MODULE 1: INTRODUCTION TO JAVA AND CLASSES

Syllabus:

Introduction: Introduction, History of Java.;Java Buzzwords; Java's Byte code, Java Development Kit (JDK); Object-oriented programming; Simple Java Programs.

Introducing Classes: Classes fundamentals; Declaring objects; Constructors;this keyword;garbage collection;Overloading methods;Access Control;final key word;Nested and inner classes

History of Java:

Java is a general-purpose object oriented programming language developed by sun Microsystems of USA in the year 1991. The original name of Java is Oak. Java was designed for the development of the software for consumer electronic devices like TVs, VCRs, etc.

Introduction: Java is a general purpose programming language. We can develop two types of Java application. They are:

(1). Stand alone Java application.

(2). Web applets.

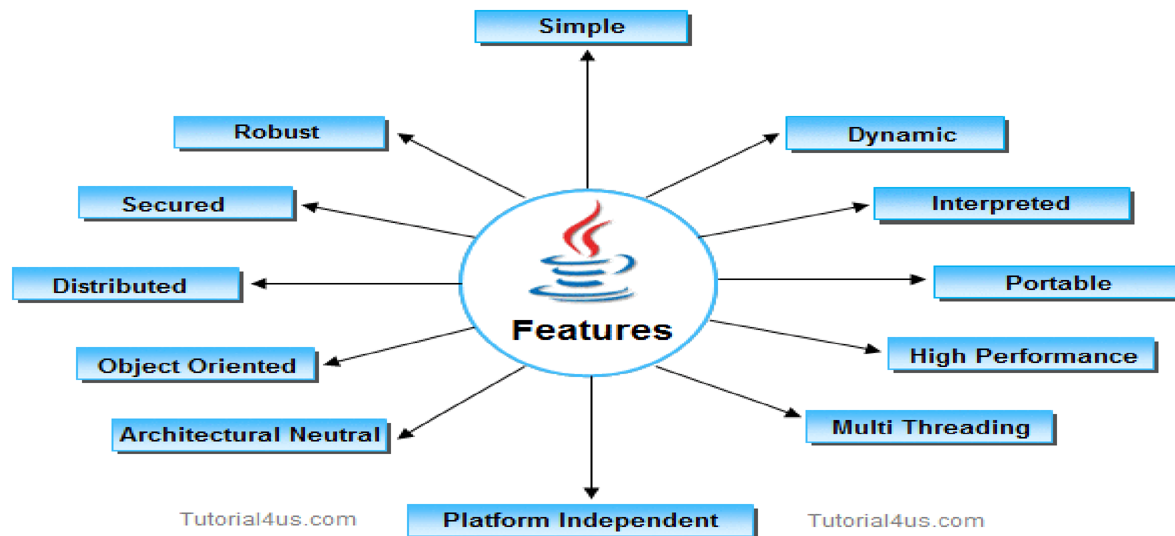
Stand alone Java application: Stand alone Java application are programs written in Java to carry out certain tasks on a certain stand alone system. Executing a stand-alone Java program contains two phases:

(a) Compiling source coded into bytecode using javac compiler.

(b) Executing the bytecodede program using Java interpreter.

Java applet: Applets are small Java program developed for Internet application. An applet located on a distant computer can be downloaded via Internet and execute on local computer.

Java Features/Buzz words:



1. Simple

Learning java will be much easier if we already understand the basic concept of object oriented program, then moving to java will need less effort.

Java is simple because of the following factors:

- Java is **free from pointer** due to this execution time of application is improve. [whenever we write a Java program without pointers then internally it is converted into the equivalent pointer program].
- Java have **Rich set of API** (application protocol interface).
- Java have **Garbage Collector** which is always used to collect un-Referenced (unused) Memory location for improving performance of a Java program.
- Java contains user friendly syntax for developing Java applications.

2. Object Oriented:

Java was not designed to be source code compatible with any other language . One outcome of this was a clean, usable, pragmatic approach to objects. Objects model in simple/easy to extend.

3. Robust:

- To gain reliability, javas restrict you in few key areas, to force you to find your mistakes early in program development. It also frees you from having to worry about the most common causes of programming errors.
- Java is a strictly typed language, it checks user code at compile time and run time. The main reasons for failure are,

1. Memory management mistakes
2. Mishandled exceptional conditions

- Java virtually eliminates these problem by managing memory allocation and deallocation. Exceptions are handled by providing object oriented exception handling .

4.Portability:

- Many types of computer and OS are in use throughout the world and many are connected to internet. For programs , to be dynamically downloaded , some means of generating portable executable code is needed.
- If any language supports platform independent and architectural neutral feature known as portable. The languages like C, CPP, Pascal are treated as non-portable language. JAVA is a portable language.

Portability = platform independent + architecture

5.Security:

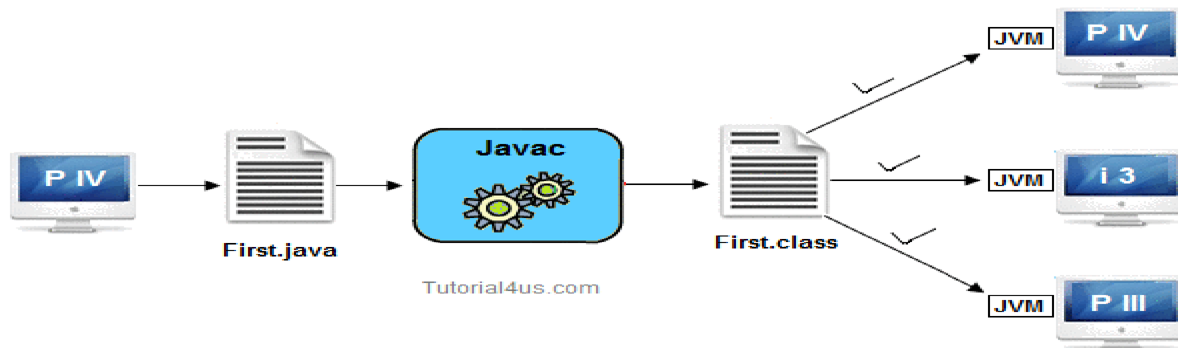
Java provides a firewall between a networked applet and your computer. When you use a java compatible web browser, you can safely download java applets without fear of viral infection. Java is more secured language compare to other language; All java code is covered into **byte code** after compilation which is not readable by human.

6.Multithread:

- It allows you to write an elegant yet sophisticated solution for multiprocess synchronization that enables you to construct smoothly running interactive systems.
- A flow of control is known as thread. When any Language execute multiple thread at a time that language is known as multithreaded Language. java and .net are multithreaded Language.

7.Architectural Neutral:

- The goal of java designers to develop "**write once, run anywhere**, anytime, forever" so that a program can be independent of the architecture of the system in which it is running.
- A Language or Technology is said to be Architectural neutral which can run on any available processors in the real world without considering there architecture and vendor (providers) irrespect to its development and compilation



8.High Performance:

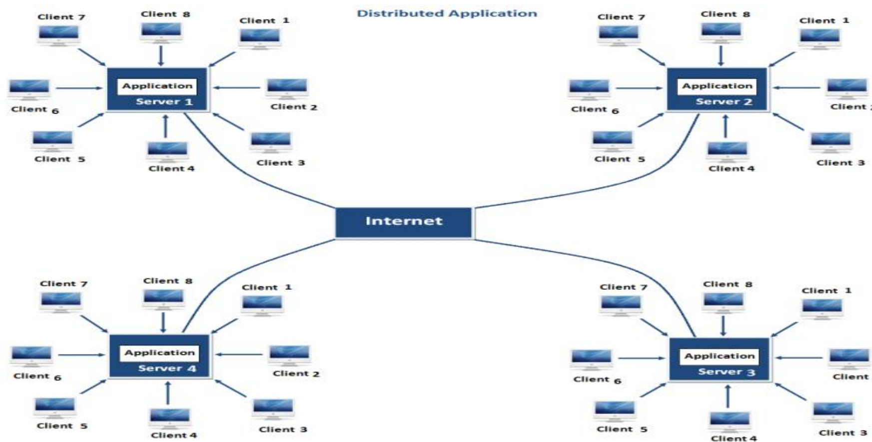
It enables the creation of cross-platform programs by compiling into an intermediate representation called java bytecode .It can be interpreted on any system that provide JVM.

Java have high performance because of following reasons;

- Java **uses Bytecode** which is more faster than ordinary pointer code so Performance of java is high.
- **Garbage collector**, collect the unused memory space and improve the performance of java application.
- Java have **no pointers** so that using java program we can develop an application very easily.
- It **support multithreading**, because of this time consuming process can be reduced to execute the program.

9.Distributed:

- Java is designed for distributed environment of internet ,since it handles TCP/IP protocol .The original version of java included features for intra address-space messaging. This allowed objects on two different computers to execute procedures remotely.
- We can create distributed applications in java. RMI and EJB are used for creating distributed applications. In distributed application multiple client system are depends on multiple server systems so that even problem occurred in one server will never be reflected on any client system.



10. **Dynamic:**

- Java programs carry with them substantial amount of run time type information that is used to verify and resolve accesses to objects at run time thus making it possible to dynamically link code in a safe and expedient manner
- Java programming support Dynamic memory allocation due to this memory wastage is reduce and improve performance of application. The process of allocating the memory space to the input of the program at a run-time is known as dynamic memory allocation, In java programming to allocate memory space by dynamically we use an operator called 'new' 'new' operator is known as dynamic memory allocation operator.

Java Environment variable/JAVA runtime Environment(JRE):

Java environment is a collection of tools and class, methods. The developments tools are part of system called as java development kit(JDK) and classes ,methods are part of the java standard library(JSL) also known as the Application programming interface(API).

1) Java Development kit (JDK).

2) Application programming interface (API)/Java Standard library(JSL)

Java Development kits(java software:jdk1.6): Java development kit comes with a number of Java development tools. They are:

(1) Appletviewer: Enables to run Java applet.
(2) javac: Java compiler.
(3) java : Java interpreter.
(4) javah : Produces header files for use with native methods.
(5) javap : Java disassembler.
(6) javadoc : Creates HTML documents for Java source code file.
(7) jdb : Java debugger which helps us to find the error.

A source program written in java is compiled using "javac" (java compiler) and executed using "java" (java interpreter). The "jdb" (java debugger) is used to locate errors if any in the source file.

Tool	*Description*
appletviewer	Enables us to run Java applets (without actually using a Java-compatible browser).
java	Java interpreter, which runs applets and applications by reading and interpreting bytecode files.
javac	The Java compiler, which translates Java sourcecode to bytecode files that the interpreter can understand.
javadoc	Creates HTML-format documentation from Java source code files.
javah	Produces header files for use with native methods.
javap	Java disassembler, which enables us to convert bytecode files into a program description.
jdb	Java debugger, which helps us to find errors in our programs.

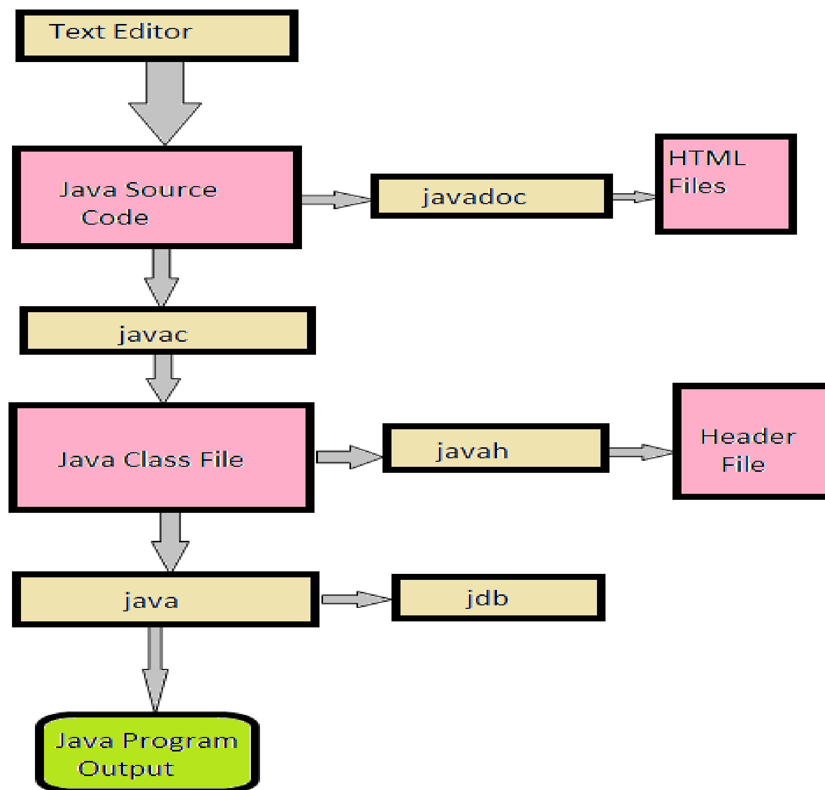


Fig:Process of building and running java application

Application programming interface(API):

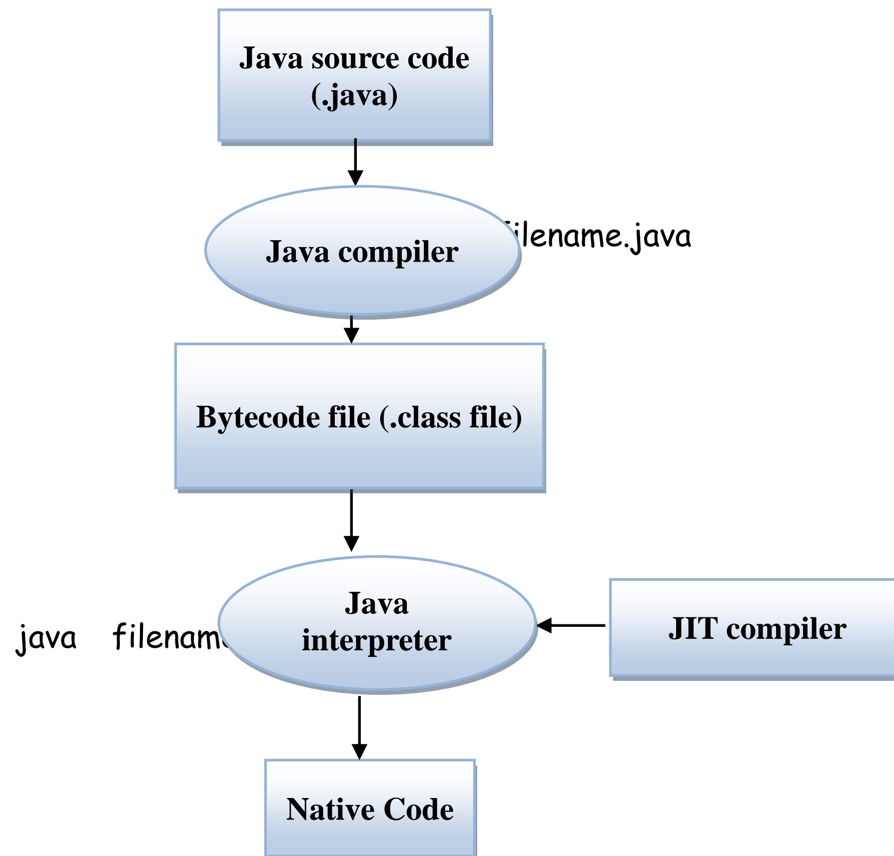
API is a collection of classes and methods are grouped into several different packages.

- Language Support package: Is a collection of classes and methods required for implementing basic features of java. EX: **import java.lang.*;**
- Utilities package: Is a collection of classes to provide utility function such as data,time function Ex: **import java.util.*;**
- Input/output package: Is a collection of classes required for input and output manipulation EX: **import java.io.*;**
- Networking package: Is a collection of classes for communicating with other computer via internet. Ex: **import java.net.*;**

- AWT package: The abstract window tool kit package contains classes that implements platform independent graphical user interface. EX: **import.java.awt.*;**
- Applet package: This include a set of classes that allows us to create java applets. Ex: **import java.applets.*;**

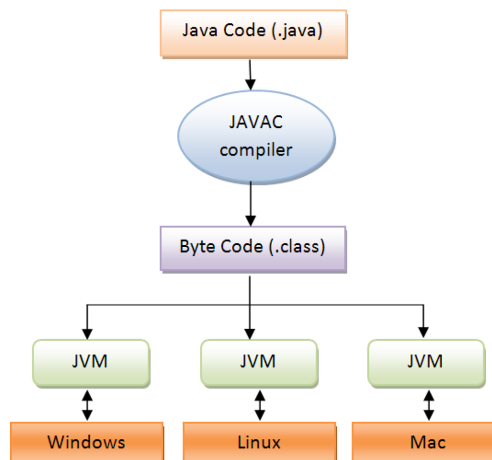
JAVA IS INTERPRETED:

Java as a language initially gained popularity mainly due to its platform independent architecture or portability feature. The reason for java to be portable is that it is interpreted.



Just in time is added in jvm which makes the program execution much faster.

JVM(Java Virtual Machine)



The concept of Write-once-run-anywhere (known as the Platform independent) is one of the important key feature of java language that makes java as the most powerful language. Not even a single language is idle to this feature but java is closer to this feature. The programs written on one platform can run on any platform provided the platform must have the JVM(**Java Virtual Machine**). A Java virtual machine (JVM) is a virtual machine that can execute Java bytecode. It is the code execution component of the Java software platform.

Basic concepts of object oriented programming

Object:

This is the basic unit of object oriented programming. That is both data and method that operate on data are bundled as a unit called as object. **It is a real world entity (Ex:a person, book, tables, chairs etc…)**

Class:

Class is a **collection of objects** or class is a **collection of instance variables and methods**. When you define a class, you define a blueprint for an object. This doesn't actually define any data, but it does define what the class name means,

that is, what an object of the class will consist of and what operations can be performed on such an object.

Abstraction:

Data abstraction refers to, **providing only essential information** to the outside word and hiding their background details ie. to represent the needed information in program without presenting the details.
For example, a database system hides certain details of how data is stored and created and maintained. Similar way, C++ classes provides different methods to the outside world without giving internal detail about those methods and data.

Encapsulation:

Encapsulation is ***placing the data and the methods/functions*** that work on that data in the same place. While working with procedural languages, it is not always clear which functions work on which variables but object-oriented programming provides you framework to place the data and the relevant functions together in the same object.

Inheritance:

One of the most useful aspects of object-oriented programming is **code reusability**. As the name suggests Inheritance is the process of forming a new class from an existing class that is from the existing class called as base class, new class is formed called as derived class.
This is a very important concept of object oriented programming since this feature helps to reduce the code size.

Polymorphism:

The ability to use a method/function in different ways in other words giving different meaning for method/ functions is called polymorphism. Poly refers many. That is a single method/function functioning in many ways different upon the usage is called polymorphism.

Simple java Program

Java Building and running Process:

1. **Open the notepad and type the below program**

Simple Java program:

Example:

```
class Sampleone
{
	public static void main(String args[])
	{
		System.out.println("Welcome to JAVA");
	}
}
```

Description:

(1) **Class declaration:** "class sampleone" declares a class, which is an object-oriented construct. Sampleone is a Java identifier that specifies the name of the class to be defined.

(2) **Opening braces:** Every class definition of Java starts with opening braces and ends with matching one.

