");
        }
    }
    public class MainThread
    {
        public static void main(String[] arg)
        {
            ThreadSample d = new ThreadSample();
            Thread s = new Thread(d);
            s.start();
            try
            {
                for (int i = 5; i > 0; i--)
                {
                    System.out.println("Main Thread" + i);
                    Thread.sleep(5000);
                }
            }
            catch (InterruptedException e)
            {
                System.out.println("Main interrupted");
            }
            System.out.println("Exiting Main Thread");
        }}
```

2. Extending Thread Class

The second 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.

Example:

```java
public class ThreadSample extends Thread
{
    public void run()
    {
        try
        {
            for (int i = 5; i > 0; i--)
            {
                System.out.println("Child Thread" + i);
                Thread.sleep(1000);
            }
        }
        catch (InterruptedException e)
        {
            System.out.println("Child interrupted");
        }
        System.out.println("Exiting Child Thread");
    }
}
public class MainThread
{
    public static void main(String[] arg)
    {
        ThreadSample d = new ThreadSample();
        d.start();
        try
        {
```

```java
                for (int i = 5; i > 0; i--)
                {
                        System.out.println("Main Thread" + i);
                        Thread.sleep(5000);
                }
        }
        catch (InterruptedException e)
        {
                System.out.println("Main interrupted");
        }
        System.out.println("Exiting Main Thread");
    }
}
```

Thread priority:

Each thread have 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:

     public static int MIN_PRIORITY

     public static int NORM_PRIORITY

     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.

Example :

```java
public class MyThread1 extends Thread {
        MyThread1(String s)
        {
```

```java
            super(s);
            start();
        }
    public void run()
    {
    for(int i=0;i<5;i++)
        {
            Thread cur=Thread.currentThread();
            cur.setPriority(Thread.MAX_PRIORITY);
            int p=cur.getPriority();
    System.out.println("Thread
Name"+Thread.currentThread().getName());
            System.out.println("Thread Priority"+cur);
        }
        }
    }
class MyThread2 extends Thread {
    MyThread2(String s)
    {
            super(s);
            start();
    }
    public void run()
    {
    for(int i=0;i<5;i++)
        {
            Thread cur=Thread.currentThread();
            cur.setPriority(Thread.MIN_PRIORITY);
            System.out.println(cur.getPriority());
```

```java
            int p=cur.getPriority();
        System.out.println("Thread
Name"+Thread.currentThread().getName());
            System.out.println("Thread Priority"+cur);
        }
    }
}
public class ThreadPriority {
    public static void main(String[] args)
    {
            MyThread1 m1=new MyThread1("MyThread1");
            MyThread2 m2=new MyThread2("MyThread2");
    }
    }
```

Synchronizing Threads

- o Synchronization in java is the capability to control the access of multiple threads to any shared resource.
- o Java Synchronization is better option where we want to allow only one thread to access the shared resource

General Syntax :

```java
synchronized(object)
{
//statement to be synchronized
}
```

Why use Synchronization

The synchronization is mainly used to

- o To prevent thread interference.
- o To prevent consistency problem.

Types of Synchronization

There are two types of synchronization

- Process Synchronization
- Thread Synchronization

Thread Synchronization

There are two types of thread synchronization mutual exclusive and inter-thread communication.

1. Mutual Exclusive

- Synchronized method.
- Synchronized block.
- static synchronization.

2.Cooperation (Inter-thread communication in java)

Synchronized method

- If you declare any method as synchronized, it is known as synchronized method.
- Synchronized method is used to lock an object for any shared resource.
- When a thread invokes a synchronized method, it automatically acquires the lock for that object and releases it when the thread completes its task.

Example of synchronized method

```
package Thread;
public class SynThread
{
        public static void main(String args[])
        {
                share s = new share();
                MyThread m1 = new MyThread(s, "Thread1");
                MyThread m2 = new MyThread(s, "Thread2");
```

```java
                MyThread m3 = new MyThread(s, "Thread3");

        }
}
class MyThread extends Thread
{
        share s;
        MyThread(share s, String str)
        {
                super(str);
                this.s = s;
                start();
        }
        public void run()
        {
                s.doword(Thread.currentThread().getName());
        }
}
class share
{
        public synchronized void doword(String str)
        {
                for (int i = 0; i < 5; i++)
                {
                        System.out.println("Started:" + str);
                        try
                        {
                                Thread.sleep(1000);
                        }
                        catch (Exception e)
```

```
                    {
                    }}}}
```

Synchronized block.

- Synchronized block can be used to perform synchronization on any specific resource of the method.

- Suppose you have 50 lines of code in your method, but you want to synchronize only 5 lines, you can use synchronized block.

- If you put all the codes of the method in the synchronized block, it will work same as the synchronized method.

Example of synchronized block

```
class Table{
 void printTable(int n){
  synchronized(this){//synchronized block
   for(int i=1;i<=5;i++){
    System.out.println(n*i);
    try{
     Thread.sleep(400);
    }catch(Exception e){System.out.println(e);}
   }
  }
 }//end of the method
}
 class MyThread1 extends Thread{
Table t;
MyThread1(Table t){
this.t=t;
}
public void run(){
t.printTable(5);
```

```
}
}
class MyThread2 extends Thread{
Table t;
MyThread2(Table t){
this.t=t;
}
public void run(){
t.printTable(100);
}
}
public class TestSynchronizedBlock1{
public static void main(String args[]){
Table obj = new Table();//only one object
MyThread1 t1=new MyThread1(obj);
MyThread2 t2=new MyThread2(obj);
t1.start();
t2.start();
}
}
```

Static synchronization

If you make any static method as synchronized, the lock will be on the class not on object.

Example of static synchronization

In this example we are applying synchronized keyword on the static method to perform static synchronization.