(3) **The main line:** the line " public static void main(String args[]) " defines a method name main. Java application program must include this main. This is the starting point of the interpreter from where it starts executing. A Java program can have any number of classes but only one class will have the main method.

(4) **Public:** This key word is an access specifier that declares the main method as unprotected and therefore making it accessible to the all other classes.

(5) **Static:** Static keyword defines the method as one that belongs to the entire class and not for a particular object of the class. The main must always be declared as static.

(6) **Void:** the type modifier void specifies that the method main does not return any value.

(7) **The println:** It is a method of the object out of system class. It is similar to the printf or cout of c or c++.

2. Save the above program with .java extension, here file name and class name should be same, ex: Sampleone.java,
3. Open the command prompt and **Compile** the above program

   javac Sampleone.java

From the above compilation the java compiler produces a bytecode(.class file)

4. Finally run the program through the **interpreter**

   java Sapleone.java

Output of the program:
Welcome to JAVA

Implementing a Java program: Java program implementation contains three stages. They are:

1. Create the source code.
2. Compile the source code.
3. Execute the program.

(1) Create the source code:

1. Any editor can be used to create the Java source code.
2. After coding the Java program must be saved in a file having the same name of the class containing main() method.
3. Java code file must have .Java extension.

(2) Compile the source code:

1. Compilation of source code will generate the bytecode.
2. JDK must be installed before completion.
3. Java program can be compiled by typing javac <filename>.java
4. It will create a file called <filename>.class containing the bytecode.

(3) Executing the program:

1. Java program once compiled can be run at any system.
2. Java program can be execute by typing Java <filename>

Java Program structure: Java program structure contains six stages. They are:

(1) Documentation section: The documentation section contains a set of comment lines describing about the program.

(2) Package statement: The first statement allowed in a Java file is a package statement. This statement declares a package name and informs the compiler that the class defined here belong to the package.

Package student;

(3) Import statements: Import statements instruct the compiler to load the specific class belongs to the mentioned package.

Import student.test;

(4) Interface statements: An interface is like a class but includes a group of method deceleration. This is an optional statement.

(5) Class definition: A Java program may contain multiple class definition The class are used to map the real world object.

(6) Main method class: The main method creates objects of various classes and establish communication between them. On reaching to the end of main the program terminates and the control goes back to operating system.

Java command line arguments: Command line arguments are the parameters that are supplied to the application program at the time when they are invoked. The main() method of Java program will take the command line arguments as the parameter of the args[ ] variable which is a string array.

Example
:

```
Class Comlinetest
{
        public static void main(String args[ ] )
        {
                int count, n =
                0; string str;
                count = args.length;
                System.out.println ( " Number of arguments :" +
                count); While ( n < count )
                {
                        str = args[ n
                        ]; n = n + 1;
                        System.out.println( n + " : " + str);
                }
        }
```

Java API

Run/Calling the program:

javac Comlinetest.java

java Comlinetest Java c cpp

fortran Output:

1 : Java

2 : c

3 : cpp

4 : fortran

Java standard library includes hundreds of classes and methods grouped into several functional packages. Most commonly used packages are:

(a) Language support Package.

(b) Utilities packages.

(c) Input/output packages

(d) Networking packages

(e) AWT packages.

(f) Applet packages.

INTRODUCING CLASSES:

Definition

A class is a template for an object, and defines the data fields and methods of the object. The class methods provide access to manipulate the data fields. The "data fields" of an object are often called "instance variables."

Example Program:

Program to calculate Area of Rectangle

```
class Rectangle
{
        int length;                         //Data Member or instance Variables
        int width;

        void getdata(int x,int y)                 //Method
        {
                length=x;
                width=y;
        }

        int rectArea()                            //Method
        {
                return(length*width);
        }
}


class RectangleArea
{
    public static void main(String args[])
    {
        Rectangle rect1=new Rectangle();//object creation rect1.getdata(10,20);
            //calling methods using object with dot(.)

        int area1=rect1.rectArea(); System.out.println("Area1="+area1);

    }
}
```

- After defining a class, it can be used to create objects by instantiating the class. Each object occupies some memory to hold its instance variables (i.e. its state).
- After an object is created, it can be used to get the desired functionality together with its class.

Creating instance of a class/Declaring objects:

```
Rectangle rect1=new Rectangle() Rectangle
rect2=new Rectangle()
```

- The above two statements declares an **object rect1 and rect2 is of type Rectangle class using new operator** , this operator dynamically allocates memory for an object and returns a refernce to it.in java all class objects must be dynamically allocated.

We can also declare the object like this:

Rectangle rect1;   // declare reference to object.

rect1 =new Rectangle()   // allocate memory in the Rectangle object.

The Constructors:

- A constructor initializes an object when it is created. It has the same name as its class and is syntactically similar to a method. However, constructors have no explicit return type.
- Typically, you will use a constructor to give initial values to the instance variables defined by the class, or to perform any other startup procedures required to create a fully formed object.

- All classes have constructors, whether you define one or not, because Java automatically provides a default constructor that initializes all member variables to zero. However, once you define your own constructor, the default constructor is no longer used.

Example:

Here is a simple example that uses a constructor:

```
// A simple constructor.
class MyClass

{

     int x;


   // Following is the constructor

   MyClass()

  {

      x = 10;

   }

}
```

You would call constructor to initialize objects as follows:

```
class ConsDemo

{

   public static void main(String args[])

  {

      MyClass t1 = new MyClass();
      MyClass t2 = new MyClass();

      System.out.println(t1.x + " " + t2.x);

   }

}
```

Parameterized Constructor:

- ✓ Most often you will need a constructor that accepts one or more parameters. Parameters are added to a constructor in the same way that they are added to a method: just declare them inside the parentheses after the constructor's name.

Example:

Here is a simple example that uses a constructor:

```
// A simple constructor.
class MyClass

{

   int x;

   // Following is the Parameterized constructor

   MyClass(int i )

   {

      x = 10;

   }

}
```

You would call constructor to initialize objects as follows:

```
class ConsDemo

{

   public static void main(String args[])

  {

      MyClass t1 = new MyClass( 10 );
      MyClass t2 = new MyClass( 20 );
      System.out.println(t1.x + " " + t2.x);

   }

}
```

This would produce following result:

```
10 20
```

this keyword

- this keyword can be used to refer current class instance variable.
- If there is ambiguity between the instance variable and parameter, this keyword resolves the problem of ambiguity.

Understanding the problem withoUT this keyword

Let's understand the problem if we don't use this keyword by the example given below:

```
class student

{

     int id;
   String
   name;

   student(int id,String name)
   {
        id = id;
        name = name;
   }
   void display()
   {
        System.out.println(id+" "+name);
   }
}
Class MyPgm

{

   public static void main(String args[])
   {
        student s1 = new
```

```
                student(111,"Anoop"); student s2 =
                new student(321,"Arayan");
                s1.display();
                s2.display();
        }
}
```

Output: 0 null
        0 null

In the above example, parameter (formal arguments) and instance variables are same that is why we are using this keyword to distinguish between local variable and instance variable

SolUTIOn of the above problem by this keyword

```
  //example of this keyword
 class Student
  {

          int  id;
          String
          name;

          student(int id,String name)

          {
                  this.id = id;

                  this.name = name;

          }
           void display()
          {
                  System.out.println(id+" "+name);
          }
 }
     Class MyPgm

     {

          public static void main(String args[])
          {
                  Student s1 = new
                  Student(111,"Anoop"); Student s2 =
                  new Student(222,"Aryan");
                  s1.display();
                  s2.display();
          }
     }
 Output111 Anoop
        222 Aryan
```

Garbage Collection

In Java destruction of object from memory is done automatically by the JVM. When there is no reference to an object, then that object is assumed to be no longer needed and the memories occupied by the object are released. This technique is called **Garbage Collection**. This is accomplished by the JVM.

Can the Garbage Collection be forced explicitly?

No, the Garbage Collection cannot be forced explicitly. We may request JVM for **garbage collection** by calling **System.gc()** method. But this does not guarantee that JVM will perform the garbage collection.

Advantages of Garbage Collection

1. Programmer doesn't need to worry about dereferencing an object.
2. It is done automatically by JVM.
3. Increases memory efficiency and decreases the chances for memory leak.

gc() Method

gc() method is used to call garbage collector explicitly. However **gc()** method does not guarantee that JVM will perform the garbage collection. It only requests the JVM for garbage collection. This method is present in **System** and **Runtime** class.

Example for gc() method

```
public class Test
{

    public static void main(String[] args)
    {
        Test t = new
        Test(); t=null;
        System.gc();
    }
    public void finalize()
    {
        System.out.println("Garbage Collected");
    }
}
```

Output :

Garbage Collected

Overloading Methods

When two or more methods **in the** same class have the same name
but **different** parameters, it's called **Overloading**.

```
//Demonstrate method overloading
Class Overload Demo{
     void test()  {
      System.out.println("No parameters");
                    }
//overload test for one integer parameter
Void test(int a) {
  System.out.println("a:   "  +a);
}
```

Constructor Overloading:

Constructor overloading is a technique in Java in which a class can have any number of constructors that differ in parameter lists. The compiler differentiates these constructors by taking the number of parameters, and their type.

Example:

```
class A
{
int a=10,b=20;
A(int a,int b)
{
System.out.println("Value of a: "+a);
System.out.println("Value of b: "+b);
}
```

```
A(int a,int b,int c)
{
System.out.println("Value of a: "+a);
System.out.println("Value of b: "+b);
System.out.println("Value of b: "+c);
}
public static void main(String ar[])
{
A a1=new A(2,3);
A a2=new A(2,3,4);
}
}
```

Access Control:

Which parts of a program can access the members of a class.

Access specifies are

1.**Public:**A **public** member is accessible from anywhere outside the class but within a program

2.**Private: A private** members cannot be accessed (or viewed) from outside the class.

3. **Protected** –A protected members cannot be accessed from outside the class, however, they can be accessed in inherited classes

Example:

```
Class Test
{
    int a;
    public int b;
    private int c;
//methods to access c
Void setc(int i)
{
   c=i;
}
Int getc()
{
   return c;
}
```

```
}
Class   AccessTest{
Public static void main(String args[ ]){
            Test ob=new Test();
             ob.a=10;
             ob.b=20;
             ob.c=100//Error
             ob.setc(100);
             System.out.println("a,b and c:" ob+a  +"  "+ob.b.+ " " +ob.getc());
}
}
```

Final Keyword

In java language final keyword can be used in following way.

- Final at variable level
- Final at method level
- Final at class level

1. **Final at variable level**

Final keyword is used to make a variable as a constant. This is similar to const in other language. A variable declared with the final keyword cannot be modified by the program after initialization.

Example:

```
class A
{
final  int a=10;
public static void main(String ar[])
{
System.out.println("static variable a="+a1.a); // no error
System.out.println("static variable a="+a1.a++); //final variable cannot be modified (error)
}
}
```

2. **Final at method level**

- It makes a method final, meaning that sub classes cannot override this method. The compiler checks and gives an error if you try to override the method.

- When we want to restrict overriding, then make a method as a final.
- Example:

```
class A
{
final void add()
{
System.out.println("sum="+(2+3));
}
}
class B extends A
{
void add()   // error because final method cannot override
{
System.out.println("sum="+(2+3));
}
public static void main(String ar[])
{ A a1=new A();
a1.add();
}
}
```

3. **Final at class level**

It makes a class final, meaning that the class cannot be inheriting by other classes. When we want to restrict inheritance then make class as a final.

Example:

```
final class A
```

```
{
void add()
{
System.out.println("sum="+(2+3));
}
}
class B extends A  // error because final class cannot inherited.
{
public static void main(String ar[])
{
A a1=new A();
a1.add();
}
}
```

Inner class/Nested class

- It has access to all variables and methods of **Outer** class and may refer to them directly. But the reverse is not true, that is, **Outer** class cannot directly access members of **Inner** class.
- One more important thing to notice about an **Inner** class is that it can be created only within the scope of **Outer** class. Java compiler generates an error if any code outside **Outer** class attempts to instantiate **Inner** class.

Example of Inner class

```
class Outer

{

        public void display()
        {
                Inner in=new Inner();

                in.show();
        }


        class Inner
        {

                public void show()
                {
                        System.out.println("Inside inner");
                }
        }
}


class Test

{

        public static void main(String[] args)
        {
                Outer ot=new
        }       Outer(); ot.display();
}
```

Output:

Inside
inner

MODULE 2: INHERITANCE,PACKAGES AND INTERFACES

Syllabus:

Inheritance: Inheritance Basics, Using Super, Creating a multilevel heiararchy,When Constructors are called, Method overriding, Dynamic method dispatch, Using Abstract class, Using final with inheritance, The object class

Packages and Interfaces: Packages, Access Protection,Importing packages,Interfaces

Inheritance:

- ✓ As the name suggests, inheritance means to take something that is already made. It is one of the most important features of Object Oriented Programming. It is the concept that is used for **reusability** purpose.
- ✓ Inheritance is the mechanism through which we can derive classes from other classes.
- ✓ The derived class is called as child class or the subclass or we can say the extended class and the class from which we are deriving the subclass is called the base class or the parent class.
- ✓ To derive a class in java the keyword **extends** is used. The following kinds of inheritance are there in java.

Types of Inheritance

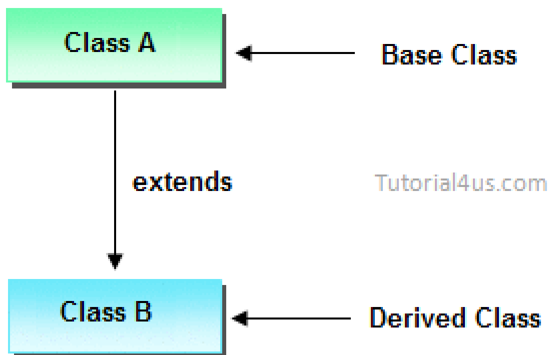
Based on number of ways inheriting the feature of base class into derived class we have five Inheritance type they are:

- Single inheritance
- Multiple inheritance(Interface)
- Hierarchical inheritance
- Multilevel inheritance
- Hybrid inheritance

1. *Single inheritance*

In single inheritance there exists single base class and single derived class.

Property of base class inherited into sub class and sub class having its own property.

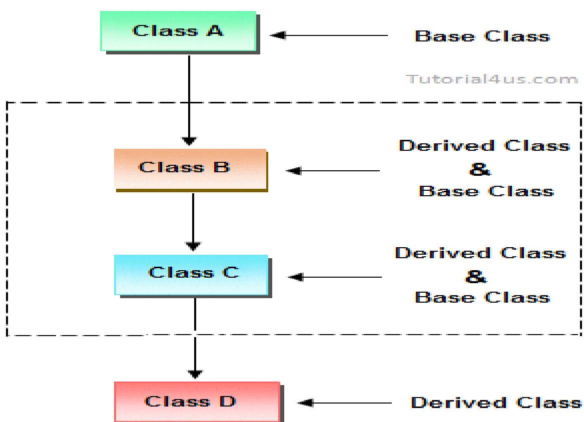


Example:

```
class A
{
void display()
{
System.out.println("base class method");
}
}
class B extends A
{
void display2()
{
System.out.println("sub class methods");
}
Public static void main(String ar[])
{
B a1=new B();
a1.display();
a1.display2();
}}
```

2.Multilevel inheritances

- In Multilevel inheritances there exists single base class, single derived class and multiple intermediate base classes.
- An intermediate base class is one in one context with access derived class and in another context same class access base class.

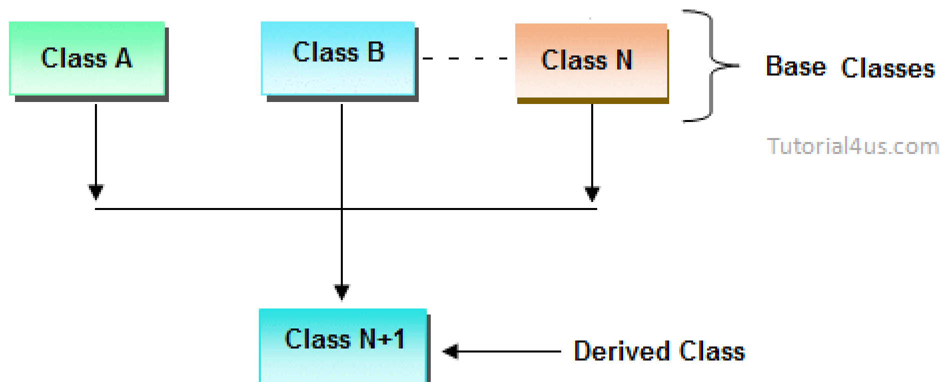


```
class A
{
void display()
{
System.out.println("A class method");
}
}

class B extends A
{
void display1()
{
System.out.println("B class methods");
}



class C extends B
{
void display2()
{
System.out.println("C class methods");
}
public static void main(String ar[])
{
```

```
C a1=new C();
a1.display();
a1.display1();
a1.display2();
}
}
```

3.Multiple inheritance

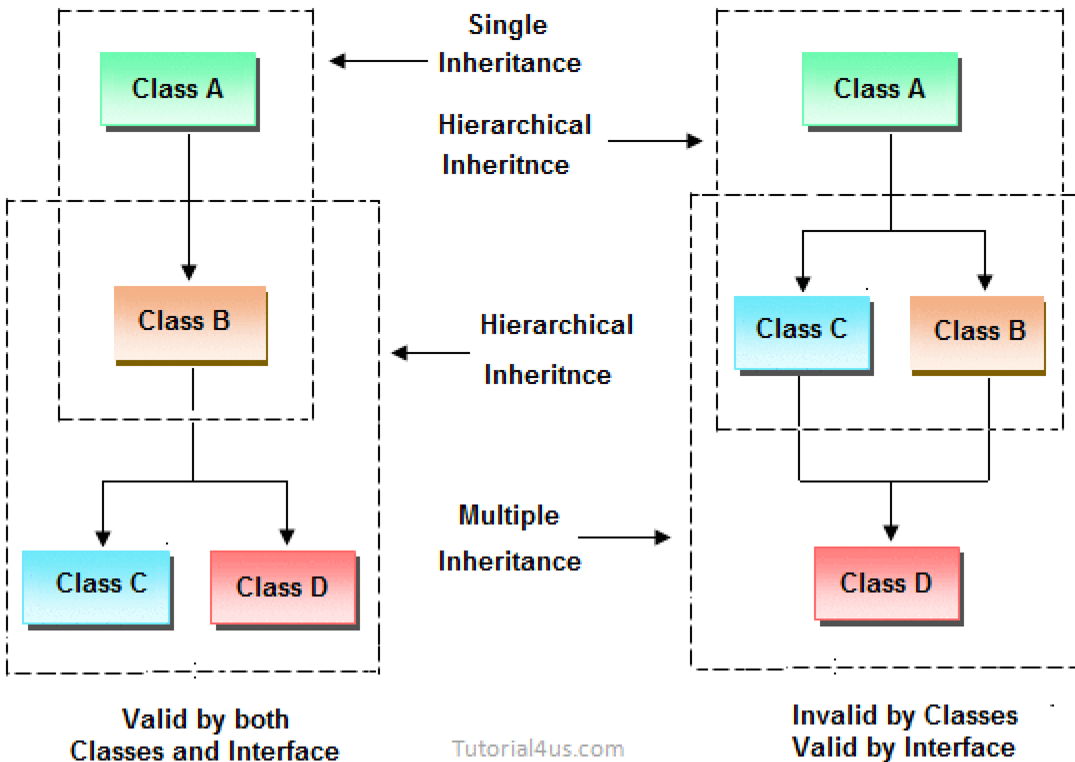
In multiple inheritance there exist multiple classes and singel derived class.



The concept of multiple inheritance is not supported in java through concept of classes but it can be supported through the concept of **interface**

4.Hybrid inheritance

In the combination if one of the combination is multiple inheritance then the inherited combination is not supported by java through the classes concept but it can be supported through the concept of interface.



Creating Multilevel Inheritance

- **When a subclass is derived from a derived class then this mechanism is known as the multilevel inheritance.**
- The derived class is called the subclass or child class for it's parent class and this parent class works as the child class for it's just above (parent) class.
- Multilevel inheritance can go up to any number of level.

```
class A
{
      int x;
      int y;
      int get(int p, int q)
      {
          x=p;
          y=q;
          return(0);
      }
      void Show()
      {
          System.out.println(x);
      }
}
```

```
class B extends A
{
      void Showb()
      {
           System.out.println("B");
      }
}

class C extends B
{
      void display()
      {
           System.out.println("C");
      }
      public static void main(String args[])
      {
           A a = new A();
             a.get(5,6);
             a.Show();
      }
}
OUTPUT
5
```

super keyword

- The super is java keyword. As the name suggest super is used to access the members of the super class. It is used for two purposes in java.
- The first use of keyword super is to access the hidden data variables of the super class hidden by the sub class.

Example: Suppose class A is the super class that has two instance variables as int a and float b. class B is the subclass that also contains its own data members named a and b. then we can access the super class (class A) variables a and b inside the subclass class B just by calling the following command.

super.member;

- Here member can either be an instance variable or a method. This form of super most useful to handle situations where the local members of a subclass hides the members of a super class having the same name. The following example clarifies all the confusions.

Example:

```
class A
{
        int a;
        float b;
        void Show()
        {
                System.out.println("b in super class: " + b);
        }

}
class B extends A
{
        int a;
        float b;
        B( int p, float q)
        {
              a = p;
              super.b = q;
        }
        void Show()
        {
              super.Show();
              System.out.println("b in super class: " + super.b);
              System.out.println("a in sub class:    " + a);
        }
}

 class Mypgm
{
         public static void main(String[] args)
         {
                        B subobj = new B(1, 5);
                           subobj.Show();
         }
}
OUTPUT
b in super class: 5.0
b in super class: 5.0
a in sub class:  1
```

Use of super to call super class constructor: The second use of the keyword super in java is to call super class constructor in the subclass. This functionality can be achieved just by using the following command.

super(param-list);

- Here parameter list is the list of the parameter requires by the constructor in the super class. super must be the first statement executed inside a super class constructor. If we want to call the default constructor then we pass the empty parameter list. The following program illustrates the use of the super keyword to call a super class constructor.

Example:

```
class A
{
        int a;
        int b;
        int c;
       A(int p, int q, int r)
    {
            a=p;
            b=q;
            c=r;
     }
}

class B extends A
{
           int d;
          B(int l, int m, int n, int o)
          {
               super(l,m,n);
               d=o;
          }

         void Show()
```

```
            {
                    System.out.println("a = " + a);
                    System.out.println("b = " + b);
                    System.out.println("c = " + c);
                    System.out.println("d = " + d);
             }
}
class Mypgm
{
          public static void main(String args[])
          {
                  B b = new B(4,3,8,7);
                    b.Show();
          }
 }
OUTPUT
a = 4
b = 3
c = 8
d = 7
```

When constructors are called

- Constructors are called in order of derivation ,from super class to subclass ,
- Super() must be the first statement executed in a subclass constructor.
- If super() is not used ,then the default constructor of each super class will be executed.

EXAMPLE

```
   Class A{
A()
{
System.out.println("Inside A's constructor");
}
class B extends A{
B(){
System.out.println("Inside B's constructor");
    }
}
Class C extends B{
C()  {
System.out,println("Inside C's constructor");
   }
}
Class CallingCons{
Public static void mai(String args[])
{
  C c=new C();
}
}
```

Output:

Inside A's constructor
Inside B's constructor
Inside C's constructor

Method Overriding

- Method overriding in java means a subclass method overriding a super class method.
- Superclass method should be non-static. Subclass uses extends keyword to extend the super class. In the example **class B** is the sub class and **class A** is the super class. **In overriding methods of both subclass and superclass possess same signatures**. Overriding is used in modifying the methods of the super class. In overriding return types and constructor parameters of methods should match.

Below example illustrates method overriding in java.

Example:

```
class A
{
        int i;
        A(int a, int b)
        {
            i = a+b;
        }
        void add()
        {
            System.out.println("Sum of a and b is: " + i);
        }
}
class B extends A
{
        int j;
        B(int a, int b, int c)
        {
            super(a, b);
            j = a+b+c;
        }
        void add()
        {
            super.add();
            System.out.println("Sum of a, b and c is: " + j);
        }
}
class MethodOverriding
{
        public static void main(String args[])
        {
            B b = new B(10, 20, 30);
            b.add();
        }
}
```

OUTPUT

```
Sum of a and b is: 30
Sum of a, b and c is: 60
```

Method Overloading

- Two or more methods have the same names but different argument lists. The arguments may differ in type or number, or both. However, the return types of overloaded methods can be the same or different is called **method overloading**. An example of the method overloading is given below:

Example:

```
class MethodOverloading
{
               int add( int a,int b)
              {
                   return(a+b);
              }
             float add(float a,float b)
             {
                  return(a+b);
             }
            double add( int a, double b,double c)
            {
                 return(a+b+c);
            }
}
class MainClass
{
            public static void main( String arr[] )
             {
                    MethodOverloading mobj = new MethodOverloading();
                    System.out.println(mobj.add(50,60));
                    System.out.println(mobj.add(3.5f,2.5f));
                    System.out.println(mobj.add(10,30.5,10.5));
             }
}
OUTPUT
110
6.0
51.0
```

Dynamic method dispatch

- Dynamic method dispatch is the mechanism by which a call to an overridde method at run time rather than compile time.
- It is important because this is how java implements run –time polymorphism
- A super class reference variable can refer to a subclass object.Java uses this fact to resolve calls to overridden methods at run time.
- When an overridden method to execute based upon the type of the object being referred to at the time the call occurs.
- If a super class contains a method that is overridden by a subclass ,then when different types of object are referred to through a superclass reference variable ,different version of the method are executed.

Here is an example thet illustrates dynamic method dispatch

```
Class A
{
Void callme() {
System.out.printnl("Inside A's callme method");
}
}
Class B extends A{
//override callme()
Void callme() {
System.out.printl("Inside B's callme method");
}
}
Class C extends A {
//override callme()
Void callme() {
System.out.println("Inside C's callme method");
}
}
Class Dispatch {
Public static void main(String args[ ] ) {
A a =new A();
```

```
B b=new B();
C c=new C();
A r;
r=a;        //r refers to an A object
   r.callme(); //calls A's version of callme

   r=b;        //r refers to an B object
   r.callme(); //calls B's version of callme
   r=c;        //r refers to an C object
   r.callme(); //calls C's version of callme
```

OUTPUT:

Inside A's callme method
Inside B's callme method
Inside C's callme method

Abstract Class

- **abstract keyword** is used to make a class abstract.
- Abstract class can't be instantiated with new operator.
- We can use abstract keyword to create an abstract method; an abstract method doesn't have body.
- If classes have abstract methods, then the class also needs to be made abstract using abstract keyword, else it will not compile.
- Abstract classes are used to provide common method implementation to all the subclasses or to provide default implementation.

Example Program:

```
abstract Class AreaPgm
{
        double dim1,dim2;
        AreaPgm(double x,double y)
        {
                dim1=x;
                dim2=y;
        }
```

```
        abstract double area();
}
  class rectangle extends AreaPgm
  {
        rectangle(double a,double b)
        {
                super(a,b);
        }
        double area()
        {
                System.out.println("Rectangle Area");
                return dim1*dim2;
        }
}
  class triangle extends figure
  {
        triangle(double x,double y)
        {
                super(x,y);
```

```
        }
        double area()
        {
                System.out.println("Traingle Area");
                return dim1*dim2/2;
        }

}
  class MyPgm
  {
        public static void main(String args[])
        {

        AreaPgm a=new AreaPgm(10,10); // error, AreaPgm is a abstract class.

                rectangle r=new rectangle(10,5);
                System.out.println("Area="+r.area());

                triangle t=new triangle(10,8);
                AreaPgm ar;
                ar=obj;
                System.out.println("Area="+ar.area());
        }

}
```

final Keyword In Java

The **final keyword** in java is used to restrict the user. The final keyword can be used in many context. Final can be:

1. variable
2. method
3. class

1) final variable: If you make any variable as final, you cannot change the value of final variable(It will be constant).

Example: There is a final variable speedlimit, we are going to change the value of this variable, but It can't be changed because final variable once assigned a value

can never be changed.

```
class Bike
  {
         final int speedlimit=90;//final variable
        void run()
        {
                speedlimit=400;
         }
}
  Class MyPgm
  {
         public static void main(String args[])
        {
                Bike obj=new Bike();
               obj.run();
         }
}
```

Output:Compile Time Error

---

2) final method: If you make any method as final, you cannot override it.

Example:

```
class Bike
  {
        final void run()
        {
               System.out.println("running");
        }
}
  class Honda extends Bike
  {
        void run()
        {
               System.out.println("running safely with 100kmph");
        }
```

```
}
```

```
Class MyPgm
{
        public static void main(String args[])
        {
                Honda honda= new Honda();
                honda.run();
        }
}
```

Output:Compile Time Error

---

3) final class: If you make any class as final, you cannot extend it.

Example:

```
final class Bike
{

}

class Honda extends Bike
{
        void run()
        {
                System.out.println("running safely with 50kmph");
        }
}

Class MyPgm
{

        public static void main(String args[])
        {
                Honda honda= new Honda();
                        honda.run();
        }
}
```

Output:Compile Time Error

The object class

- There is one special class ,**Object** definedvy 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 thet 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 ru 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 executio

Packages and Interfaces in JAVA

- ✓ 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.

The **package keyword** is used to create a package in java.

```
//save as
Simple.java
package mypack;
public class Simple
{
        public static void main(String args[])
       {
                System.out.println("Welcome to package");
        }
}
```

How to access package from another package?

There are three ways to access the package from outside the package.

1. import package.*;
2. import package.classname;
3. fully qualified name.

1) Using packagename.*

If you use package.* then all the classes and interfaces of this package will be accessible but not subpackages.

The import keyword is used to make the classes and interface of another package accessible to the current package.

Example of package that import the packagename.*

```
//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
```

---

2) Using packagename.classname

If you import package.classname then only declared class of this package

will be accessible.

Example of package by import package.classname

```
//save by A.java

package pack;
public class A
{
 public void msg(){System.out.println("Hello");



}
}

//save       by
B.java package
mypack;
import pack.A;

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

Output:Hello

3) Using Fully Qualified name

If you use fully qualified name then only declared class of this package will be accessible. Now there is no need to import. But you need to use fully qualified name every time when you are accessing the class or interface.

It is generally used when two packages have same class name e.g. java.util and java.sql packages contain Date class.

Example of package by import fully qualified name

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

//save        by
B.java package
mypack;   class
B
{
        public static void main(String args[])
```

```
        {
                pack.A obj = new pack.A();//using fully qualified name
                obj.msg();
        }
}
Output:Hello
```

Access Modifiers/Specifiers

The access modifiers in java specify 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.

2) default access modifier

If you don't use any modifier, it is treated as **default** by default. The default modifier is accessible only within 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.

4) public access modifier

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

Understanding all java access modifiers by a simple table.

Access Modifier	**within class**	**within package**	**outside package by subclass only**	**outside package**
Private	Y	N	N	N
Default	Y	Y	N	N
Protected	Y	Y	Y	N
Public	Y	Y	Y	Y

Interface in java

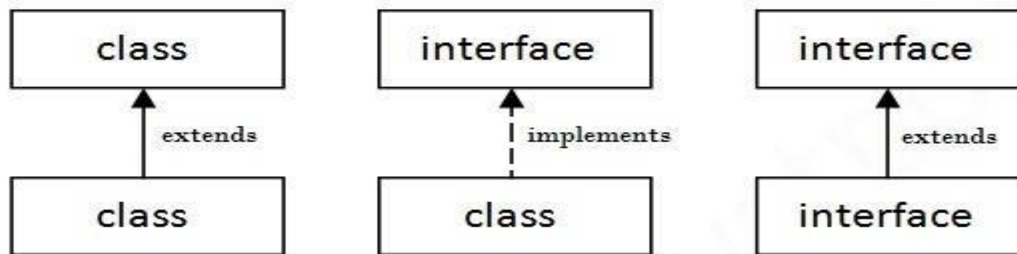
- ✓ An **interface in java** is a blueprint of a class. It has static final variables and abstract methods.
- ✓ The interface in java is **a mechanism to achieve abstraction**. There can be only abstract methods in the java interface does not contain method body. It is used to achieve abstraction and multiple inheritance in Java.
- ✓ It cannot be instantiated just like abstract class.
- ✓ Interface fields are public, static and final by default, and methods are public and abstract.

There are mainly three reasons to use interface. They are given below.

- It is used to achieve abstraction.
- By interface, we can support the functionality of multiple inheritance.

Understanding relationship between classes and interfaces

As shown in the figure given below, a class extends another class, an interface extends another interface but a **class implements an interface**.



Example 1

In this example, Printable interface has only one method, its implementation is provided in the Pgm1 class.

```
interface printable
{
        void print();
}
class Pgm1 implements printable
{
        public void print()
        {
                System.out.println("Hello");
        }
}
 class IntefacePgm1
{
        public static void main(String args[])
        {
                Pgm1 obj = new Pgm1
                (); obj.print();
         }
}
```

Output

Hello

Example 2

In this example, Drawable interface has only one method. Its implementation is provided by Rectangle and Circle classes. In real scenario, interface is defined by someone but implementation is provided by different implementation providers. And, it is used by someone else. The implementation part is hidden by the user which uses the interface.

```
//Interface  declaration:  by  first

user interface Drawable
{
        void draw();
}
//Implementation: by second user

class Rectangle implements Drawable
{
        public void draw()
        {
                System.out.println("drawing rectangle");
        }
}

class Circle implements Drawable
{
        public void draw()
        {
                System.out.println("drawing circle");
        }
}
```

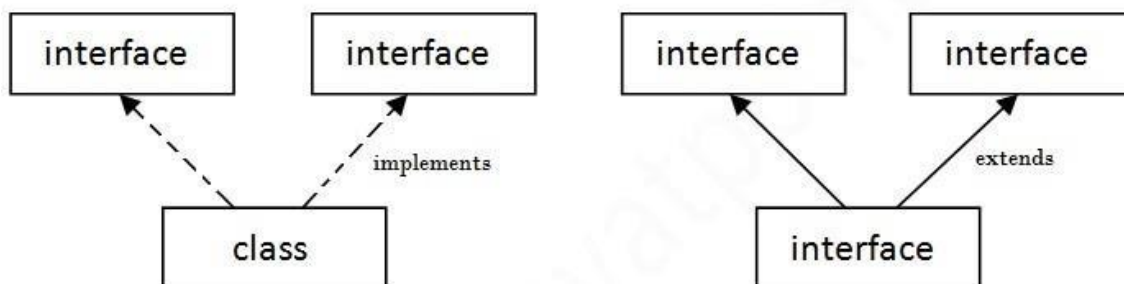
```
//Using interface: by third
user class TestInterface1
{
        public static void main(String args[])
        {
                //In real scenario, object is provided by method e.g.
                getDrawable()
                Drawable d=new Circle();

                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.



Multiple Inheritance in Java

Example

```
interface Printable
{
        void print();
}

interface Showable
{
        void show();
}
class Pgm2 implements Printable,Showable
{
        public void print()
        {
                System.out.println("Hello");
        }

        public void show()
        {
                System.out.println("Welcome");
        }
}
Class InterfaceDemo
{
        public static void main(String args[])
        {
                Pgm2 obj = new Pgm2 ();
                obj.print();
                obj.show();
        }
}
```

Output:
Hello
welcome

Multiple inheritance is not supported through class in java but it is possible by interface, why?

- ✓ As we have explained in the inheritance chapter, multiple inheritance is not supported in case of class because of ambiguity.
- ✓ But it is supported in case of interface because there is no ambiguity as implementation is provided by the implementation class. For example:

Example

```
interface Printable
{
	void print();
}

interface Showable
{
	void print();
}

class InterfacePgm1 implements Printable, Showable
{
	public void print()
	{
		System.out.println("Hello");
	}
}
class InterfaceDemo
{
	public static void main(String args[])
	{
		InterfacePgm1 obj = new
```

```
                InterfacePgm1 (); obj.print();
        }
}
```

Output
:
 Hello

As you can see in the above example, Printable and Showable interface have same methods but its implementation is provided by class TestTnterface1, so there is no ambiguity.

Interface inheritance

- ✓ A class implements interface but one interface extends another

```
interface. interface Printable
{
        void print();
}
interface Showable extends  Printable
{
        void show();
}
class InterfacePgm2 implements Showable
{
        public void print()
        {
                System.out.println("Hello");
        }
        public void show()
        {
                System.out.println("Welcome");
        }
Class InterfaceDemo2
{
        public static void main(String args[])
        {
                InterfacePgm2 obj = new
                InterfacePgm2 (); obj.print();
                obj.show();
```

```
        }
}
```

Output:

```
Hello
welcome
```

Program to implement Stack

```
public class StackDemo
{
            private static final int
            capacity = 3; int arr[] = new
            int[capacity];
            int top = -1;

    public void push(int pushedElement)
    {
            if (top < capacity - 1)
            {
                    top++;
                    arr[top] = pushedElement;
            System.out.println("Element " + pushedElement + " is pushed to
            Stack !")
    ;
            printElements();
            }
            else
             {
                    System.out.println("Stack Overflow !");
            }
    }

    public void pop()
    {
            if (top >= 0)
             {
                                    }
                                                    }
            }
            els
            e
            {
```

```
top--;
System.out.prin
tln("Pop
operation done        System.out.println("Stack Underflow !");
!");


            public void printElements()
            {
                    if (top >= 0)
                    {
                            System.out.println("Elements in
                            stack :"); for (int i = 0; i <= top; i++)
                            {
                                  System.out.println(arr[i]);
                            }


                    }
            }
class MyPgm
    {
            public static void main(String[] args)
            {
                        StackDemo stackDemo = new
                                StackDemo();


                    stackDemo.pop();
                    stackDemo.push(23);
                    stackDemo.push(2);
                    stackDemo.push(73);
                    stackDemo.push(21);
                    stackDemo.pop();
                    stackDemo.pop();
                    stackDemo.pop();
                    stackDemo.pop();
            }