```
class Table{
 synchronized static void printTable(int n){
   for(int i=1;i<=10;i++){
```

```java
    System.out.println(n*i);
    try{
     Thread.sleep(400);
    }catch(Exception e){}
   }
 }
}
class MyThread1 extends Thread{
public void run(){
Table.printTable(1);
}
}
class MyThread2 extends Thread{
public void run(){
Table.printTable(10);
}
}
class MyThread3 extends Thread{
public void run(){
Table.printTable(100);
}
}
class MyThread4 extends Thread{
public void run(){
Table.printTable(1000);
}
}
public class TestSynchronization4{
public static void main(String t[]){
```

```
MyThread1 t1=new MyThread1();

MyThread2 t2=new MyThread2();

MyThread3 t3=new MyThread3();

MyThread4 t4=new MyThread4();

t1.start();

t2.start();

t3.start();

t4.start();

}

}
```

Inter-thread communication

Inter-thread communication or Co-operation is all about allowing synchronized threads to communicate with each other.

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()

wait()

tells calling thread to give up monitor and go to sleep until some other thread enters the same monitor and call notify.

notify()

wakes up a thread that called wait() on same object.

notifyAll()

wakes up all the thread that called wait() on same object.

wait()	sleep()
wait() method releases the lock	sleep() method doesn't release the lock.
is the method of Object class	is the method of Thread class
is the non-static method	is the static method
should be notified by notify() or notifyAll() methods	after the specified amount of time, sleep is completed.

Example of inter thread communication in java

```
class Customer{
int amount=10000;
synchronized void withdraw(int amount){
System.out.println("going to withdraw...");
if(this.amount<amount){
System.out.println("Less balance; waiting for deposit...");
try{wait();}catch(Exception e){}
}
this.amount-=amount;
System.out.println("withdraw completed...");
}
synchronized void deposit(int amount){
System.out.println("going to deposit...");
this.amount+=amount;
System.out.println("deposit completed... ");
notify();
```

```
}
}
class Test{
public static void main(String args[]){
final Customer c=new Customer();
new Thread(){
public void run(){c.withdraw(15000);}
}.start();
new Thread(){
public void run(){c.deposit(10000);}
}.start();
}}
```

Daemon Thread in Java

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.

Example:

```
public class TestDaemonThread1 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();

 }

}

Thread Group

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.

Constructors of ThreadGroup class

There are only two constructors of ThreadGroup class.

1.ThreadGroup(String name)-creates a thread group with given name.

2.ThreadGroup(ThreadGroup parent, String name)-creates a thread group with given parent group and name.

Important methods of ThreadGroup class

There are many methods in ThreadGroup class. A list of important methods are given below.

1)int activeCount()-returns no. of threads running in current group.

2)int activeGroupCount()-returns a no. of active group in this thread group.

3)void destroy()-destroys this thread group and all its sub groups.

4)String getName()-returns the name of this group.

5)ThreadGroup getParent()-returns the parent of this group.

6)void interrupt()-interrupts all threads of this group.

7)void list()-prints information of this group to standard console.

Let's see a code to group multiple threads.

ThreadGroup tg1 = new ThreadGroup("Group A");

Thread t1 = new Thread(tg1,new MyRunnable(),"one");

Thread t2 = new Thread(tg1,new MyRunnable(),"two");

Thread t3 = new Thread(tg1,new MyRunnable(),"three");

Now all 3 threads belong to one group. Here, tg1 is the thread group name, MyRunnable is the class that implements Runnable interface and "one", "two" and "three" are the thread names.

Now we can interrupt all threads by a single line of code only.

Thread.currentThread().getThreadGroup().interrupt();

ThreadGroup Example

```
public class ThreadGroupDemo implements Runnable{
    public void run() {
        System.out.println(Thread.currentThread().getName());
    }
    public static void main(String[] args) {
      ThreadGroupDemo runnable = new ThreadGroupDemo();
        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();

        }

    }
```

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]

Generic Programming

      Generic programming enables the programmer to create classes,interfaces and methods that automatically works with all types of data(Integer, String, Float etc). It has expanded the ability to reuse the code safely and easily.

Advantage of Java Generics

There are mainly 3 advantages of generics. They are as follows:

1)Type-safety : We can hold only a single type of objects in generics. It doesn't allow to store other objects.

2)Type casting is not required: There is no need to typecast the object.

3)Compile-Time Checking: It is checked at compile time so problem will not occur at runtime. The good programming strategy says it is far better to handle the problem at compile time than runtime.

Generic class

- A class that can refer to any type is known as generic class.
- Generic class declaration defines set of parameterized type one for each possible invocation of the type parameters

Example:

```
class TwoGen<T, V>
{
     T ob1;
     V ob2;
     TwoGen(T o1, V o2)
     {
          ob1 = o1;
          ob2 = o2;
     }
     void showTypes() {
          System.out.println("Type of T is " + ob1.getClass().getName());
          System.out.println("Type of V is " + ob2.getClass().getName());
     }
     T getob1()
     {
          return ob1;
     }


     V getob2()
     {
          return ob2;
     }
}
public class MainClass
```

```java
{
        public static void main(String args[])
            {
TwoGen<Integer, String> tgObj = new TwoGen<Integer,
String>(88,"Generics");
            tgObj.showTypes();
            int v = tgObj.getob1();
            System.out.println("value: " + v);
            String str = tgObj.getob2();
            System.out.println("value: " + str);
            }
}
```

Generic Method

Like generic class, we can create generic method that can accept any type of argument.

```java
public class TestGenerics4{
  public static < E > void printArray(E[] elements) {
    for ( E element : elements){
      System.out.println(element );
    }
    System.out.println();
  }
  public static void main( String args[] ) {
    Integer[] intArray = { 10, 20, 30, 40, 50 };
    Character[] charArray = { 'J', 'A', 'V', 'A'};
    System.out.println( "Printing Integer Array" );
    printArray( intArray  );
    System.out.println( "Printing Character Array" );
```

```
    printArray( charArray );
  }
}
```

Bounded type

The type parameters could be replaced by any class type. This is fine for many purposes, but sometimes it is useful to limit the types that can be passed to a type parameter

Syntax :

`<T extends superclass>`

Example

```
class Stats<T extends Number> {
  T[] nums;
   Stats(T[] o) {
   nums = o;   }
   double average() {
    double sum = 0.0;
    for(int i=0; i < nums.length; i++)
      sum += nums[i].doubleValue();
    return sum / nums.length;
  }
}
public class MainClass {
  public static void main(String args[]) {
    Integer inums[] = { 1, 2, 3, 4, 5 };
    Stats<Integer> iob = new Stats<Integer>(inums);
    double v = iob.average();
    System.out.println("iob average is " + v);
    Double dnums[] = { 1.1, 2.2, 3.3, 4.4, 5.5 };
    Stats<Double> dob = new Stats<Double>(dnums);
```

```
    double w = dob.average();

    System.out.println("dob average is " + w);

  }

}
```