    }
```

Output

```
Stack Underflow !
Element 23 is pushed to Stack !
Elements in stack :
23
Element 2 is pushed to Stack !
Elements in stack :
23
2
Element 73 is pushed to Stack !
Elements in stack :
23
2
73
Stack Overflow !
Pop operation done !
Pop operation done !
Pop operation done !
Stack Underflow !
```

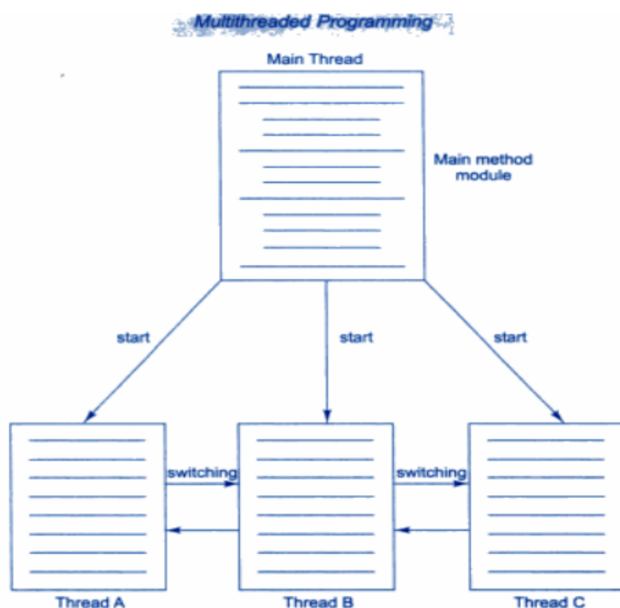
Syllabus:

Multithreading: The Java Thread Model, The Main Method, Creating a Thread, Creating Multiple Thread, Using isAlive() and join(),Thread priorities,Synchronization,Interthread Communication,Suspending,Resuming and Stopping Threads, Using Multithreading

Module 3: Multithreading and Event handling

1. Multithreading programming :

- Multithreading is a conceptual programming paradigm where a program is divided into two or more subprogram, which can be implemented at the same time in parallel.
- This is something similar to dividing a task into subtask and assigning them to processor for execution independently and simultaneously.
- Multithreading is a specialized form of multitasking. In process-based multitasking, a program is the smallest unit of code that can be dispatched by the scheduler.
- In a *thread-based* multitasking environment, the thread is the smallest unit of dispatchable code. This means that a single program can perform two or more tasks simultaneously.



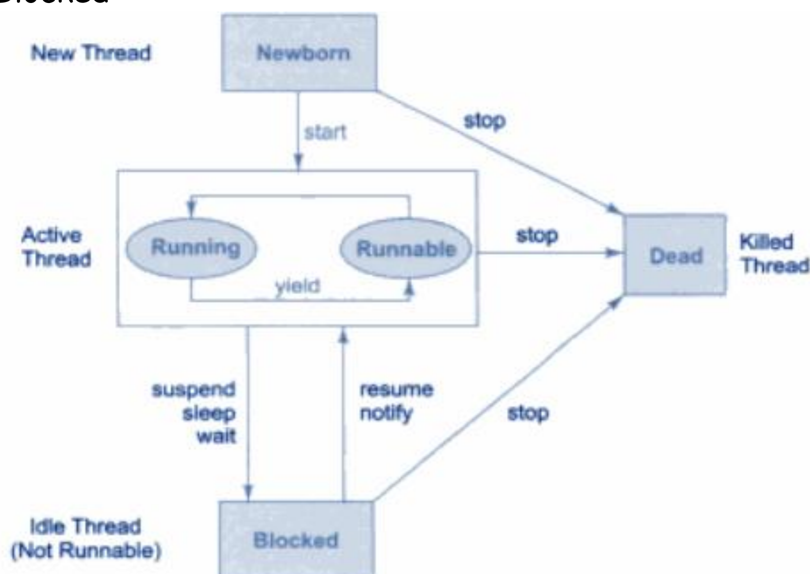
2 .**Thread:**

- The small unit of program or sub module is called as thread.
- Each thread defines a separate path of execution
- Threads that do things like memory management and signal handling but from the application programmer's point of view, you start with just one thread, called the main thread.
- Main thread has the ability to create additional threads.

3. Life cycle of thread:

Thread can enter into different state during life of thread, different stages in thread are as follows:

- New born
- Runnable
- Running
- Dead
- Blocked



New Born state:

- When we create the thread class object, then thread is said to be born and move to new born state. But still thread is not under running state.

- Thread can be moved to either one of two state as follows:
- Thread can move from born state to dead state when we invoke stop() method.
- Thread can move from born state to runnable state when we invoke start() method.

Runnable state:

- Runnable state means thread is ready for execution and waiting for the processor to free.
- All thread are joined in queue and waiting for execution.
- If all the threads have equal priority, Then they are given a time slot for execution in round robin fashion ie. FCFS fashion.

Running state:

Running state means, Thread is under execution. Thread will run until its relinquish control on its own or its is preempted by the higher priority thread.

Blocked state:

- Thread is said to be blocked when it is pre-empted from entering into runnable state and subsequently the running state.
- This can be happen when thread is suspend, wait, sleep in order to satisfy certain requirements.

Dead State:

- Thread can be killing as soon as its born state by calling stop() method.
- Thread will automatically kill, as soon as its completed the operation

3.Creating thread:

Thread can be created in two ways:

- By creating a thread class(**Extending Thread class**)
- By converting a class to a Thread Class(**Implementing Runnable interface**)

1.By Creating a Thread class(Extending Thread class):

- Declare a class by extending Thread Class
- implement the run() method,Where run() method is the override method in order to implement the actual code to be executed by Thread.
- Create a thread object and call a start method to initiate the thread execution

Syntax:

```
class class_name extends Thread
{
public void run()
{
/*Implement actual code*/
}
public static void main(String ar[])
{
class_name object=new class_name();
object.start();
}
}
```

Program:

```
class A extends Thread
{
public void run()
{
for(int i=0;i<10;i++)
System.out.println("A class="+i);
}
}
class B extends A
{
public void run()
{
```

```
for(int i=0;i<10;i++)
System.out.println("class B="+i);
}
public static void main(String ar[])
{
A a1=new A();
B b1=new B();
a1.start();
b1.start();
}
}
```

2.By converting a class to a Threadable Class(Implementing Runnable interface)

- Runnable interface declare the run() method that is required for implementing thread in our program. The following are the steps are taken to implement the runnable interface.
- Declare the class by implementing Runnable interface
- Implement the run() method.
- Crteate a thred by defining an object that is instantiated from this Runnabel class as the target of that thread
- Call the start() method to run the thread.

Syntax:

```
class Class_Name implements Runnable
{
public void run()
{
/* Implements operation */
}
public static void main(String ar[])
{
Class_Name object=new Class_Name();
Thread object1=new Thread(object);
```

```
object1.start();
}
}
```

Program:

```
class A implements Runnable
{
public void run()
{
for(int i=0;i<10;i++)
System.out.println("Class A="+i);
}
}
class B implements Runnable
{
public void run()
{
for(int i=0;i<10;i++)
System.out.println("class B="+i);
}
public static void main(String ar[])
{
A a1=new A();
B b1=new B();
Thread t1=new Thread(a1);
Thread t2=new Thread(b1);
t1.start();
t2.start();
}
}
```

<u>5. Threads methods</u>

The different threads methods are as follows:

Yield(),stop(),suspend(),resume(),wait(),notify(),notifyall().

1.yield()

Calling **yield()** will move the current thread from running to runnable, to give other threads a chance to execute. However the scheduler may still bring the same thread back to running when processor is free.

Program:

```
class A extends Thread
{
public void run()
{
for(int i=0;i<10;i++)
{
if(i==2) yield();
}
}
}
class B
{
public static void main(String ar[])
{
A a1=new A();
a1.start();
}
}
```

Stop():

When stop() is called then processor will kill thread permanently.It means thread move to dead state.

```
class A extends Thread
{

public void run()
{
for(int i=0;i<10;i++)
{
if(i==2) stop();
}
```

```
}
}

class B
{
public static void main(String ar[])
{
A a1=new A();
a1.start();
}
}
```

Sleep():

- When sleep() is called then processor will stop the execution of thread for the specified amount of time from the execution.
- This static **sleep()** method causes the thread to suspend execution for a given time. The sleep method has two overloaded versions:
  - static void sleep (long milliseconds) throws Interrupted Exception
  - static void sleep (long milliseconds, int nanoseconds) throws Interrupted Exception

Suspend():

- When suspend() is called then processor Sends the calling thread into block state.
- Using resume() method its bring the thread back from block state to running state.

Program for sleep and suspend resume methods:

```
class A extends Thread
{
public void run()
{
for(int i=0;i<10;i++)
{
if(i==2) try { sleep(100);} catch(Exception e){ s.o.p(e) resume();}
}
}
}
class B extends Thread
{
public void run()
{
for(int i=0;i<10;i++)
{
if(i==2) suspend();
}
}
}
class Mainclass
{
public static void main(String ar[])
{
A a1=new A();
B b1=new B();
a1.start();
b1.start();
}
}
```

Thread Priority:

- Each thread assigned a priority,which effects the order in which it is scheduled for running.
- Thread of same priority are given equal treatment by the java scheduler and there for they share the processor on FCFS basis
- Java permits us to set the priority of the thread using setPriority()methods.

Final void setPriority(int level)

- Where level specify the new priority setting for the calling thread. Level is the integer constant as follows:

MAX_PRIORITY

MIN_PRIORITY

NORM_PRIORITY

- The MAX_priority value is 10,MIN_PRIORITY values is 1 And NORM_PRIORITY is the default priority whose value is 5.
- We can also obtain the current priority setting value by calling getPriority() method of thread.

Final int getPriority()

Program:

```
class A extends Thread
{
public void run()
{
for(int i=0;i<10;i++)
{
System.out.println("class a thread="+i);
}
}
}
```

```
class B extends Thread
{
public void run()
{
for(int i=0;i<10;i++)
{
s.o.p("Class b thread="+i);
}
}
class MainThread
{
public static void main(String ar[])
{
A a1=new A();
B b1=new B();
a1.setPriority(Thread.MAX_PRIORITY);
b1.setPriority(Thread.MIN_PRIORITY);
a1.start();
b1.start();
b1.setPriority(a1.getPriority()+10);
}
}
```

Synchronization:

- When two or more thread needs access to the shared resource, they need some way to ensure that the resource will be used by only one thread at a time.
- The process by which this is achieved is called synchronization.
- Key to synchronized is the concepts of monitor or semaphores. A monitor is an object that is used as mutually exclusive lock or mutex. Only one thread can own a monitor at a given time. When one thread acquires a lock it is said to have entered the monitor.

- All other thread attempting to enter the locked monitor will be suspended until the first thread exits the monitor. These other thread are said to be waiting for monitor.
- This can be achieved by using keyword synchronized to method.

Syntax:

```
synchronized void method_name()
{  /* implementation or operation
}
```

Program:

```
class A
{
Synchronized void display()
{
for(int i=0;i<10;i++)
System.out.println("i="+i);
}
}
class B extends Thread
{
public void run()
{
A a1=new A();
System.out.println("class A thread");
For(int i=0;i<10;i++)
a1.display();
}
}
class C extends Thread
{
public void run()
{
A a1=new A();
Sysem.out.println("class B thread");
```

```
For(int i=0;i<10;i++)
a1.display();
}
}
class D
{
public static void main(String ar[])
{
B b1=new B();
C c1=new C();
b1.start();
c1.start();
}
}
```

Inter-thread communication:

- Inter-thread communication can be defined as exchange of message between two or more threads. The transfer of message takes place before or after changes of state of thread.
- The inter-thread communication can be achieved with the help of three methods as follows:

  Wait(),notify() notifyall()

  Wait()- tells the calling thread to give up the monitor and go to sleep mode until some of other thread enters the same monitor and call the notify() methods

  Notify()-wakes up a thread that called wait() method on the same object.

  Notifyall()-wakes up all thread that called wait() methods on the same object.
- Inter-thread communication can be implemented by using the key word as synchronized to methods.
- Different types of inter-thread communication example are as follows:

- Producer-consumer problem
- Reader –writer problem
- Bounded-Buffer problem(Also called as Producer-consumer problem)

Producer-consumer problem/bounded buffer problem:

- Producer thread goes on producing an item unless an until buffer is full.
- Producer thread check before producing an item weather buffer is full or not, if buffer is full producer wait producing an item unless and until consumer thread consume a item.
- Consumer thread goes on consuming an item which is produced by the producer. The consumer thread check weather buffer is empty before consuming. If buffer is empty consumer thread as to wait until producer has to produce an item.

Program:

```
class A
{
int stack[]=new int[10];
int top=-1;
Synchronized void produce(int item)
{
if(top==10)
try
{
 wait();
}
catch(Exception e)
{
System.out.println(e);
```

```
}
Satck[++top]=item;
notify();
}

Synchronized void consume()
{
if(top==-1)
try
{
 wait();
}
catch(Exception e)
{
System.out.println(e);
}
item=Satck[top++];
System.out.println("consumed item is"+item);
notify();
}
}
class Producer extends Thread
{
public void run()
{
A a1=new A();
for(int i=0;i<10;i++)
a1.produce(i);
}
}
class Consumer extends Thread
{
public void run()
{
A a1=new A();
for(int i=0 ;i<12;i++)
a1.consume();
}
```

```
}
class MainThread
{
public static void main(String ar[])
{
Produce p=new Produce();
Consume c=new Consume();
p.start();
c.start();
}
}
```

Reader-writer problem:

- Reader thread reading an item from the buffer, Where as writer thread writing an item to buffer.
- If reader is reading then writer has to wait unless and until reading is finish.
- While writing thread writing an content then no other thread read the content unless and until writing is over.
- This problem can be achieved using wait, notify and nitifyall method and using synchronized keyword to method.

isAlive() and join()

- The final **isAlive()** method returns true if the thread is still running or the Thread has not terminated.
- **final join()**
- The final **join()** method waits until thread on which it is called is terminated. For example, thread1.join() suspends the current thread until thread1 dies.
- The join() method can throw an Interrupted Exception if the current thread is interrupted by another thread.

Program:

```
 class A extends Thread
{

public void run()
{
for(int i=0;i<10;i++)
System.out.println("class A="+i);
}
}
Class B extends Thread
{
public void run()
{
for(int i=0;i<10;i++)
System.out.println("class B="+i);
}
}

class C
{
public static void main(String args[])
{
A a1=new A();
```

```
B b1=new B();
a1.start();
b1.start();
System.out.println(a1.isAlive());
System.out.println(b1.isAlive());
try
{
A1.join();
B1.join();
}
catch(Exception e)
{
System.out.println(e);
}
System.out.println("main thread dead");
}
}
```

Creating multiple thread:

- More than one thread can be created using single object of thread class. Where all the thread can execute parallel.

Program:

```
class A implements Runnable
{
A()
{
Thread t=new Thread(this);
t.start();
}
public void run()
{
```