Restrictions on Generics

To use Java generics effectively, you must consider the following restrictions:

- Cannot Instantiate Generic Types with Primitive Types

- Cannot Create Instances of Type Parameters

- Cannot Declare Static Fields Whose Types are Type Parameters

- Cannot Use Casts or instanceof With Parameterized Types

- Cannot Create Arrays of Parameterized Types

- Cannot Create, Catch, or Throw Objects of Parameterized Types

- Cannot Overload a Method Where the Formal Parameter Types of Each Overload Erase to the Same Raw Type

EVENT DRIVEN PROGRAMMING

Graphics programming-Frame-Components-working with 2D shapes-Using color, fonts, and images-Basics of event Handling-event handlers-adapter classes-actions mouse events-AWT event hierarchy-Introduction to Swing-layout management-Swing Components-Text Fields, Text Areas-Buttons-Check Boxes-Radio Buttons-Lists-choices-Scrollbars-windows-Menus-Dialog Boxes and Interfaces, Exception handling, Multithreaded programming, Strings, Input/output

Graphics programming

- Java contains support for graphics that enable programmers to visually enhance applications
- Java contains many more sophisticated drawing capabilities as part of the Java 2D API

AWT

- Java AWT (Abstract Window Toolkit) is an API to develop GUI or window-based applications in java.
- Java AWT components are platform-dependent i.e. components are displayed according to the view of operating system.
- AWT is heavyweight i.e. its components are using the resources of OS.The java.awt package provides classes for AWT api such as TextField, Label, TextArea, RadioButton, CheckBox, Choice, List etc.

Java AWT Hierarchy

The hierarchy of Java AWT classes are given below.
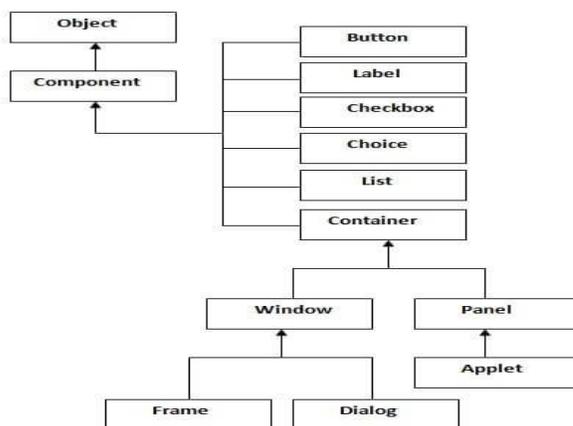
Container

The Container is a component in AWT that can contain another components 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.

There are two ways to create a Frame. They are,

- By Instantiating Frame class
- By extending Frame class

Example:

```
import java.awt.*;
import java.awt.event.*;
class MyLoginWindow extends Frame
{
  TextField name,pass;
  Button b1,b2;
  MyLoginWindow()
  {
    setLayout(new FlowLayout());
    this.setLayout(null);
    Label n=new Label("Name:",Label.CENTER);
    Label p=new Label("password:",Label.CENTER);
    name=new TextField(20);
    pass=new TextField(20);
    pass.setEchoChar('#');
    b1=new Button("submit");
    b2=new Button("cancel");
    this.add(n);
    this.add(name);
    this.add(p);
    this.add(pass);
    this.add(b1);
    this.add(b2);
```

```
    n.setBounds(70,90,90,60);
    p.setBounds(70,130,90,60);
    name.setBounds(200,100,90,20);
    pass.setBounds(200,140,90,20);
    b1.setBounds(100,260,70,40);
    b2.setBounds(180,260,70,40);
  }
  public static void main(String args[])
  {
    MyLoginWindow ml=new MyLoginWindow();
    ml.setVisible(true);
    ml.setSize(400,400);
    ml.setTitle("my login window");
  }}
```

Output:

Event handling:

Changing the state of an object is known as an event. For example, click on button, dragging mouse etc. The java.awt.event package provides many event classes and Listener interfaces for event handling. Event handling has three main components,

- **Events** : An event is a change in state of an object.

- **Events Source** : Event source is an object that generates an event.

- **Listeners** : A listener is an object that listens to the event. A listener gets notified when an event occur

How Events are handled ?

A source generates an Event and send it to one or more listeners registered with the source. Once event is received by the listener, they process the event and then return. Events are supported by a number of Java packages, like **java.util**, **java.awt** and **java.awt.event**.

Important Event Classes and Interface

Event Classes	Description	Listener Interface
ActionEvent	generated when button is pressed, menu-item is selected, list-item is double clicked	ActionListener
MouseEvent	generated when mouse is dragged, moved,clicked,pressed or released and also when it enters or exit a component	MouseListener
KeyEvent	generated when input is received from keyboard	KeyListener
ItemEvent	generated when check-box or list item is clicked	ItemListener
TextEvent	generated when value of textarea or textfield is changed	TextListener
MouseWheelEvent	generated when mouse wheel is moved	MouseWheelListener
WindowEvent	generated when window is activated, deactivated, deiconified, iconified, opened or closed	WindowListener
ComponentEvent	generated when component is hidden, moved, resized or set visible	ComponentEventListener
ContainerEvent	generated when component is added or removed from container	ContainerListener
AdjustmentEvent	generated when scroll bar is manipulated	AdjustmentListener

FocusEvent	generated when component gains or loses keyboard focus	FocusListener

Steps to handle events:

- Implement appropriate interface in the class.
- Register the component with the listener.

How to implement Listener

1. Declare an event handler class and specify that the class either implements an ActionListener(any listener) interface or extends a class that implements an ActionListener interface. For example:

public class MyClass implements ActionListener

{

// Set of Code

}

2. Register an instance of the event handler class as a listener on one or more components. For example:

someComponent.addActionListener(instanceOfMyClass);

3. Include code that implements the methods in listener interface. For example:

public void actionPerformed(ActionEvent e) {

   //code that reacts to the action

}

Mouse Listener

package Listener;

import java.awt.Frame;

import java.awt.Label;

import java.awt.TextArea;

import java.awt.event.MouseEvent;

import java.awt.event.MouseListener;

```java
public class Mouse implements MouseListener {
        TextArea s;
public Mouse()
{
        Frame d=new Frame("kkkk");
        s=new TextArea("");
        d.add(s);
        s.addMouseListener(this);
        d.setSize(190, 190);
    d.show();
  }
  public void mousePressed(MouseEvent e) {
    System.out.println("MousePressed");
    int a=e.getX();
        int b=e.getY();
        System.out.println("X="+a+"Y="+b);
  }
  public void mouseReleased(MouseEvent e) {
        System.out.println("MouseReleased");
        }
  public void mouseEntered(MouseEvent e) {
        System.out.println("MouseEntered");
  }
  public void mouseExited(MouseEvent e) {
        System.out.println("MouseExited");
  }
  public void mouseClicked(MouseEvent e) {
        System.out.println("MouseClicked");
    }
  public static void main(String arg[])
  {
  Mouse a=new Mouse();
```

```
        }

}
```

Mouse Motion Listener

```java
package Listener;
import java.awt.event.MouseEvent;
import java.awt.event.MouseMotionListener;
import javax.swing.JFrame;
import javax.swing.JPanel;
import javax.swing.JTextArea;
public class MouseMotionEventDemo extends JPanel implements MouseMotionListener {
MouseMotionEventDemo()
{
        JTextArea a=new JTextArea();
        a.addMouseMotionListener(this);
        JFrame b=new JFrame();
        b.add(a);
        b.setVisible(true);
}
public void mouseMoved(MouseEvent e) {
System.out.println("Mouse is Moving");
}
public void mouseDragged(MouseEvent e) {
        System.out.println("MouseDragged");
}
public static void main(String arg[])
{
        MouseMotionEventDemo a=new MouseMotionEventDemo();
}
}
}
```

KEY LISTENER

```java
package Listener;
import java.awt.event.KeyEvent;
import java.awt.event.KeyListener;
import javax.swing.JFrame;
import javax.swing.JTextField;
public class KeyEventDemo implements KeyListener
{
  public KeyEventDemo()
  {
  JFrame s=new JFrame("hai");
      JTextField typingArea = new JTextField(20);
      typingArea.addKeyListener(this);
      s.add(typingArea);
      s.setVisible(true);
  }
      public void keyTyped(KeyEvent e) {
      System.out.println("KeyTyped");
  }
  /** Handle the key-pressed event from the text field. */
  public void keyPressed(KeyEvent e) {
      System.out.println("KeyPressed");
  }
  /** Handle the key-released event from the text field. */
  public void keyReleased(KeyEvent e) {
      System.out.println("Keyreleased");
  }
  public static void main(String g[])
  {
        KeyEventDemo a=new KeyEventDemo();
  }
    }
```

ITEM LISTENER

```
package Listener;
import java.awt.event.ItemEvent;
import java.awt.event.ItemListener;
import java.awt.event.KeyEvent;
import java.awt.event.KeyListener;
import javax.swing.JButton;
import javax.swing.JCheckBox;
import javax.swing.JFrame;
import javax.swing.JTextField;
public class itemlistener implements ItemListener
{
   public itemlistener()
   {
   JFrame s=new JFrame("hai");
        JCheckBox a=new JCheckBox("Ok");
        a.addItemListener(this);
        s.add(a);
        s.setVisible(true);
   }
    public static void main(String g[])
  {
         itemlistener l=new itemlistener();
  }
public void itemStateChanged(ItemEvent arg0) {
System.out.println("State changed");
}
   }
```

Window Listener

```java
package Listener;
import java.awt.event.MouseEvent;
import java.awt.event.MouseMotionListener;
import java.awt.event.WindowEvent;
import java.awt.event.WindowListener;
import javax.swing.JFrame;
import javax.swing.JPanel;
import javax.swing.JTextArea;
public class window extends JPanel implements WindowListener {
window()
{
        JFrame b=new JFrame();
        b.addWindowListener(this);
        b.setVisible(true);
}
public static void main(String arg[])
{
        window a=new window();
}
public void windowActivated(WindowEvent arg0) {
        System.out.println("Window activated");
        }
public void windowClosed(WindowEvent arg0) {
        // TODO Auto-generated method stub
        System.out.println("Window closed");
}
public void windowClosing(WindowEvent arg0) {
        // TODO Auto-generated method stub
        System.out.println("Window closing");
}
public void windowDeactivated(WindowEvent arg0) {
        // TODO Auto-generated method stub
```

```java
        System.out.println("Window deactivated");
}
public void windowDeiconified(WindowEvent arg0) {
        // TODO Auto-generated method stub
        System.out.println("Window deiconified");
}
public void windowIconified(WindowEvent arg0) {
        // TODO Auto-generated method stub
        System.out.println("Window Iconified");
}
public void windowOpened(WindowEvent arg0) {
        // TODO Auto-generated method stub
        System.out.println("Window opened");
}}
```

WINDOW FOCUS LISTENER

```java
package Listener;
import java.awt.event.MouseEvent;
import java.awt.event.MouseMotionListener;
import java.awt.event.WindowEvent;
import java.awt.event.WindowFocusListener;
import javax.swing.JFrame;
import javax.swing.JPanel;
import javax.swing.JTextArea;
public class window1 extends JPanel implements WindowFocusListener {
window1()
{
        JFrame b=new JFrame();
        b.addWindowFocusListener(this);
        b.setVisible(true);
}
```

```java
public static void main(String arg[])
{
        window1 b=new window1();
}
public void windowGainedFocus(WindowEvent e) {
        // TODO Auto-generated method stub
        System.out.println("Window gained");
}
public void windowLostFocus(WindowEvent e) {
        // TODO Auto-generated method stub
        System.out.println("Windowlostfocus");
}}
```

WindowStateListener

```java
package Listener;
import java.awt.event.MouseEvent;
import java.awt.event.MouseMotionListener;
import java.awt.event.WindowEvent;
import java.awt.event.WindowStateListener;
import javax.swing.JFrame;
import javax.swing.JPanel;
import javax.swing.JTextArea;
public class window2 extends JPanel implements WindowStateListener {
window2()
{
        JFrame b=new JFrame();
        b.addWindowStateListener(this);
        b.setVisible(true);
}
public static void main(String arg[])
{
        window2 b=new window2();
```

```
}
public void windowStateChanged(WindowEvent e) {
        // TODO Auto-generated method stub
        System.out.println("State Changed");
}}
```