```
for(int i=0;i<10;i++)
System.out.println(i);
}
}
class MainThread
{
public static void main(String args[])
{
A a1=new A();
A a2=new A();
}
}
```

Syllabus:

Event Handling: Two Event Handling Mechanisms, Delegation Event Model, Event Classes, Sources of Events, Listener Interfaces, Using the Delegation Event Model, Adapter Classes, Inner Classes

APPLETS: Applet Basics, Applet Architecture, An Applet Skeleton, Simple Applet Display Methods ,The AppletHTML Tag,Passing Parameters to Applets,getDocumentBase() and get Codebase(),Applet Context and show Document()

Module 4 : Event Handling

Event Handling

Event Handling is the mechanism that controls the event and decides what should happen if an event occurs. This mechanism has the code which is known as event handler that is executed when an event occurs. Java Uses the Delegation Event Model to handle the events. This model defines the standard mechanism to generate and handle the events. Let's have a brief introduction to this model.

Any program that uses GUI (graphical user interface) such as Java application written for windows, is event driven. Event describes the change of state of any object.

Example: Pressing a button, Entering a character in Textbox.

Two Event Handling Mechanism

- Events are handled by the original version of java (1.0) and modern versions of Java.
- The 1.0 method of event handling is still supported, but it is not recommended for new programs.
- Many of the methods that support the old 1.0 event model have been deprecated.

1.Delegation event model :

- It defines standard and consistent mechanisms to generate and process events. Here the source generates an event and sends it to one or more listeners.
- The listener simply waits until it receives an event. Once it is obtained, It processes this event and returns.
- Listeners should register themselves with a source in order to receive an even notification. Notifications are sent only to listeners that want to receive them.

Components of Event Handling

Event handling has three main components,

Events:

- An event is a change of state of an object. In the delegation model, an *event* is an object that describes a state change in a source.
- It can be generated as a consequence of a person interacting with the elements in a graphical user interface.

Events Source:

- Event source is an object that generates an event.
- This occurs when the interval state of that object changes in some way.Source event generate more than one type of events.

Listeners:

A listener is an object that listens to the event. A listener gets notified when an event occurs.It has two major requirements

- It must have been registered with one or more source to receive notification about specific types of event
- It must implement methods to receive and process these notification

2.Event class:

The classes that represent events are at the core of Java's event handling mechanism.

Event Object: It is at the root of the Java event class hierarchy in **java.util.** It is the super class for all events.

It's one constructor is shown here: Event Object (Object *src)*

Here, *src* is the object that generates this event. Event Object contains two methods: get Source( ) and to String( ). The get Source ( ) method returns the source of the event.

Event Class	**Description**
Action Event	Generated when a button is pressed, a list item is Double-clicked, or a menu item is selected.

Adjustment Event	Generated when a scroll bar is manipulated.
Component Event	Generated when a component is hidden, moved, resized, or becomes visible.
Container Event	Generated when a component is added to or removed From a container.
Focus Event	Generated when a component gains or loses Keyboard focus.
Input Event	Abstract super class for all component input event classes.
Item Event	Generated when a check box or list item is clicked; also occurs when a choice selection is made or a checkable Menu item is selected or deselected.
Key Event	Generated when input is received from the keyboard.
Mouse Event	Generated when the mouse is dragged, moved, clicked, Pressed, or released; also generated when the mouse enters or exits a component.
MouseWheelEvent	Generated when the mouse wheel is moved.
Text Event	Generated when the value of a text area or text field is Changed.
Window Event	Generated when a window is activated, closed, deactivated, deiconified, iconified, opened, or quit.

1.ActionEvent Class

- An **Action Event** is generated when a button is pressed, a list item is double-clicked, or
- a menu item is selected.
- The **ActionEvent** class defines four integer constants that can be used to identify any modifiers associated with an action event: **ALT_MASK,CTRL_MASK, META_MASK,** and **SHIFT_MASK.**
- **ActionEvent** has these three constructors:

  ActionEvent(Object *src*, int *type,* String *cmd)*

  ActionEvent(Object *src,* int *type,* String *cmd,* int *modifiers)*

  ActionEvent(Object *src*, int *type*, String *cmd*, long *when,* int *modifiers)*

- Here, *src* is a reference to the object that generated this event. The type of the event is specified by *type,* and its command string is *cmd.* The argument *modifiers* indicates which modifier keys (ALT, CTRL, META, and/or SHIFT) were pressed when the event was generated. The *when* parameter specifies when the event occurred

2 The AdjustmentEvent Class

An **AdjustmentEvent** is generated by a scroll bar There are five types of adjustment events each defines integer constants that can be used to identify them

BLOCK_DECREMENT-the user clicked inside the scroll bar to decrese its value

BLOCK_INCREMENT- The user clicked inside the scroll bar to increses its value

TRACK-the slider was dragged

UNIT_DECREMENT-The button at the end of scroll bar was clicked to decreses its value

UNIT_INCREMENT-The button at the end of scroll bar was clicked to increses its value

Here is one AdjustmentEvent constructor

AdjustmentEvent(Adjustable src,int id,int type,int data);

Here, *src* is a reference to the object that generated this event. The type of the event is specified by *type and its associated data is data.*

The ***getAdjustable()*** *method returns the object that generated the event.*

getAdjustmentType() *method returns one of the constant defined by the AdjustmentEvent.*

3.ComponentEvents class:

A **ComponentEvent** is generated when the size, position, or visibility of a component is changed. There are four types of component events .There are four integer constanst

COMPONENT_HIDDEN-the component was hidden

COMPONENT_MOVED-the component was moved

COMPONENT_RESIZED-the component was resized

COMPONENT_SHOWN-the component was shown.

There is one constructor

ComponentEvent(Component src,int type);

Here, *src* is a reference to the object that generated this event. The type of the event is specified by *type.*

getComponent() *method returns the component that was generated the event.*

4.ContainerEvent class:

A **ContainerEvent** is generated when a component is added to or removed from a container

Its has two integer constant:

COMPONENT_ADDED-the component has been added to.

COMPONENT_REMOVED-The component has been removed out.

There is one constructor:

ContainerEvent(Component src,int type, component comp);

Here, *src* is a reference to the object that generated this event. The type of the

event is specified by *type and comp is the argument that indicates that component is added.*

getContainer() method generates the event.

getChild() method returns a reference to the component that was added or removed from the container.

5.ItemEvent classs:

An **ItemEvent** is generated when a check box or a list item is clicked or when a checkable menu item is selected or deselected.there are two ineger constants:

DESELECTED-the user deselected an item

SELECTED-user selected an item

One constructor:

ItemEvent(itemSelectable src ,int type,object entry,int state);

Here, *src* is a reference to the object that generated this event. The type of the event is specified by *type.*

getItem()*method can be used to obtain a reference to the item that generated an event.*

getItemSelectable()-*method can be used to obtain a reference to the itemSelectable object that generated an event.*

getStateChange()-*method returns the state change for the event.*

6.KeyEvent class:

A **KeyEvent** is generated when keyboard input occurs.

There is one constructor:

KeyEvent(Component src,int type,long when,int modifier,int code,char ch);

Here, *src* is a reference to the object that generated this event. The type of the event is specified by *type.the system time at which key pressed,modifier argument indicates which modifier were pressed when key event generated.*

getChar() methods returns CHAR_UNDEFINED when a KEY_TYPED event occurs.

getKeyCode() method returns VK_UNDEFINED.

7.MouseEVENT class:

There are eight types of mouse event:

MOUSE_CLICKED-the user clicked the mouse

MOUSE_DRAGGED-the user dragged the mouse

MOUSE_ENTERED-the user entered the mouse

MOUSE_EXITED-the user exit the mouse

MOUSE_MOVED-the user moved the mouse

MOUSE_PRESSED-the user pressed the mouse

MOUSE_RELEASED-the user released the mouse

There is one constructor:

MouseEvent(Component src,int type,long when,int modifier,int x,int y,int click,boolen triggersPopup)

Here, *src* is a reference to the object that generated this event. The type of the event is specified by *type.the system time at which key pressed,modifier argument indicates which modifier were pressed when key event generated.*

8.TextEvent class:

The TextEvent Class Instances of this class describe text events. These are generated by text fields and text areas when characters are entered by a user or program.

There is one constructor:

TextEvent(Object src,int type);

Here, *src* is a reference to the object that generated this event. The type of the event is specified by *type*

9.FocusEvent classs

A **Focus Event** is generated when a component gains or losses input focus. These events
are identified by the integer constants **FOCUS_GAINED** and **FOCUS_LOST**.
Focus Event is a subclass of **Component Event** and has these constructors:

FocusEvent(Component *src*, int *type*)
FocusEvent(Component *src*, int *type*, boolean *temporaryFlag*)
Focus Event(Component *src*, int *type*, boolean *temporaryFlag*, Component *other*)

Here, *src* is a reference to the component that generated this event. The type of the event is specified by *type*. The argument *temporaryFlag* is set to **true** if the focus event is
temporary. Otherwise, it is set to **false**. (A temporary focus event occurs as a result of another user interface operation.

You can determine the other component by calling **getOppositeComponent( )**, shown here.

Component getOppositeComponent( )

The **isTemporary( )** method indicates if this focus change is temporary. Its form is shown here:

boolean isTemporary( )
The method returns **true** if the change is temporary. Otherwise, it returns **false**.

10.InputEvent class

The abstract class **InputEvent** is a subclass of **ComponentEvent** and is the superclass for component input events. Its subclasses are **KeyEvent** and **MouseEvent**.

InputEvent defines several integer constants that represent any modifiers, such as the control key being pressed, that might be associated with the event. Originally, the **InputEvent** class defined the following eight values to represent the modifiers.

ALT_MASK BUTTON2_MASK META_MASK
ALT_GRAPH_MASK BUTTON3_MASK SHIFT_MASK
BUTTON1_MASK CTRL_MASK

To test if a modifier was pressed at the time an event is generated, use the **isAltDown( )**, **isAltGraphDown( )**, **isControlDown( )**, **isMetaDown( )**, and **isShiftDown( )** methods. The forms of these methods are shown here:

boolean isAltDown( )
boolean isAltGraphDown( )
boolean isControlDown( )
boolean isMetaDown( )

boolean isShiftDown( )

You can obtain a value that contains all of the original modifier flags by calling
the **getModifiers( )** method. It is shown here:
int getModifiers( )

You can obtain the extended modifiers by called **getModifiersEx( )**, which is shown here.
int getModifiersEx( )

11.MouseWheelEvent Class

The **MouseWheelEvent** class encapsulates a mouse wheel event. It is a subclass of **MouseEvent** .

If a mouse has a wheel, it is located between the left and right buttons. Mouse wheels are used for scrolling. **MouseWheelEvent** defines these two integer constants.
WHEEL_BLOCK_SCROLL A page-up or page-down scroll event occurred.
WHEEL_UNIT_SCROLL A line-up or line-down scroll event occurred.

MouseWheelEvent defines the following constructor.
MouseWheelEvent(Component *src*, int *type*, long *when*, int *modifiers*,
int *x*, int *y*, int *clicks*, boolean *triggersPopup*,
int *scrollHow*, int *amount*, int *count*)

Here, *src* is a reference to the object that generated the event. The type of the event is specified by *type*. The system time at which the mouse event occurred is passed in *when*. The *modifiers* argument indicates which modifiers were pressed when the event occurred. The coordinates of the mouse are passed in *x* and *y*. The number of clicks the wheel has rotated is passed in *clicks*. The *triggersPopup* flag indicates if this event causes a pop-up menu to appear on this platform. The *scrollHow* value must be either **WHEEL_UNIT_SCROLL** or **WHEEL_BLOCK_SCROLL**. The number of units to scroll is passed in *amount*. The *count* parameter indicates the number of rotational units that the wheel moved.

MouseWheelEvent defines methods that give you access to the wheel event.
To obtain the number of rotational units, call **getWheelRotation( )**, shown here.
int getWheelRotation( )

It returns the number of rotational units. If the value is positive, the wheel moved counterclockwise. If the value is negative, the wheel moved clockwise.

To obtain the type of scroll, call **getScrollType( )**, shown next.
int getScrollType( )

It returns either **WHEEL_UNIT_SCROLL** or **WHEEL_BLOCK_SCROLL**.
If the scroll type is **WHEEL_UNIT_SCROLL**, you can obtain the number of units to scroll by calling **getScrollAmount( )**. It is shown here.
int getScrollAmount( )

The WindowEvent Class

There are ten types of window events. The **Window Event** class defines integer Constants that can be used to identify them. The constants and their meanings are shown here:

WINDOW_ACTIVATED	The window was activated.
WINDOW_CLOSED	The window has been closed.
WINDOW_CLOSING	The user requested that the window be closed.
WINDOW_DEACTIVATED	The window was deactivated.
WINDOW_DEICONIFIED	The window was deiconified.
WINDOW_GAINED_FOCUS	The window gained input focus.
WINDOW_ICONIFIED	The window was iconified.
WINDOW_LOST_FOCUS	The window lost input focus.
WINDOW_OPENED	The window was opened.
WINDOW_STATE_CHANGED	The state of the window changed.

WindowEvent is a subclass of **ComponentEvent**. It defines several constructors. The first is
WindowEvent(Window *src*, int *type*)

Here, *src* is a reference to the object that generated this event. The type of the event is *type.*
WindowEvent(Window *src*, int *type*, int *fromState*, int *toState*)
WindowEvent(Window *src*, int *type*, Window *other*, int *fromState*, int *toState*)

Here, *other* specifies the opposite window when a focus event occurs. The *fromState*

specifies the prior state of the window and *toState* specifies the new state that the window will have when a window state change occurs.

The most commonly used method in this class is **getWindow( )**. It returns the **Window** object that generated the event. Its general form is shown here:
Window getWindow( ).

3.Source Event:

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

Event Source	Description
Button	Generates action events when the button is pressed.
Checkbox	Generates item events when the check box is selected or deselected.
Choice	Generates item events when the choice is changed.
List	Generates action events when an item is double-clicked; generates item events when an item is selected or deselected.
Menu Item	Generates action events when a menu item is selected; generates item events when a checkable menu item is selected or deselected.
Scrollbar	Generates adjustment events when the scroll bar is manipulated.
Text components	Generates text events when the user enters a character.
Window	Generates window events when a window is activated, closed, deactivated, deiconified, iconified, opened, or quit.

4.Event Listeners:

Interface	Description
ActionListener	Defines one method to receive action events.
AdjustmentListener	Defines one method to receive adjustment events.
ComponentListener	Defines four methods to recognize when a component is hidden, moved, resized, or shown.
ContainerListener	Defines two methods to recognize when a component is added to or removed from a container.
FocusListener	Defines two methods to recognize when a component gains or loses keyboard focus.
ItemListener	Defines one method to recognize when the state of an item changes.
KeyListener	Defines three methods to recognize when a key is pressed, released, or typed.
MouseListener	Defines five methods to recognize when the mouse is clicked, enters a component, exits a component, is pressed, or is released.
MouseMotionListener	Defines two methods to recognize when the mouse is dragged or moved.
MouseWheelListener	Defines one method to recognize when the mouse wheel is moved. (Added by Java 2, version 1.4)
TextListener	Defines one method to recognize when a text value changes.
WindowFocusListener	Defines two methods to recognize when a window gains or loses input focus. (Added by Java 2, version 1.4)
WindowListener	Defines seven methods to recognize when a window is activated, closed, deactivated, deiconified, iconified, opened, or quit.

1.ActionListener interface:

This interface define the actionPerformed() method that is invoked when an action event occurs.

void actionPerformed(ActionEvent ae);

2.AdjustmentListener interface:

This interface define the adjustmentValueChanged() method that is invoked when an adjustment event ocuurs.

void adjustmentValueChanged(AdjustmentEvent ae);

3.ComponentListener inetface:

This inetface define four methods that are invoked when a component is resized,moved,shown etc.

void componentResized(ComponentEvent ce);

void componentMoved(ComponentEvent ce);

void componentShown(ComponentEvent ce);

void componentHidden(ComponentEvent ce);

4.ItemListener inetface:

This interface define the itemStateChanged() method that is invoked when the state of an item changed.

void itemStateChanged(ItemEvent ie);

5.KeyListener interface:

This interface define three method,when key is ressed,released.

void keyPressed(KeyEvent ke);

void keyRelesed(KeyEvent ke);

void keyTyped(KeyEvent ke);

5.MouseListener interface:

This inetface define five methds

void mouseClicked(MouseEvent me);

void mouseEneterd(MouseEvent me);

void mouseExited(MouseEvent me);

void mousePressed(MouseEvent me);

void mouseReleased(MouseEvent me);

Program for handling keyboard events

```
import java.awt.*;

import java.awt.event.*;

import java.applet.*;

import java.applet.*;

import java.awt.event.*;

import java.awt.*;


public class Test extends Applet implements KeyListener
{
String msg="";
```

```
public void init()
{
addKeyListener(this);
}
public void keyPressed(KeyEvent k)
{
showStatus("KeyPressed");
}
public void keyReleased(KeyEvent k)
{
System.out.println("KeyRealesed");
}
public void keyTyped(KeyEvent k)
{
msg = msg+k.getKeyChar();
repaint();
}
public void paint(Graphics g)
{
g.drawString(msg, 20, 40);
}}
```

5.Adapter class:

Adapters are abstract classes for receiving various events. The methods in these classes are empty. These classes exists as convenience for creating listener objects.

AWT Adapters:

Following is the list of commonly used adapters while listening GUI events in AWT.

Sr. No.	Adapter class	Description
1	FocusAdapter	An abstract adapter class for receiving focus events.
2	KeyAdapter	An abstract adapter class for receiving key events.
3	MouseAdapter	An abstract adapter class for receiving mouse events.
4	MouseMotionAdapter	An abstract adapter class for receiving mouse motion events.
5	WindowAdapter	An abstract adapter class for receiving window events.

Program :

```
class A extends Applet
{
public void init()
{
addMoseListener(new B(this));
}
}
class B extends MouseAdater
{
A  a1;
B( A a1)
{
this.a1=a1;
}
public void mouseClicked(MouseEvent me)
{
a1.showStatus("mouse clicked");
} }
```

6.Inner class/nested class:

- Class within other class is called nested class or inner class.
- Inner class is the member of outer class
- Outer class can access all the member of inner class, where as inner class cannot access the outer class member.

Program:

```
class A extends Applet
{
public void inti()
{
```

```
addMoseListener(new B(this));
}
class B extends MouseAdater
{
public void mouseClicked(MouseEvent me)
{
a1.showStatus("mouse clicked");
}
}
}
```

APPLETS

The Applet Introduction:

- ✓ Applets are small Java program/applications that are accessed on an Internet Server, transported over the Internet, automatically installed, and run as part of a Web document

- ✓ An applet is a program written in the Java programming language that can be included in an HTML page, much in the same way an image is included in a page. When you use a Java technology enabled browser to view a page that contains an applet, the applet's code is transferred to your system and executed by the browser's Java Virtual Machine (JVM)

Two Types of Applets

There are two varieties of applets. They are

1. Based on the Applet class: **Applet**
2. Based on the Swing Class Applet: **JApplet**

1. Based on the Applet class.

- ➢ These Applet uses the Abstract Window Toolkit(AWT) to provide the graphical user interface.
- ➢ This type of applet has been widely available since java was first created.

2. Based on the Swing Class Applet.

- ➢ This applet uses the swing class to provide GUI.
- ➢ Swing offers a rich and easier to use interface than AWT.

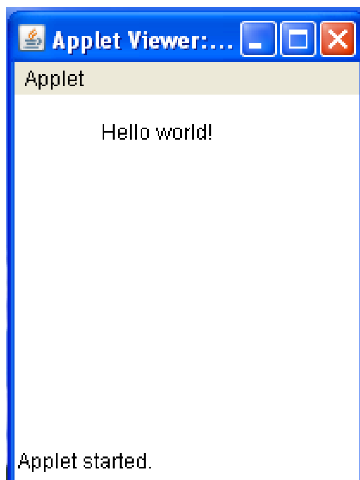
- Swing based applets are the most popular in practice.

Applet Basics

- ✓ The reason people are excited about Java as more than just another OOP language is because it allows them to write interactive applets on the web. Hello World isn't a very interactive program, but let's look at a webbed version.

```
import java.applet.Applet;
import java.awt.Graphics;


public class HelloWorldApplet extends Applet
{
      public void paint(Graphics g)
      {
            g.drawString("Hello world!", 50, 25);
      }
}
```

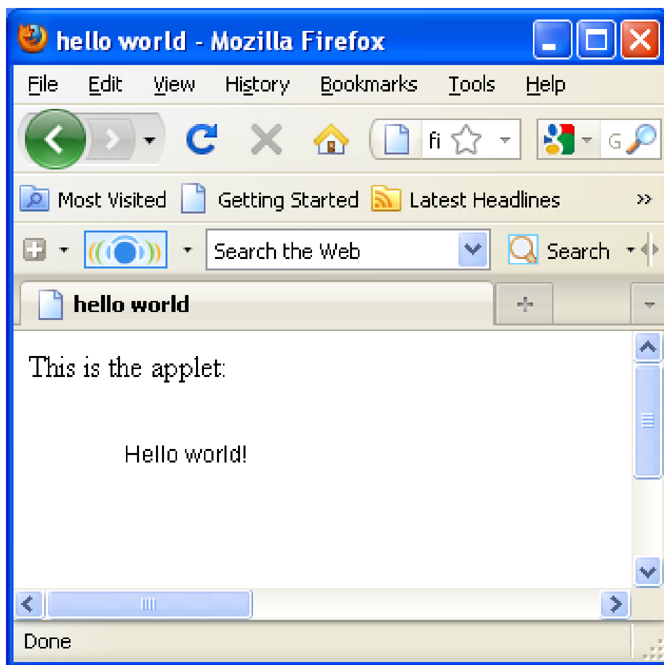
OUTPUT



- ✓ The applet version of HelloWorld is a little more complicated than the HelloWorld application, and it will take a little more effort to run it as well.
- ✓ First type in the source code and save it into file called HelloWorldApplet.java. Compile this file in the usual way. If all is well a file called HelloWorldApplet.class will be created. Now you need to create an HTML file that will include your applet. The following simple HTML file will do.

```
<html>
<head>
<title> hello world </title>
</head>

<body>
This is the applet:<P>
<applet code="HelloWorldApplet" width="150" height="50">
</applet>
</body>
</html>
```

OUTPUT



- ✓ Save this file as HelloWorldApplet.html in the same directory as the HelloWorldApplet.class file. When you have done that, load the HTML file into a Java enabled browser and see the output in the browser window.
- ✓ If the applet compiled without error and produced a HelloWorldApplet.class file, and yet you don't see the string "Hello World" in your browser chances are that the .class file is in the wrong place. Make sure HelloWorldApplet.class is in the same directory as HelloWorldApplet.html. Also make sure that your browsers support Java or that the Java plugin has been installed. Not all browsers support Java out of the box.

The Applet Class

- ✓ An applet is a small program that is intended not to be run on its own, but rather to be embedded inside another application.
- ✓ The **Applet class** must be the super class of any applet that is to be embedded in a Web page or viewed by the Java Applet Viewer. The Applet class provides a standard interface between applets and their environment.

Method	Summary
Void **destroy**()	Called by the browser or applet viewer to inform this applet that it is being reclaimed and that it should destroy any resources that it has allocated.
AccessibleContext **getAccessibleContext**()	Gets the AccessibleContext associated with this Applet.
AppletContext **getAppletContext**()	Determines this applet's context, which allows the applet to query and affect the environment in which it runs.
String **getAppletInfo**()	Returns information about this applet.
AudioClip **getAudioClip**(URL url)	Returns the AudioClip object specified by the URL argument.
AudioClip **getAudioClip**(URL url,	Returns the AudioClip object specified by the URL and name arguments.

String name)	
URL **getCodeBase**()	Gets the base URL.
URL **getDocumentBase**()	Returns an absolute URL naming the directory of the document in which the applet is embedded.
Image **getImage**(URL url)	Returns an Image object that can then be painted on the screen.
Image **getImage**(URL url, String name)	Returns an Image object that can then be painted on the screen.
Locale **getLocale**()	Gets the Locale for the applet, if it has been set.
String **getParameter**(String name)	Returns the value of the named parameter in the HTML tag.
String[][] **getParameterInfo**()	Returns information about the parameters than are understood by this applet.
Void **init**()	Called by the browser or applet viewer to inform this applet that it has been loaded into the system.
Boolean **isActive**()	Determines if this applet is active.
static AudioClip **newAudioClip**(URL url)	Get an audio clip from the given URL.
Void **play**(URL url)	Plays the audio clip at the specified absolute URL.
Void **play**(URL url, String name)	Plays the audio clip given the URL and a specifier that is relative to it.
Void **resize**(Dimension d)	Requests that this applet be resized.
Void **resize**(int width, int height)	Requests that this applet be resized.
Void **setStub**(AppletStub stub)	Sets this applet's stub.
Void	Requests that the argument string be displayed in the

showStatus(String msg)	"status window".
void **start**()	Called by the browser or applet viewer to inform this applet that it should start its execution.
void **stop**()	Called by the browser or applet viewer to inform this applet that it should stop its execution.

Applet Architecture

✓ An applet is a window-based program, its architecture different from the console-based programs. There are two key concepts to understand the architecture they are

1. Applets are Event driven

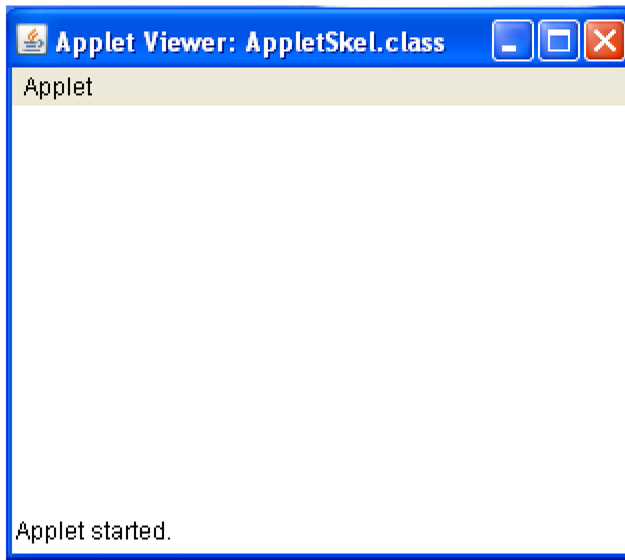
- An applet waits until an event occurs.
- The *AWT* notifies the applet about an event by calling event handler that has been provided by the applet.The applet takes appropriate action and then quickly return control to *AWT*
- All *Swing* components descend from the AWT Container class

2. User initiates interaction with an Applet (*and not the other way aroUND*)

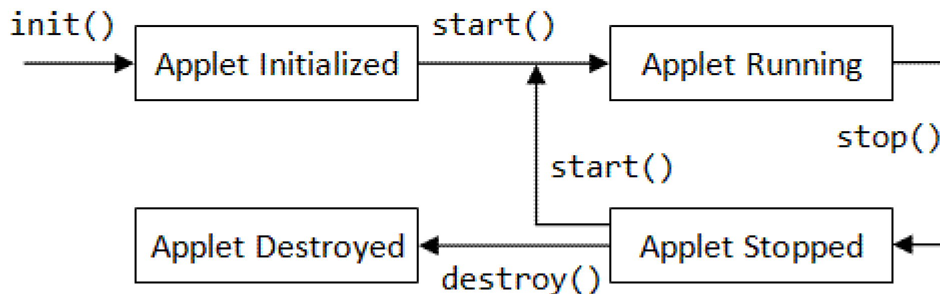
An Applet Skelton

```
// An Applet skeleton.
import java.awt.*;
import javax.swing.*;
/*
<applet code="AppletSkel" width=300 height=100>
</applet>
*/

public class AppletSkel extends JApplet
{
    // Called first.
     public void init()
     {
        // initialization
     }

  /* Called second, after init(). Also called whenever the applet is restarted. */
     public void start()
     {
         // start or resume execution
      }

  // Called when the applet is stopped.
      public void stop()
      {
              // suspends execution
       }

 /* Called when applet is terminated. This is the last method executed. */
      public void destroy()
      {
        // perform shutdown activities
      }

  // Called when an applet's window must be restored.
     public void paint(Graphics g)
     {
       // redisplay contents of window
     }
}
```

OUTPUT



Applet Initialization and Termination/Applet Life Cycle



It is important to understand the order in which the various methods shown in the skeleton are called. **When an applet begins**, the AWT calls the following **initialization methods**, in

this sequence:

1. **init( )**
2. **start( )**
3. **paint( )**

When an applet is terminated, the following sequence of method calls takes place:

1. **stop( )**
2. **destroy( )**

1. init( )

The **init( )** method is the first method to be called. This is where you should initialize variables. This method is called only once during the run time of your applet.

2. start( )

The **start( )** method is called after **init( )**. It is also called to restart an applet after it has been stopped. Whereas **init( )** is called once—the first time an applet is loaded **start( )** is called each time an applet's HTML document is displayed onscreen. So, if a user leaves a web page and comes back, the applet resumes execution at **start( )**.

3. paint( )

The **paint( )** method is called each time your applet's output must be redrawn. **paint( )** is also called when the applet begins execution. Whatever the cause, whenever the applet must redraw its output, **paint( )** is called.

The **paint( )** method has one parameter of type **Graphics**. This parameter will contain the graphics context, which describes the graphics environment in which the applet is running. This context is used whenever output to the applet is required.

4. stop( )

The **stop( )** method is called when a web browser leaves the HTML document containing the applet when it goes to another page, for example. When **stop( )** is called, the applet is probably running. You should use **stop( )** to suspend threads

that don't need to run when the applet is not visible. You can restart them when **start( )** is called if the user returns to the page.

5. destroy( )

The **destroy( )** method is called when the environment determines that your applet needs to be removed completely from memory. At this point, you should free up any resources the applet may be using. The **stop( )** method is always called before **destroy()**.

Simple Applet display methods

```
import java.applet.Applet;
import java.awt.Graphics;

 public class HelloWorldApplet extends Applet
 {
         public void paint(Graphics  g)
         {
                 g.drawString("Hello world!", 50, 25);
         }
}
```

- ✓ Consider the above program to output a string to an applet, use drawString() this is a member of the Graphics class, this drawstring is called from within either update() or paint() as shown in the above program example .The general form of is

  drawString(String msg,int x, int y)

- ✓ The **msg** indicates that string to be output beginning **at x,y**. in java window the upper-left corner location is 0,0.the drawstring() method will not recognize newline character.

- ✓ To set the background color of an applet window use **setBackground**() and to set the foreground color for example the color in which text is shown use **setForeground**().these methods are defined by Component and they have the following general forms

void setBackground(Color newColor) ,

void setForeground(Color newColor)

The **newColor** specifies that new color. The class Color defines the constant shown below that can be used to specify colors.

```
Color.black     Color.lightGray Color.yellow
Color.blue      Color.magenta   Color.red
Color.cyan      Color.orange    Color.white
Color.darkGray  Color.pink      Color.gray     Color.green
```

Example:

setBackround(Color.cyan);

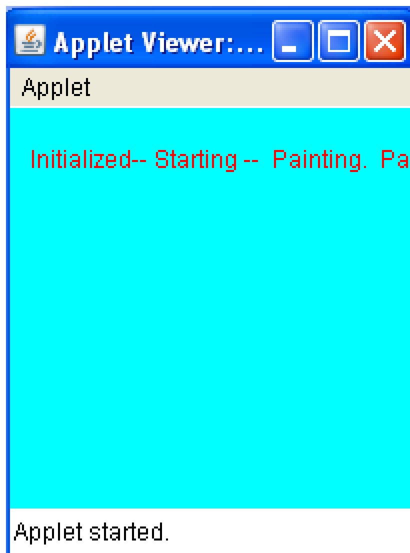
setForeground(Color.red)

Example Program:

```
/* This Applet sets the foreground and background colors and out puts a string. */
import java.applet.*;
import java.awt.*;

public class Simple extends Applet
{
        String msg;

  // set the foreground and background colors.
        public void init()
        {
           setBackground(Color.cyan);
           setForeground(Color.red);
           msg = "Initialized--";
        }
```

```
  // Add to the string to be displayed.
  public void start()
  {
        msg += " Starting --";
  }

  // Display the msg in the applet window.
  public void paint(Graphics g)
  {
        msg += " Painting.";
        g.drawString(msg, 10, 30);
   }
}
```

OUTPUT:



Requesting repainting;

- ✓ The **repaint()** method is defined by the AWT. It causes the AWT run time system to call to your applet's update() method, which in its default implementation, calls paint(). Again for example if a part of your applet

needs to output a string, it can store this string in a variable and then call repaint(). Inside paint(), you can output the string using drawstring().

The repaint method has four forms.

```
void repaint()

void repaint(int left, int top, int width, int height)

void repaint(long maxDelay)

void repaint(long maxDelay, int x, int y, int width, int height)
```

void repaint()

This causes the entire window to be repainted

void repaint(int left, int top, int width, int height)

This specifies a region that will be repainted. the integers left, top, width and height are in pixels. You save time by specifying a region to repaint instead of the whole window.

void repaint(long maxDelay)
void repaint(long maxDelay, int x, int y, int width, int height)

Calling repaint() is essentially a request that your applet be repainted sometime soon. However, if your system is slow or busy, update() might not be called immediately. This gives rise to a problem of update() being called sporadically. If your task requires consistent update time, like in animation, then use the above two forms of repaint(). Here, the maxDelay() is the maximum number of milliseconds that can elaspe before update() is called.

Using the Status Window

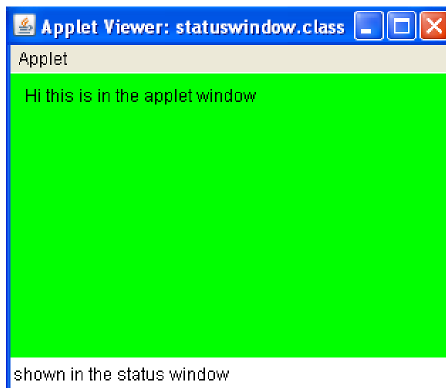
- ✓ If the user has chosen to show the Status Bar in their browser then messages can be put there from an applet.

  The **showStatus()** method would do it for this applet, if the applet was running in a browser.

Example

```
import java.awt.*;
import java.applet.*;
import java.awt.Graphics;

public class statuswindow extends Applet
{
        public void init()
        {
                setBackground(Color.green);

        }
        public void paint(Graphics g)
        {
        g.drawString("Hi this is in the applet window",10,20);
        showStatus("shown in the status window");
        }
}
```

OUTPUT



The HTML APPLET Tag

- ✓ The APPLET tag is used to start an applet from both an HTML document and from an applet viewer.
- ✓ An applet viewer will execute each APPLET tag that it finds in a separate window, while web browsers like Netscape Navigator, Internet Explorer, and HotJava will allow many applets on a single page.

The syntax for the standard APPLET tag is shown here. Bracketed items are optional.

```
< APPLET
      [CODEBASE = codebaseURL]
       CODE = appletFile
      [ALT = alternateText]
      [NAME = appletInstanceName]
      WIDTH = pixels HEIGHT = pixels
      [ALIGN = alignment]
      [VSPACE = pixels] [HSPACE = pixels]
>

  [< PARAM NAME = AttributeName VALUE = AttributeValue>]
  [< PARAM NAME = AttributeName2 VALUE = AttributeValue>]
  . . .
  [HTML Displayed in the absence of Java]
</APPLET>
```

CODEBASE: CODEBASE is an optional attribute that specifies the base URL of the applet code, which is the directory that will be searched for the applet's executable class file (specified by the CODE tag).

CODE: CODE is a required attribute that gives the name of the file containing your applet's compiled **.class** file. This file is relative to the code base URL of the applet, which is the directory that the HTML file was in or the directory indicated by CODEBASE if set.

ALT The ALT tag is an optional attribute used to specify a short text message that should be displayed if the browser understands the APPLET tag but can't currently run Java applets. This is distinct from the alternate HTML you provide for browsers that don't support applets.

WIDTH AND HEIGHT: WIDTH and HEIGHT are required attributes that give the size (in pixels) of the applet display area.

ALIGN: ALIGN is an optional attribute that specifies the alignment of the applet. This attribute is treated the same as the HTML IMG tag with these possible values: LEFT, RIGHT, TOP, BOTTOM, MIDDLE, BASELINE, TEXTTOP, ABSMIDDLE, and ABSBOTTOM.

VSPACE AND HSPACE: These attributes are optional. VSPACE specifies the space, in pixels, above and below the applet. HSPACE specifies the space, in pixels, on each side of the applet. They're treated the same as the IMG tag's VSPACE and HSPACE attributes.

PARAM NAME AND VALUE: The PARAM tag allows you to specify appletspecific arguments in an HTML page. Applets access their attributes with the **getParameter( )** method.

Passing parameters to Applets;

- ✓ Parameters are passed to applets in **NAME=VALUE pairs in <PARAM> tags** between the opening and closing APPLET tags. Inside the applet, you read the values passed through the PARAM tags with the **getParameter() method** of the java.applet.Applet class.

The program below demonstrates this with a generic string drawing applet. The applet parameter "Message" is the string to be drawn.

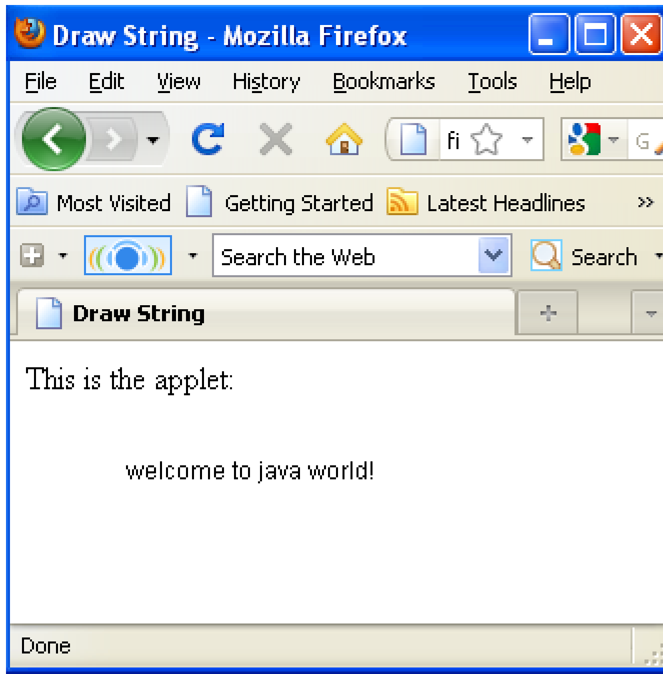
Example:

```
import java.applet.*;
import java.awt.*;
public class DrawStringApplet extends Applet
{
            private String defaultMessage = "Hello!";

            public void paint(Graphics g)
            {
                  String inputFromPage = this.getParameter("Message");
                   if (inputFromPage == null)

                  inputFromPage = defaultMessage;

                  g.drawString(inputFromPage, 50, 25);
             }
}
```

You also need an HTML file that references your applet. The following simple HTML file will do:

```
 <HTML>
<HEAD>
<TITLE> Draw String </TITLE>
</HEAD>

<BODY>
This is the applet:<P>
<APPLET code="DrawStringApplet" width="300" height="50">
<PARAM name="Message" value="welcome to java world!">
This page will be very boring if your
browser doesn't understand Java.
</APPLET>
</BODY>
</HTML>
```

OUTPUT



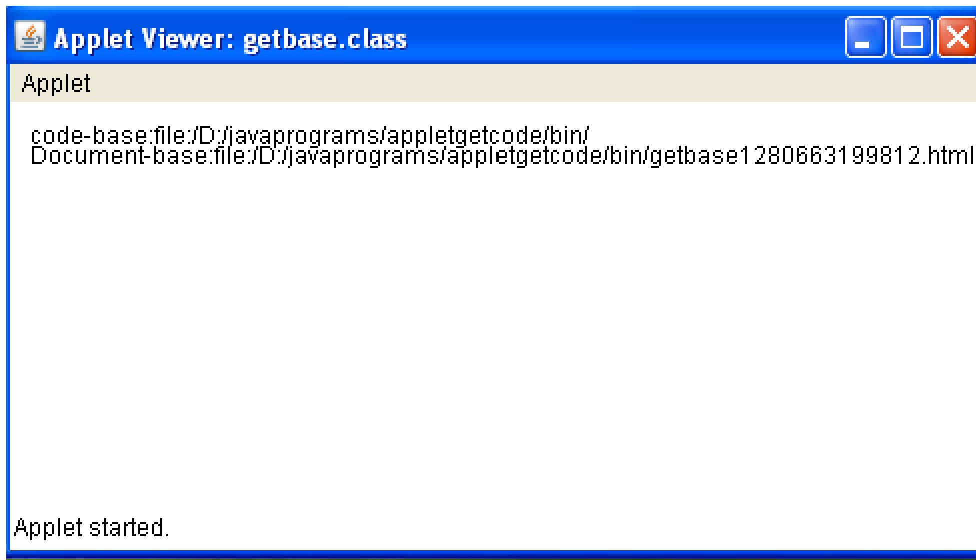
- ✓ You pass getParameter() a string that names the parameter you want. This string should match the name of a PARAM element in the HTML page. getParameter() returns the value of the parameter.
- ✓ All values are passed as strings. If you want to get another type like an integer, then you'll need to pass it as a string and convert it to the type you really want.
- ✓ The PARAM element is also straightforward. It occurs between <APPLET> and </APPLET>. It has two attributes of its own, NAME and VALUE. NAME identifies which PARAM this is. VALUE is the string value of the PARAM. Both should be enclosed in double quote marks if they contain white space.

getDocumentbase() and getCodebase()

- ✓ We sometimes need to load media and Text with the help of Applets. We have the facility to load the data from the directory which holds the HTML file which started the applet and the directory from which the applet's class loaded. These directories are returned in the form of URL by **getDocumnetBase( ) and getCodeBase( ) methods.**

```
import java.awt.*;
import java.applet.*;
import java.net.*;
public class getbase extends Applet
{
	public void paint(Graphics g)
	{
		String message;
	    URL url=getCodeBase();
		message="code-base:"+url.toString();
		g.drawString(message,10,20);
		url=getDocumentBase();
		message="Document-base:"+url.toString();
		g.drawString(message,10,30);

	}
}
```

OUTPUT



AppletContext and showDocument()

- ✓ AppletContext is an interface which helps us to get the required information from the environment in which the applet is running and getting executed.
- ✓ This information is derived by **getAppletContext( )** method which is defined by Applet. Once we get the information with the above mentioned method, we can easily bring another document into view by calling **showDocument( )** method. The basic functionality of this method is that it returns no value and never throw any exception even if it fails hence needed to be implemented with utmost care and caution.

There are two showDocument( ) methods.

1. The **method showDocument(URL)** displays the document at the specified URL.

2. The **method showDocument(URL, where)** displays the specified document at the specified location within the browser window.