ACTION LISTENER

```
import java.awt.*;
import javax.swing.*;
import java.awt.event.*;
import javax.swing.event.*;
public class A extends JFrame implements ActionListener {
    Scientific() {
        JPanel buttonpanel = new JPanel();
        JButton b1 = new JButton("Hai");
        buttonpanel.add(b1);
        b1.addActionListener(this);
    }
    public void actionPerformed(ActionEvent e) {
        System.out.println("Hai button");
    }
    public static void main(String args[]) {
        A f = new A();
        f.setTitle("ActionListener");
        f.setSize(500,500);
        f.setVisible(true);
}}
```

Java adapter classes

Java adapter classes provide the default implementation of listener interfaces. If you inherit the adapter class, you will not be forced to provide the implementation of all the methods of listener interfaces. So it saves code.

The adapter classes are found in java.awt.event, java.awt.dnd and javax.swing.event packages.

Adapter class	Listener interface
WindowAdapter	WindowListener
KeyAdapter	KeyListener
MouseAdapter	MouseListener
MouseMotionAdapter	MouseMotionListener
FocusAdapter	FocusListener
ComponentAdapter	ComponentListener
ContainerAdapter	ContainerListener
HierarchyBoundsAdapter	HierarchyBoundsListener

Java WindowAdapter 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();
      } }
```

Java MouseAdapter Example

```java
import java.awt.*;
import java.awt.event.*;
public class MouseAdapterExample extends MouseAdapter{
  Frame f;
  MouseAdapterExample(){
    f=new Frame("Mouse Adapter");
    f.addMouseListener(this);
     f.setSize(300,300);
    f.setLayout(null);
    f.setVisible(true);
  }
  public void mouseClicked(MouseEvent e) {
    Graphics g=f.getGraphics();
    g.setColor(Color.BLUE);
    g.fillOval(e.getX(),e.getY(),30,30);
  }
   public static void main(String[] args) {
  new MouseAdapterExample();
} }
```

Java MouseMotionAdapter Example

```java
import java.awt.*;
import java.awt.event.*;
public class MouseMotionAdapterExample extends MouseMotionAdapter{
  Frame f;
  MouseMotionAdapterExample(){
    f=new Frame("Mouse Motion Adapter");
```

```java
        f.addMouseMotionListener(this);
          f.setSize(300,300);
        f.setLayout(null);
        f.setVisible(true);
   }
public void mouseDragged(MouseEvent e) {
   Graphics g=f.getGraphics();
   g.setColor(Color.ORANGE);
   g.fillOval(e.getX(),e.getY(),20,20);
}
public static void main(String[] args) {
   new MouseMotionAdapterExample();
} }
```

Java KeyAdapter Example

```java
import java.awt.*;
import java.awt.event.*;
public class KeyAdapterExample extends KeyAdapter{
   Label l;
   TextArea area;
   Frame f;
   KeyAdapterExample(){
      f=new Frame("Key Adapter");
      l=new Label();
      l.setBounds(20,50,200,20);
      area=new TextArea();
      area.setBounds(20,80,300, 300);
      area.addKeyListener(this);
       f.add(l);f.add(area);
      f.setSize(400,400);
      f.setLayout(null);
      f.setVisible(true);
   }
```
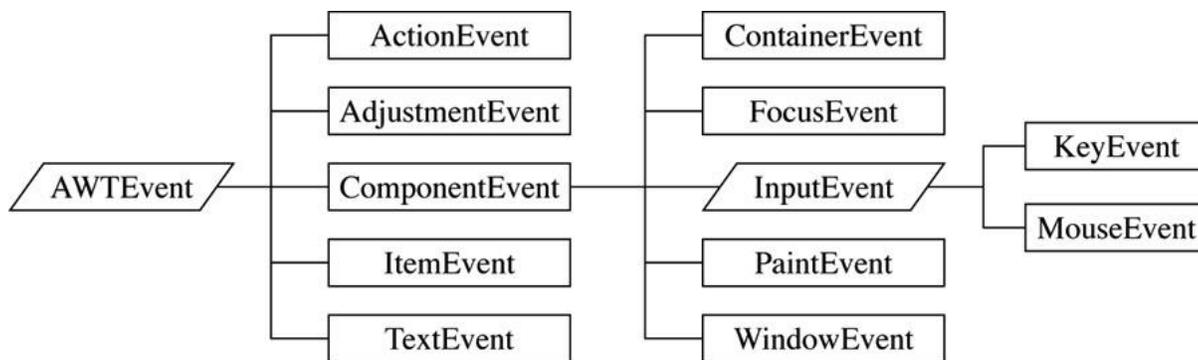
```java
public void keyReleased(KeyEvent e) {
    String text=area.getText();
    String words[]=text.split("\\s");
    l.setText("Words: "+words.length+" Characters:"+text.length());
}
 public static void main(String[] args) {
    new KeyAdapterExample();
} }
```

AWT EVENT HIERARCHY

Swing

- Java Swing tutorial 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.

The hierarchy of java swing API is given below
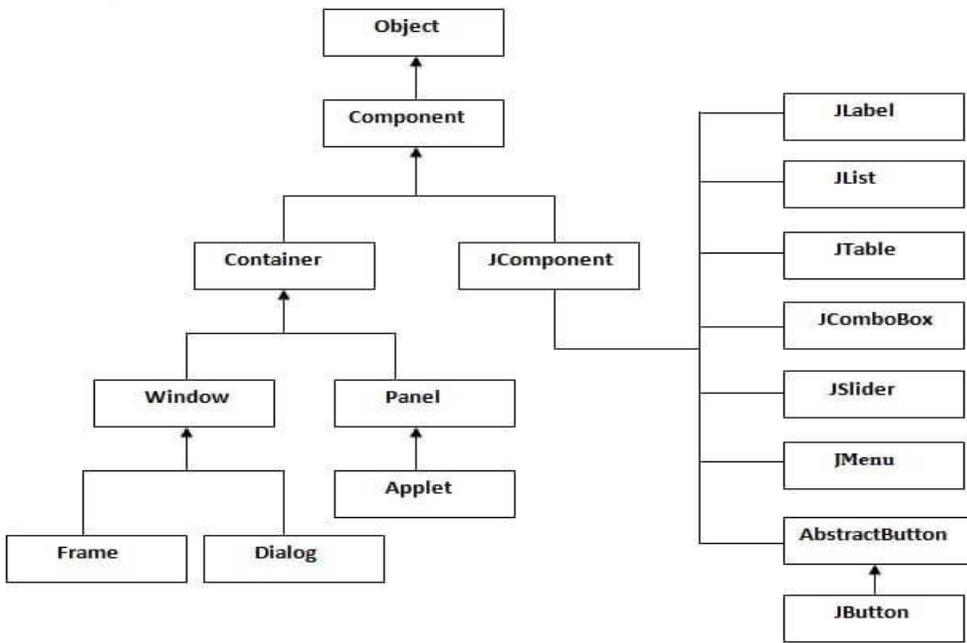
Difference between AWT and Swing

There are many differences between java awt and swing that are given below.

No.	Java AWT	Java Swing
1)	AWT components are **platform-dependent**.	Java swing components are **platform-independent**.
2)	AWT components are **heavyweight**.	Swing components are **lightweight**.
3)	AWT **doesn't support pluggable look and feel**.	Swing **supports pluggable look and feel**.
4)	AWT provides **less components** than Swing.	Swing provides **more powerful components**such as tables, lists, scrollpanes, colorchooser, tabbedpane etc.
5)	AWT **doesn't follows MVC**(Model View Controller) where model represents data, view represents presentation and controller acts as an interface between model and view.	Swing **follows MVC**.

Layout management

Java LayoutManagers

The LayoutManagers are used to arrange components in a particular manner. LayoutManager is an interface that is implemented by all the classes of layout managers. There are following classes that represents the layout managers:

AWT Layout Manager Classes

Following is the list of commonly used controls while designing GUI using AWT.

Sr.No.	LayoutManager & Description
1	**BorderLayout** The borderlayout arranges the components to fit in the five regions: east, west, north, south, and center.
2	**CardLayout** The CardLayout object treats each component in the container as a card. Only one card is visible at a time.
3	**FlowLayout** The FlowLayout is the default layout. It layout the components in a directional flow.
4	**GridLayout** The GridLayout manages the components in the form of a rectangular grid.
5	**GridBagLayout** This is the most flexible layout manager class. The object of GridBagLayout aligns the component vertically, horizontally, or along their baseline without requiring the components of the same size.
6	**GroupLayout** The GroupLayout hierarchically groups the components in order to position them in a Container.
7	**SpringLayout** A SpringLayout positions the children of its associated container according to a set of constraints.
8	**BoxLayout** The BoxLayout is used to arrange the components either vertically or horizontally.
9	**ScrollPaneLayout** The layout manager used by JScrollPane. JScrollPaneLayout is responsible for nine components: a viewport, two scrollbars, a row header, a column header, and four "corner" components.

Border layout:

Example:

```
import java.awt.*;
import javax.swing.*;
 public class Border {
JFrame f;
Border(){
  f=new JFrame();
    JButton b1=new JButton("NORTH");;
  JButton b2=new JButton("SOUTH");;
  JButton b3=new JButton("EAST");;
  JButton b4=new JButton("WEST");;
  JButton b5=new JButton("CENTER");;
      f.add(b1,BorderLayout.NORTH);
  f.add(b2,BorderLayout.SOUTH);
  f.add(b3,BorderLayout.EAST);
  f.add(b4,BorderLayout.WEST);
  f.add(b5,BorderLayout.CENTER);
   f.setSize(300,300);
  f.setVisible(true);
}
public static void main(String[] args) {
  new Border();
} }
```

ScrollPaneLayout:

```
import javax.swing.ImageIcon;
import javax.swing.JFrame;
import javax.swing.JLabel;
import javax.swing.JScrollPane;
public class ScrollPaneDemo extends JFrame
```

```
{
public ScrollPaneDemo() {
super("ScrollPane Demo");
ImageIcon img = new ImageIcon("child.png");
  JScrollPane png = new JScrollPane(new JLabel(img));
  getContentPane().add(png);
setSize(300,250);
setVisible(true);
}
  public static void main(String[] args) {
new ScrollPaneDemo();
}  }
```

Boxlayout

```
import java.awt.*;
import javax.swing.*;
  public class BoxLayoutExample1 extends Frame {
 Button buttons[];
  public BoxLayoutExample1 () {
  buttons = new Button [5];
     for (int i = 0;i<5;i++) {
    buttons[i] = new Button ("Button " + (i + 1));
    add (buttons[i]);
   }
  setLayout (new BoxLayout (this, BoxLayout.Y_AXIS));
setSize(400,400);
setVisible(true);
}
  public static void main(String args[]){
BoxLayoutExample1 b=new BoxLayoutExample1();
}
```

Group layout:

Example

```
public class GroupExample
{
   public static void main(String[] args)
   {
       JFrame frame = new JFrame("GroupLayoutExample");
       frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
       Container contentPanel = frame.getContentPane();
       GroupLayout groupLayout = new GroupLayout(contentPanel);
        contentPanel.setLayout(groupLayout);
        JLabel clickMe = new JLabel("Click Here");
       JButton button = new JButton("This Button");
        groupLayout.setHorizontalGroup(
               groupLayout.createSequentialGroup()
                      .addComponent(clickMe)
                      .addGap(10, 20, 100)
                      .addComponent(button));
       groupLayout.setVerticalGroup(
               groupLayout.createParallelGroup(GroupLayout.Alignment.BASELINE)
                      .addComponent(clickMe)
                      .addComponent(button));
       frame.pack();
       frame.setVisible(true);
   } } }
```

Swing components:

Text Fields

      The object of a JTextField class is a text component that allows the editing of a single line text. It inherits JTextComponent class.

Text Areas

      The object of a JTextArea class is a multi line region that displays text. It allows the editing of multiple line text. It inherits JTextComponent class

Buttons

The JButton class is used to create a labeled button that has platform independent implementation. The application result in some action when the button is pushed. It inherits AbstractButton class.
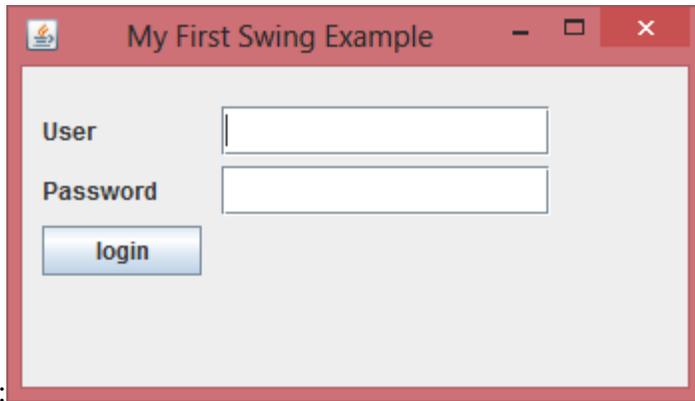
```java
import javax.swing.JButton;