```
import java.awt.*;
import java.applet.*;
import java.net.*;
public class contextdoc extends Applet
{
        public void start()
        {
                AppletContext ac=getAppletContext();
                URL url=getCodeBase();
                try
                {
                        ac.showDocument(new URL(url+"demo.html"));
                }
                catch(MalformedURLException e)
                {
                        showStatus("URL not found");

                }

        }
}
```

Syllabus:

SERVLETS: Java Servlet and common Gateway Interface Programming, A simple Java Servlet, Anatomy of a Java Servlet, Reading Data from a Client, Reading HTTP Requests Headers,

JAVA SERVER PAGES: JSP, JSP Tags. Tomcat, Request String, User Sessions, Cookies, Session Objects.

Servlet

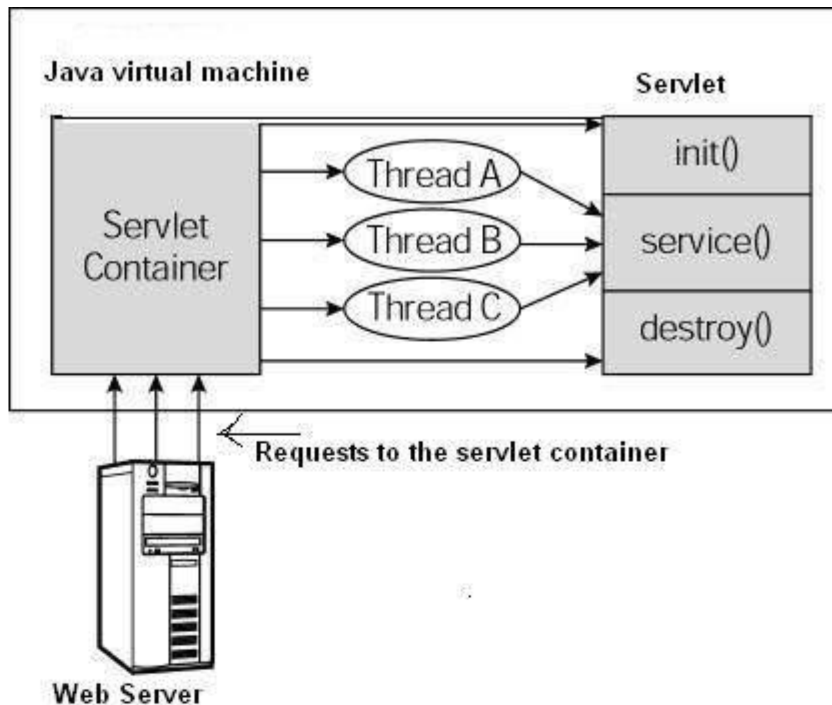
- A servlet is a small Java program that runs within a Web server.
- Servlets receive and respond to requests from Web clients, usually across HTTP, the Hyper Text Transfer Protocol.
- Servlet is an opposite of applet as a server-side applet.
- Applet is an application running on client while servlet is running on server.
- Servlets are server side components that provide a powerful mechanism for developing web applications.
- Using servlets we can create fast and efficient server side applications and can run it on any servlet enabled web server.
- Servlet runs entirely inside the JVM (Java Virtual Machine).
- Since the servlet runs on server side so it does not depend on browser compatibility.

Advantages of using Servlets

- Less response time because each request runs in a separate thread.
- Servlets are scalable.
- Servlets are robust and object oriented.
- Servlets are platform independent.

Servlet Architecture:

- The following figure depicts a typical servlet life-cycle scenario.
    - First the HTTP requests coming to the server are delegated to the servlet container.
    - The servlet container loads the servlet before invoking the service() method.
- Then the servlet container handles multiple requests by spawning multiple threads, each thread executing the service() method of a single instance of the



servlet.

- User sends request for a servlet by clicking a link that has URL to a servlet.
- The container finds the servlet using **deployment descriptor** and creates two objects

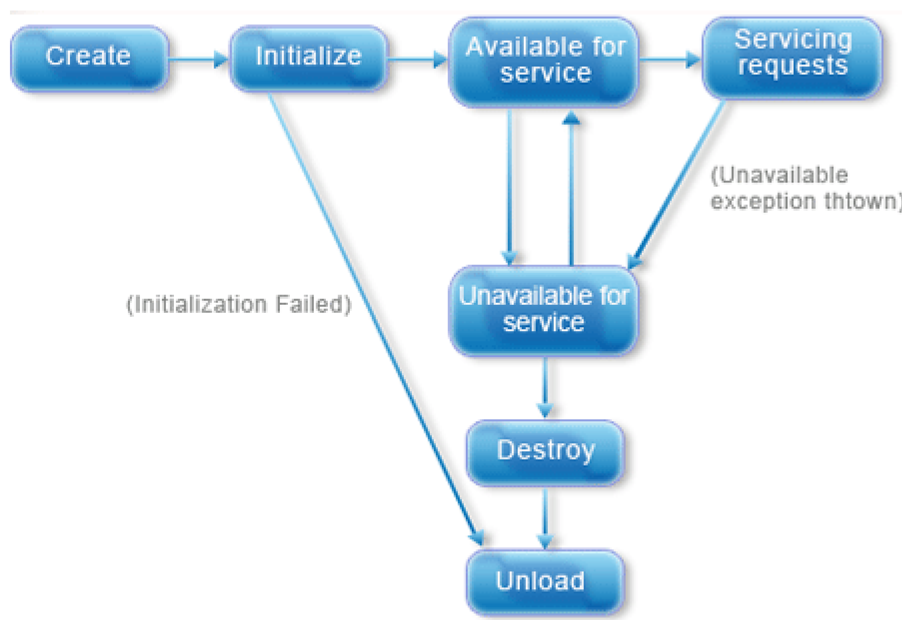
HttpServletRequest

HttpServletResponse

- Then the container creates or allocates a thread for that request and calls the Servlet's service() method and passes the **request, response** objects as arguments.
- The service() method, then decides which servlet method, doGet() or doPost() to call, based on **HTTP Request Method**(Get, Post etc) sent by the client. Suppose the client sent an HTTP GET request, so the service() will call Servlet's doGet() method.
- Then the Servlet uses response object to write the response back to the client.
- After the service() method is completed the **thread** dies. And the request and response objects are ready for **garbage collection**.

Life cycle of servlet:

- **The init() method**
- **The service() method**
- **The destroy() method**



The init() method :

- The init method is designed to be called only once. It is called when the servlet is first created, and not called again for each user request.
- So, it is used for one-time initializations, just as with the init method of applets.
- The servlet is normally created when a user first invokes a URL corresponding to the servlet, but you can also specify that the servlet be loaded when the server is first started.
- When a user invokes a servlet, a single instance of each servlet gets created, with each user request resulting in a new thread that is handed off to doGet or doPost as appropriate.
- The init() method simply creates or loads some data that will be used throughout the life of the servlet.
- The init method definition looks like this:

```
public void init() throws ServletException
{
        // Initialization code...
}
```

The service() method :

- The service() method is the main method to perform the actual task. The servlet container (i.e. web server) calls the service() method to handle requests coming from the client(browsers) and to write the formatted response back to the client.
- Each time the server receives a request for a servlet, the server spawns a new thread and calls service. The service() method checks the HTTP

request type (GET, POST, PUT, DELETE, etc.) and calls doGet, doPost, doPut, doDelete, etc. methods as appropriate.

- Here is the signature of this method:

```
public void service(ServletRequest request, ServletResponse response)
throws ServletException, IOException
{
}
```

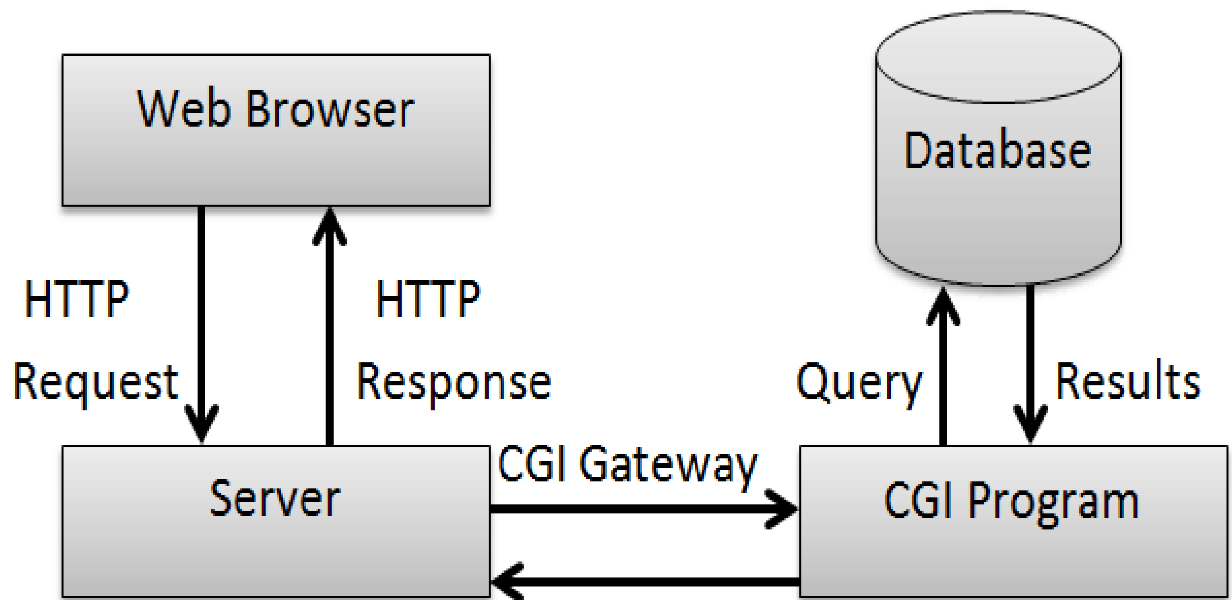
- The service () method is called by the container and service method invokes doGe, doPost, doPut, doDelete, etc. methods as appropriate. So you have nothing to do with service() method but you override either doGet() or doPost() depending on what type of request you receive from the client.
- The doGet() and doPost() are most frequently used methods with in each service request. Here is the signature of these two methods.

The destroy() method :

- The destroy() method is called only once at the end of the life cycle of a servlet. This method gives your servlet a chance to close database connections, halt background threads, write cookie lists or hit counts to disk, and perform other such cleanup activities.
- After the destroy() method is called, the servlet object is marked for garbage collection. The destroy method definition looks like this:

```
public void destroy()
{
// Finalization code...
}
```

Common Gateway Interface:



- It is a very well defined and supported standard.
- CGI scripts are generally written in either Perl, C, or maybe just a simple shell script.
- CGI is a technology that interfaces with HTML.
- CGI is the best method to create a counter because it is currently the quickest.
- CGI standard is generally the most compatible with today's browsers.
- The Common Gateway Interface (CGI) standard is a data-passing specification used when a Web server must send or receive data from an application such as a database.

- A CGI script passes the request from the Web server to a database, gets the output and returns it to the Web client.

General structure of servlet/SKELEton of servlet:

```
imort javax.servlet.*;
class className extends GenericServlet
{
      public void init() throws ServletException
       {
              // Initialization code...
       }
    public void service(ServletRequest request, ServletResponse
  response)throws  ServletException, IOException
    {
    }
    public void destroy()
    {
    // Finalization code...
    }
    }
```

Simple Servlet program:

```
import java.io.*;

import javax.servlet.*;

public class A extends GenericServlet

{

public void service(ServletRequest req ,ServletResponse res)throws

ServletException,IOException

{

res.setContentType("text/html") ;

PrintWriter out=res.getWriter();

out.println("<p> My First Servlet program </p> ");

}

}
```

Web deployment: (web.xml)

```
<servlet>

<servlet-name>CSA</servlet>

<servlet-class>A.class</sertlet-class>

</servlet>

<servlet-mapping>

<servlet-name>CSA</servlet>

<url-patter>*.dll</url-patter> </servlet-mapping>
```

Servlet API:

javax.servlet - The javax.servlet package contains a number of classes and

interfaces that describe and define the contracts between a servlet class and the runtime environment

provided for an instance of such a class by a conforming servlet container.

javax.servlet.http-The javax.servlet.http package contains a number of classes and interfaces that describe and define the contracts between a servlet class running under the HTTP protocol and the runtime environment provided for an instance of such a class by a conforming servlet container

The javax.servlet Package

The javax.servlet package contains a number of interfaces and classes that establish the framework in which servlets operate.

Interface Summary	
RequestDispatcher	Defines an object that receives requests from the client and sends them to any resource (such as a servlet, HTML file, or JSP file) on the server.
Servlet	Defines methods that all servlets must implement.
ServletConfig	A servlet configuration object used by a servlet container to pass information to a servlet during initialization.
ServletContext	Defines a set of methods that a servlet uses to communicate with its servlet container, for example, to get the MIME type of a file, dispatch requests, or write to a log file.
ServletRequest	Defines an object to provide client request information to a servlet.
ServletResponse	Defines an object to assist a servlet in sending a response to the client.

The following table summarizes the core classes that are provided in the javax.servlet package.

Class Summary	
GenericServlet	Defines a generic, protocol-independent servlet.
ServletInputStream	Provides an input stream for reading binary data from a client request, including an efficient `readLine` method for reading data one line at a time.
ServletOutputStream	Provides an output stream for sending binary data to the client.
ServletRequestAttributeEvent	This is the event class for notifications of changes to the attributes of the servlet request in an application.
ServletRequestEvent	Events of this kind indicate lifecycle events for a ServletRequest.

The javax.servlet.http Package

The javax.servlet.http package contains a number of interfaces classes that are commonly used by servlet developers.

Interface Summary	
HttpServlet	
HttpServletRequest	Extends the `ServletRequest` interface to provide request information for HTTP servlets.
HttpServletResponse	Extends the `ServletResponse` interface to provide HTTP-specific functionality in sending a response.
HttpSession	Provides a way to identify a user across more than one page request or visit to a Web site and to store information about that user.

Following are the class :

class	description
Cookie	Allow state information to be stored on client machine
HttpServlet	Provide Methods to handle Http Request and response
HttpSessionEvent	Encapsulate a session changed event
HttpSessionBindingEvent	Indicate when a listener is bounded to or unbounded from session value

Reading Servlet Parameters

- The ServletRequest class includes methods that allow you to read the names and values of parameters that are included in a client request.
- The example contains two files. A Web page is defined in PostParameters.htm and a servlet is defined in PostParametersServlet.java.

- Different mathods to read parameter are as follows:
    - getParameter(string)
    - getParamaterNames();
    - getParamaterValues();

getParameter()-returns a value of parameter in the string form

getParameterNames()-returns an enumeration of the parameter names.These are processed in loop

Program: To display greeting message on the browser Hello UserName How Are You accept username from the client.

```
import java.io.*;
import javax.servlet.ServletException;
import javax.servlet.http*;
public class A extends GenericServlet
{
public void service(ServletRequest req ,ServletResponse res)throws
ServletException,IOException
{
res.setContentType("text/html") ;
PrintWriter out=res.getWriter();
String msg=req.getParameter("t1");
out.println("hello"+msg+"how are you");
}
}
```

HTML code

```
<html>
<body>
<form action=http://localhost:8080/A >
<input type="text box" name="t1" value=" ">
<input type="submit" name="submit">
</form>
</body>
</html>
```

using getParameterName() method:

```
import java.io.*;
import javax.servlet.ServletException;
import javax.servlet.http*;
public class A extends GenericServlet
{
public void service(ServletRequest req ,ServletResponse res)throws
ServletException,IOException
{
res.setContentType("text/html") ;
PrintWriter out=res.getWriter();

Enumeration e=req.getParameterNames();
while(e.hasMoreElements())
```

```
{
Sting a=e.nextElement();
String msg=request.getParameter(a);
out.println(msg);
}
}
```

```
<html>
<body>
<form action=http://localhost:8080/A>
<input type="text box" name="t1" value=" ">
<input type="text box" name="t2" value=" ">
<input type="submit" name="submit">
</form>
</body>
</html>
```

Handling Http request and Http response:

- The HttpServlet class provide a specialized methods that handle the various types of HTTP request.
- The different methods are:

doGet(),doPost(),doOperation(),doPut(),doTrace(),doDelete()

HTTP doGet() method:

- The doGet() method is the method inside a servlet that gets called every time a request from a html or jsp page is submitted.
- The control first reaches the doGet() method of the servlet and then the servlet decides what functionality to invoke based on the submit

request. The get method called when the type of page submission is "GET".

- doGet is used when there is are requirement of sending data appended to a query string in the URL.
- The doGet models the GET method of Http and it is used to retrieve the info on the client from some server as a request to it.
- The doGet cannot be used to send too much info appended as a query stream. GET puts the form values into the URL string.
- GET is limited to about 256 characters (usually a browser limitation) and creates really ugly URLs.

Program:

```
import java.io.*;
import javax.servlet.ServletException;
import javax.servlet.http*;
public class A extends HttpServlet {
public void doGet(HttpServletRequest request, HttpServletResponse
response)throws
ServletException,IOException
{
res.setContentType("text/html") ;
PrintWriter out=res.getWriter();
String msg=req.getParameter("t1");
out.println("hello"+msg+"how are you");
}
}
```

```
<html>
<body>
<form action=http://localhost:8080/A method=GET >
<input type="text box" name="t1" value=" ">
<input type="submit" name="submit">
</form>
</body>
</html>
```

HTTP doPost() method:

- The doPost() method is the method inside a servlet that gets called every time a requests from a HTML or jsp page calls the servlet using "POST" method.
- doPost allows you to have extremely dense forms and pass that to the server without clutter or limitation in size. e.g. you obviously can't send a file from the client to the server via doGet.
- doPost has no limit on the amount of data you can send and because the data does not show up on the URL you can send passwords.
- But this does not mean that POST is truly secure. It is more secure in comparison to doGet method.

Program:

```
import java.io.*;
import javax.servlet.ServletException;
import javax.servlet.http*;
public class A extends HttpServlet {
```

```
public void doPost(HttpServletRequest request, HttpServletResponse
response)throws
ServletException,IOException
{
res.setContentType("text/html") ;
PrintWriter out=res.getWriter();
String msg=req.getParameter("t1");
out.println("hello"+msg+"how are you");
}
}
```

```
<html>
<body>
<form action=http://localhost:8080/A  method=POST>
<input type="text box" name="t1" value=" ">
<input type="submit" name="submit">
</form>
</body>
</html>
```

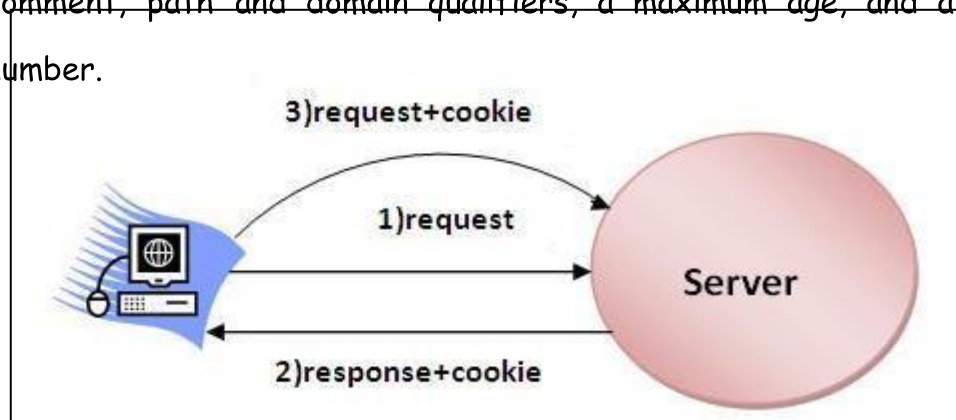
Difference between HTTP doGet and HTTP doPost methods of Servlet

Difference Type	**GET (doGet())**	**POST (doPost())**
HTTP Request	The request contains only the request line and	Along with request line and header it also contains HTTP
URL Pattern	Query string or form data is simply appended	Form name-value pairs are sent in the body of the

Parameter passing	The form elements are passed to the server by	The form elements are passed in the body of the
Size	The parameter data is limited (the limit	Can send huge amount of data to the server.
Idempotency	GET is Idempotent(can be applied multiple times without changing the values	POST is not idempotent(warns if applied multiple times without
Usage	Generally used to fetch some	Generally used to process the sent data.
Security	Not Safe - A person standing over your shoulder can view your userid/pwd if submitted via Get (Users can see data in address bar.)	Safe - No one will be able to view what data is getting submitted (Data hidden from
Data Format	Supports ASCII.	Supports ASCII + Binary.

Using Cookies

- A cookie is a small piece of information that is persisted between the multiple client requests.
- A cookie has a name, a single value, and optional attributes such as a comment, path and domain qualifiers, a maximum age, and a version number.



- Different methods in cookie class are:
  1. **String getName()**- Returns a name of cookie
  2. **String getValue()**-Returns a value of cookie
  3 .**int getMaxAge()**-Returns a maximum age of cookie in millisecond
  4. **String getDomain(**)-Returns a domain
  5. **boolean getSecure()-**Returns true if cookie is secure otherwise false
  6. **String getPath()-**Returns a path of cookie

  7.**void setPath(Sting)**- set the path of cookie

  8.**void setDomain(String)-**set the domain of cookie

  9.**void setMaxAge(int)-**set the maximum age of cookie

  10.**void setSecure(Boolean)-**set the secure of cookie.

Creating cookie:

- Cookie are created using cookie class constructor.
- Content of cookies are added the browser using addCookies() method.

Reading cookies:

- Reading the cookie information from the browser using getCookies() method.
- Find the length of cookie class.
- Retrive the information using different method belongs the cookie class.

Program: To create and read the cookie for the given cookie name as "EMPID" and its value as"AN2356".(vtu program)

```
public class A extends GenericServlet
{
public void service(ServletRequest req ,ServletResponse res)throws
```

```
ServletException,IOException
{
res.setContentType("text/html") ;
PrintWriter out=res.getWriter();
/* creating cookie object */
      Cookie c=new Cookie("EMPID","AN2356");
res.addCookie(c);//adding cookie in the response
/*reading cookies */
Cookie c[]=req.getCookies();
for(int i=0;i<c.length;i++)
{
String Name=c[i].getName();
String value= c[i].getValue();
out.println("name="+Name);
out.println("Value="+Value);
}
  } }
```

Session Tracking

- Session tracking is the capability of a server to maintain the current state of a single client's sequential requests.
- Session simply means a particular interval of time.
- Session Tracking is a way to maintain state of a user.
- The HTTP protocol used by Web servers is stateless.
- Each time user requests to the server, server treats the request as the new request.

- So we need to maintain the state of a user to recognize to particular user.
- This type of stateless transaction is not a problem unless you need to know the sequence of actions a client has performed while at your site.
- Different methods of HttpSession interface are as follows:

1.**object getAttribute(String)-**Returns the value associated with the name passed as argument.

2.**long getCreationTime()-**Returns the time when session created.

3.**String getID()-**Returns the session ID

4.**long getAccessedTIme()-**returns the time when client last made a request for this session.

5.**void setAttribute(String,object)-**Associates the values passed in the object name passed.

Program:

```
  import javax.servelt.*;
  import java.io.*;
public class A extends GenericServlet
{
public void service(ServletRequest req ,ServletResponse res)throws
ServletException,IOException
{
res.setContentType("text/html") ;
PrintWriter out=res.getWriter();

HttpSession h=req.getSesssion(true);
```

```
Date d=(Date) h.getAttribute("Date");
out.println("last date and time"+d);
Date d1=new Date();
d1=h.setAttribute("date",d1);
out.println("current date and time="+d1);
}
}
```

Servelt Interface:

methods	**description**
void destroy()	Called by the servlet container to indicate to a servlet that the servlet is being taken out of service.
void **init**(ServletConfig config)	Called by the servlet container to indicate to a servlet that the servlet is being placed into service.
void **service**(ServletRequest req, ServletResponse res)	Called by the servlet container to allow the servlet to respond to a request.
getServletInfo()	Returns information about the servlet, such as author, version, and copyright.
ServletConfig **getServletConfig**()	Returns a ServletConfig object, which contains initialization and startup parameters for this servlet

The GenericServlet Class

- The GenericServlet class provides implementations of the basic life cycle methods for a servlet.
- GenericServlet implements the Servlet and ServletConfig interfaces.
- In addition, a method to append a string to the server log file is available.The signatures of this method are shown here:

void log(String s)
void log(String s, Throwable e)

- Here, s is the string to be appended to the log, and e is an exception that occurred

13.using Tomcat for servlet Development:

certain steps taken to setup the tomcat

1.The examples here is Windows environment. The default location for Tomcat 5.5.17 is

C:\Program Files\Apache Software Foundation\Tomcat 5.5

2.to set the environmental variable JAVA_HOME to the top-level directory in which
the Java Software Development Kit is installed.

3.To start Tomcat, select Start Tomcat in the Start | Programs menu, , and the n press Start in the Tomcat Properties dialog. The directory

C:\Program Files\Apache Software Foundation\Tomcat 5.5\common\lib

Contain servlet.api.jar.

4.. To make this file accessible, update your CLASSPATH environment variable so that it includes

C:\Program Files\Apache Software Foundation\Tomcat5.5\common\lib\servlet.api.jar

5.First. Copy the servlet's class file into the following directory:

C:\Program Files\Apache Software

Foundation\Tomcat5.5\webapps\servlets.examples\WEB-INF\classes

6.Next, add the servlet's name and mapping to the web.xml file in the following directory

C:\Program Files\Apache Software Foundation\Tomcat 5.5\webapps\servlets.examples\WEB-INF

JSP

Java based technology that simplifies the developing of dynamic web sites.

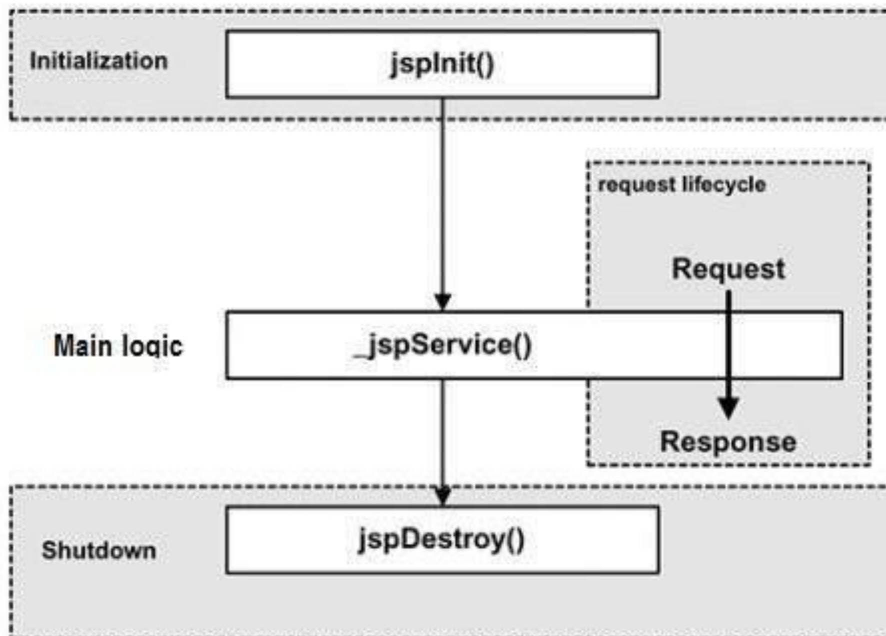
- JSP pages are HTML pages with embedded code that allows to access data from Java code running on the server.
- JSP provides separation of HTML presentation logic from the application logic.
- JSP technology provides a way to combine the worlds of HTML and Java servlet programming.
- JSP specs are built on the Java Servlet API.
- JSP supports two different styles for adding dynamic content to web pages:
- JSP pages can embed actual programming code (typically Java).
- JSP supports a set of HTML-like tags that interact with Java objects on the server (without the need for raw Java code to appear in the page).

Advantages of JSP

- JSP are translated and compiled into JAVA servlets but are easier to develop than JAVA servlets.
- JSP uses simplified scripting language based syntax for embedding HTML into JSP.
- JSP containers provide easy way for accessing standard objects and actions.
- JSP reaps all the benefits provided by JAVA servlets and web container environment, but they have an added advantage of being simpler and more natural program for web enabling enterprise developer.

- JSP use HTTP as default request / response communication paradigm and thus make JSP ideal as Web Enabling Technology.

JSP Life cycle:



Initialization:

- When a container loads a JSP it invokes the jspInit() method before servicing any requests. If you need to perform JSP-specific initialization, override the jspInit() method:

```
public void jspInit()
{
// Initialization code...
}
```

- Typically initialization is performed only once and as with the servlet init method, you generally initialize database connections, open files, and create lookup tables in the jspInit method.

JSP service:

- This phase of the JSP life cycle represents all interactions with requests until the JSP is destroyed.
- Whenever a browser requests a JSP and the page has been loaded and initialized, the JSP engine invokes the **_jspService()** method in the JSP.
- The _jspService() method takes an **HttpServletRequest** and an **HttpServletResponse** as its parameters as follows:

```
void _jspService(HttpServletRequest request, HttpServletResponse response)
{
// Service handling code...
}
```

- The _jspService() method of a JSP is invoked once per a request and is responsible for generating the response for that request and this method is also responsible for generating responses to all seven of the HTTP methods ie. GET, POST, DELETE etc.

JSP destroy:

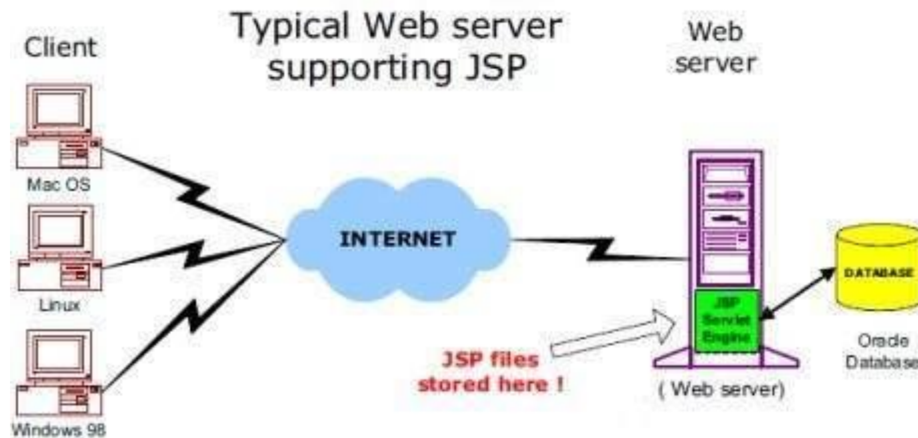
- The destruction phase of the JSP life cycle represents when a JSP is being removed from use by a container.
- The **jspDestroy()** method is the JSP equivalent of the destroy method for servlets. Override jspDestroy when you need to perform any cleanup, such as releasing database connections or closing open files.
- The jspDestroy() method has the following form:

```
public void jspDestroy()
{
// Your cleanup code goes here. }
```

JSP Architecture:

The following steps explain how the web server creates the web page using JSP:

- As with a normal page, your browser sends an HTTP request to the web server.
- The web server recognizes that the HTTP request is for a JSP page and forwards it to a JSP engine. This is done by using the URL or JSP page which ends with **.jsp** instead of .html.
- The JSP engine loads the JSP page from disk and converts it into a servlet content. This conversion is very simple in which all template text is converted to println( ) statements and all JSP elements are converted to Java code that implements the corresponding dynamic behavior of the page.
- The JSP engine compiles the servlet into an executable class and forwards the original request to a servlet engine.
- A part of the web server called the servlet engine loads the Servlet class and executes it. During execution, the servlet produces an output in HTML format, which the servlet engine passes to the web server inside an HTTP response.
- The web server forwards the HTTP response to your browser in terms of static HTML content.
- Finally web browser handles the dynamically generated HTML page inside the HTTP response exactly as if it were a static page



JSP Tags(VTU question VIMP):

JSP tags define java code that is to be executed before the output of a JSP program is sent to the browser. There are five types of JSP tags:

- Comment Tag
- Declaration statement Tag
- Directive Tag
- Expression Tag
- Scriptlet Tag

Directive Tag:

A Directive tag opens with <%@ and closes with %>. There are three commonly used directives.Used to import java packages into JSP program

Example:

```
<%@ page import = "java.sql.*" %>
```

Comment Tag:

A comment tag opens with <%-- and closes with --%>, and is followed by a

comment that usually describes the functionality of statements that follow the comment tag.

Example:

```
<%-- jsp comment tag --%>
```

Declaration Statement Tag:

A Declaration statement tag opens with <%! and is followed by a Java declaration statements that define variables, objects, and methods.

Example:

```
<%!
        int a=10;
        disp()  {   }
%>
```

Expression Tag:

- A JSP expression element contains a scripting language expression that is evaluated, converted to a String, and inserted where the expression appears in the JSP file.
- Because the value of an expression is converted to a String, you can use an expression within a line of text, whether or not it is tagged with HTML, in a JSP file.
- The expression element can contain any expression that is valid according to the Java anguage Specification but you cannot use a semicolon to end an expression.

Syntax two forms:

```
<%= expr %>
```

example:

```
<%!     int a = 5, b = 10;
            <%= a+b %>
%>
```

Scriptlet Tag:

A scriptlet tag opens with <% and contains commonly used java control statements and loops. It closes with %>

Syntax two forms:

```
<%          control statements %>
```

Example:

```
<% for (int i = 0; i < 2; i++) {    %>
            <p>Hello World!</p>
<%   }   %>
```

Program to display the grading system for the given java subject marks using control statements (VTU question VIMP):

```
<% !
int marks=65;
<% if(marks>=90)%>
<p>grade A</p>
<%else if(marks>=80 && marks<=89)%>
<p>Grade B</p>
<%else if(marks>=70 && marks<=79)%>
<p>Grade C</p>
<%else%>
<p>Fail</p>
%>
```

Request String:

- The browser generates a user **request string** whenever the submit button is selected.
- The HttpServletRequest parameter Request object has a request scope that is used to access the HTTP request data, and also provides a context to associate the request-specific data.
- Request object implements **javax.servlet.ServletRequest** interface.
- **Jsp provides the two ways of request string:**

  getParameter(String)

  getParameterNames()

Using request.getParameter()

getParameter() method requires an argument, which is the name of the field whose value you want to retrieve.

Program: Department has set the grade for java subject,accept the input from the user and display the grading on the browser. (VTU question VIMP)

above 90-grade A

80-89 grade B

70-79 grade C

below 70 Fail using jsp

A.html

```
<html>
<body>
<form action=A.jsp>
<input type="textbox" name="t1" value=" ">
```

```
<input type="submit" mane="submit">
</form>
</body>
</html>

        A.jsp
<%!
        String  t = request.getParameter("t1");
        int Marks=Integer.parseInt(t);
        <% if(marks>=90)%>
        <p>grade A</p>
        <%else if(marks>=80 && marks<=89)%>
        <p>Grade B</p>
        <%else if(marks>=70 && marks<=79)%>
        <p>Grade C</p>
        <%else%>
        <p>Fail</p>
              %>
```

using getParameterNames():

getParameterNames()-returns an enumeration of the parameter names.These are processed in loop

program:

```
<%@ import java.util.*; %>
<%!
```

```
Enumeration e=req.getParameterNames();
while(e.hasMoreElements())
{
Sting a=e.nextElement();
String msg=request.getParameter(a);
out.println(msg);
}
%>
```

A.html

```
<html>
<body>
<form action=A.jsp>
<input type=”textbox” name=”t1” value=” “>
<input type=”textbox” name=”t2” value=” “>
<input type=”submit” mane=”submit”>
</form>
</body>
</html>
```

Cookies:

- A **cookie** is a small piece of information created by a JSP program that is stored in the client’s hard disk by the browser. Cookies are used to store various kind of information such as username, password, and user

preferences, etc.

- Different methods in cookie class are:

    1.**String getName()**- Returns a name of cookie

    2.**String getValue()**-Returns a value of cookie

    3.**int getMaxAge()**-Returns a maximum age of cookie in millisecond

    4. **String getDomain(**)-Returns a domain

    5.**boolean getSecure()-**Returns true if cookie is secure otherwise false

    6.**String getPath()-**Returns a path of cookie

    7.**void setPath(Sting)**- set the path of cookie

    8.**void setDomain(String)-**set the domain of cookie

    9.**void setMaxAge(int)-**set the maximum age of cookie

    10.**void setSecure(Boolean)-**set the secure of cookie.

Creating cookie:

Cookie are created using cookie class constructor.

Content of cookies are added the browser using addCookies() method.

Reading cookies:

Reading the cookie information from the browser using getCookies() method.

Find the length of cookie class.

Retrive the information using different method belongs the cookie class

PROGRAM: To create and read the cookie for the given cookie name as "EMPID" and its value as"AN2356".(VTU question VIMP)

JSP program to create a cookie

```
<%!

        Cookie c=new Cookie("EMPID","AN2356");
        response.addCookie(c);

%>
```

JSP program to read a cookie

```
<%!
        Cookie c[]=request.getCookies();
        for(i=0;i<c.length;i++)
        {
        String name=c[i].getName();
        String value=c[i].getValue();
        out.println("name="+name);
        out.println("value="+value);
        }
%>
```

Session object(session tracking or session uses)

- The HttpSession object associated to the request
- Session object has a session scope that is an instance of javax.servlet.http.HttpSession class. Perhaps it is the most commonly used object to manage the state contexts.
- This object persist information across multiple user connection.
- Created automatically by
- Different methods of HttpSession interface are as follows:

1.**object getAttribute(String)-**Returns the value associated with the name passed as argument.

2.**long getCreationTime()-**Returns the time when session created.

3.**String getID()-**Returns the session ID

4.**long getAccessedTIme()-**returns the time when client last made a request for this session.

5.**void setAttribute(String,object)-**Associates the values passed in the object name passed.

Program:

```
<%!
        HttpSession h=req.getSesssion(true);
        Date d=(Date) h.getAttribute("Date");
        out.println("last date and time"+d);
        Date d1=new Date();
```

```
d1=h.setAttribute("date",d1);
out.println("current date and time="+d1);
%>
```