import javax.swing.JFrame;

import javax.swing.JLabel;

import javax.swing.JPanel;

import javax.swing.JPasswordField;

import javax.swing.JTextField;

public class SwingFirstExample {

    public static void main(String[] args) {

    // Creating instance of JFrame

    JFrame frame = new JFrame("My First Swing Example");

    // Setting the width and height of frame

    frame.setSize(350, 200);

    frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);

    /* Creating panel. This is same as a div tag in HTML

     * We can create several panels and add them to specific

     * positions in a JFrame. Inside panels we can add text

     * fields, buttons and other components.

     */

    JPanel panel = new JPanel();

    // adding panel to frame

    frame.add(panel);

    /* calling user defined method for adding components

     * to the panel.

     */

    placeComponents(panel);

    // Setting the frame visibility to true

    frame.setVisible(true);     }

  private static void placeComponents(JPanel panel) {

    /* We will discuss about layouts in the later sections         * of this tutorial. For now we are setting the layout         * to null         */
```

```java
        panel.setLayout(null);
        // Creating JLabel
        JLabel userLabel = new JLabel("User");
        /* This method specifies the location and size
         * of component. setBounds(x, y, width, height)
         * here (x,y) are cordinates from the top left
         * corner and remaining two arguments are the width
         * and height of the component.
         */
        userLabel.setBounds(10,20,80,25);
        panel.add(userLabel);
        /* Creating text field where user is supposed to
         * enter user name.
         */
        JTextField userText = new JTextField(20);
        userText.setBounds(100,20,165,25);
        panel.add(userText);
        // Same process for password label and text field.
        JLabel passwordLabel = new JLabel("Password");
        passwordLabel.setBounds(10,50,80,25);
        panel.add(passwordLabel);
        /*This is similar to text field but it hides the user
         * entered data and displays dots instead to protect
         * the password like we normally see on login screens.
         */
        JPasswordField passwordText = new JPasswordField(20);
        passwordText.setBounds(100,50,165,25);
        panel.add(passwordText);
        // Creating login button
        JButton loginButton = new JButton("login");
        loginButton.setBounds(10, 80, 80, 25);
        panel.add(loginButton);
    }}
```

Output:

:

Check Boxes

The JCheckBox class is used to create a checkbox. It is used to turn an option on (true) or off (false). Clicking on a CheckBox changes its state from "on" to "off" or from "off" to "on ".It inherits JToggleButton class.

Example:

```
import javax.swing.*;
public class CheckBoxExample
{
    CheckBoxExample(){
        JFrame f= new JFrame("CheckBox Example");
        JCheckBox checkBox1 = new JCheckBox("C++");
        checkBox1.setBounds(100,100, 50,50);
        JCheckBox checkBox2 = new JCheckBox("Java", true);
        checkBox2.setBounds(100,150, 50,50);
        f.add(checkBox1);
        f.add(checkBox2);
        f.setSize(400,400);
        f.setLayout(null);
        f.setVisible(true);
    }
public static void main(String args[])
    {
    new CheckBoxExample();
```

```
}}
```

Radio Buttons

The JRadioButton class is used to create a radio button. It is used to choose one option from multiple options. It is widely used in exam systems or quiz. It should be added in ButtonGroup to select one radio button only.
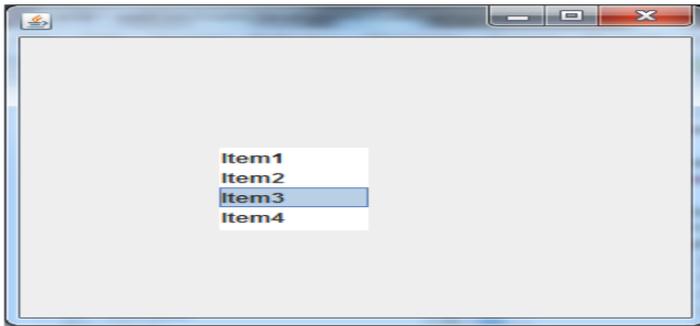
```java
import javax.swing.*;
import java.awt.event.*;
class RadioButtonExample extends JFrame implements ActionListener{
JRadioButton rb1,rb2;
JButton b;
RadioButtonExample(){
rb1=new JRadioButton("Male");
rb1.setBounds(100,50,100,30);
rb2=new JRadioButton("Female");
rb2.setBounds(100,100,100,30);
ButtonGroup bg=new ButtonGroup();
bg.add(rb1);bg.add(rb2);
b=new JButton("click");
b.setBounds(100,150,80,30);
b.addActionListener(this);
add(rb1);add(rb2);add(b);
setSize(300,300);
setLayout(null);
setVisible(true);
}
public void actionPerformed(ActionEvent e){
if(rb1.isSelected()){
JOptionPane.showMessageDialog(this,"You are Male.");
}
if(rb2.isSelected()){
JOptionPane.showMessageDialog(this,"You are Female.");
}   }
```

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

Lists

The object of JList class represents a list of text items. The list of text items can be set up so that the user can choose either one item or multiple items. It inherits JComponent class.

```
import javax.swing.*;
public class ListExample
{
   ListExample(){
     JFrame f= new JFrame();
     DefaultListModel<String> l1 = new DefaultListModel<>();
      l1.addElement("Item1");
      l1.addElement("Item2");
      l1.addElement("Item3");
      l1.addElement("Item4");
      JList<String> list = new JList<>(l1);
      list.setBounds(100,100, 75,75);
      f.add(list);
      f.setSize(400,400);
      f.setLayout(null);
      f.setVisible(true);
   }
public static void main(String args[])
  {
  new ListExample();
  }}
```

Choices (JComboBox)

The object of Choice class is used to show popup menu of choices. Choice selected by user is shown on the top of a menu. It inherits JComponent class.

```
import javax.swing.*;
public class ComboBoxExample {
JFrame f;
ComboBoxExample(){
    f=new JFrame("ComboBox Example");
    String country[]={"India","Aus","U.S.A","England","Newzealand"};
    JComboBox cb=new JComboBox(country);
    cb.setBounds(50, 50,90,20);
    f.add(cb);
    f.setLayout(null);
    f.setSize(400,500);
    f.setVisible(true);
}
public static void main(String[] args) {
    new ComboBoxExample();
}    }
```

Output:
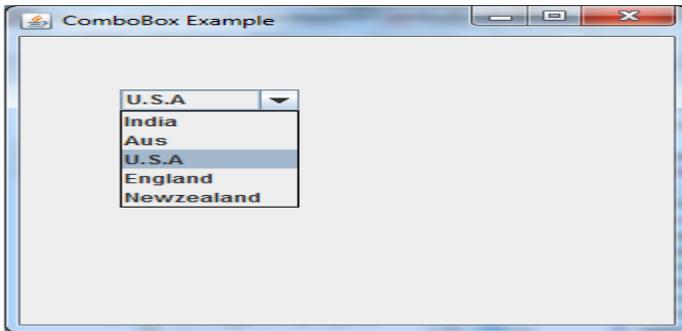
Scrollbars

The object of JScrollbar class is used to add horizontal and vertical scrollbar. It is an implementation of a scrollbar. It inherits JComponent class.

```
mport javax.swing.*;
class ScrollBarExample
{
ScrollBarExample(){
  JFrame f= new JFrame("Scrollbar Example");
 JScrollBar s=new JScrollBar();
s.setBounds(100,100, 50,100);
f.add(s);
f.setSize(400,400);
f.setLayout(null);
f.setVisible(true);
}
public static void main(String args[])
{
new ScrollBarExample();
}}
```

Output:
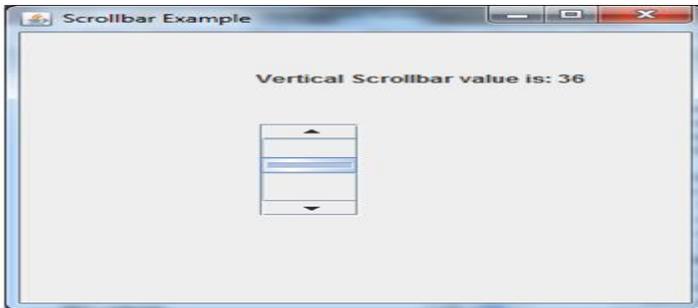
Windows

The class JWindow is a container that can be displayed but does not have the title bar

Menus

The JMenuBar class is used to display menubar on the window or frame. It may have several menus.

The object of JMenu class is a pull down menu component which is displayed from the menu bar. It inherits the JMenuItem class.

The object of JMenuItem class adds a simple labeled menu item. The items used in a menu must belong to the JMenuItem or any of its subclass.

```java
import javax.swing.*;
class MenuExample
{
        JMenu menu, submenu;
        JMenuItem i1, i2, i3, i4, i5;
        MenuExample(){
        JFrame f= new JFrame("Menu and MenuItem Example");
        JMenuBar mb=new JMenuBar();
        menu=new JMenu("Menu");
        submenu=new JMenu("Sub Menu");
        i1=new JMenuItem("Item 1");
        i2=new JMenuItem("Item 2");
        i3=new JMenuItem("Item 3");
        i4=new JMenuItem("Item 4");
        i5=new JMenuItem("Item 5");
        menu.add(i1); menu.add(i2); menu.add(i3);
        submenu.add(i4); submenu.add(i5);
```
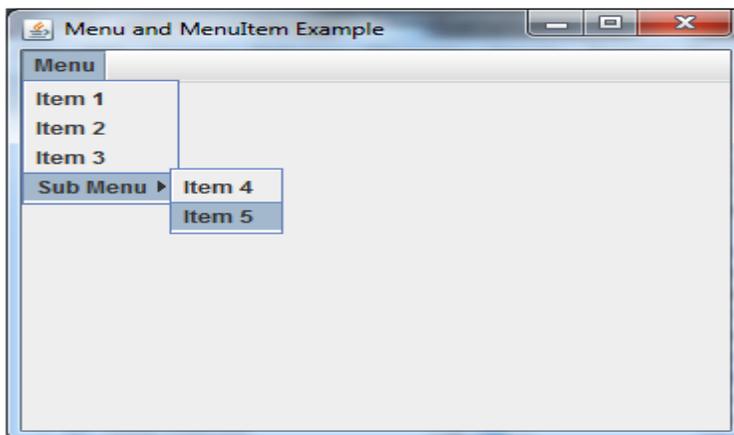
```
            menu.add(submenu);

            mb.add(menu);

            f.setJMenuBar(mb);

            f.setSize(400,400);

            f.setLayout(null);

            f.setVisible(true);

        }
        public static void main(String args[])

        {

        new MenuExample();

        }}
```

Output :

Dialog Boxes.

The JDialog control represents a top level window with a border and a title used to take some form of input from the user. It inherits the Dialog class.Unlike JFrame, it doesn't have maximize and minimize buttons.

Example:

```
import javax.swing.*;

import java.awt.*;

import java.awt.event.*;

public class DialogExample {

   private static JDialog d;

   DialogExample() {

       JFrame f= new JFrame();
```

```java
d = new JDialog(f , "Dialog Example", true);
d.setLayout( new FlowLayout() );
JButton b = new JButton ("OK");
b.addActionListener ( new ActionListener()
{
    public void actionPerformed( ActionEvent e )
    {
        DialogExample.d.setVisible(false);  }}};
d.add( new JLabel ("Click button to continue."));
d.add(b);
d.setSize(300,300);
d.setVisible(true);
}
public static void main(String args[])
{
    new DialogExample(); }}
```

Output: