By Brijesh Singh Mob. 9891428723

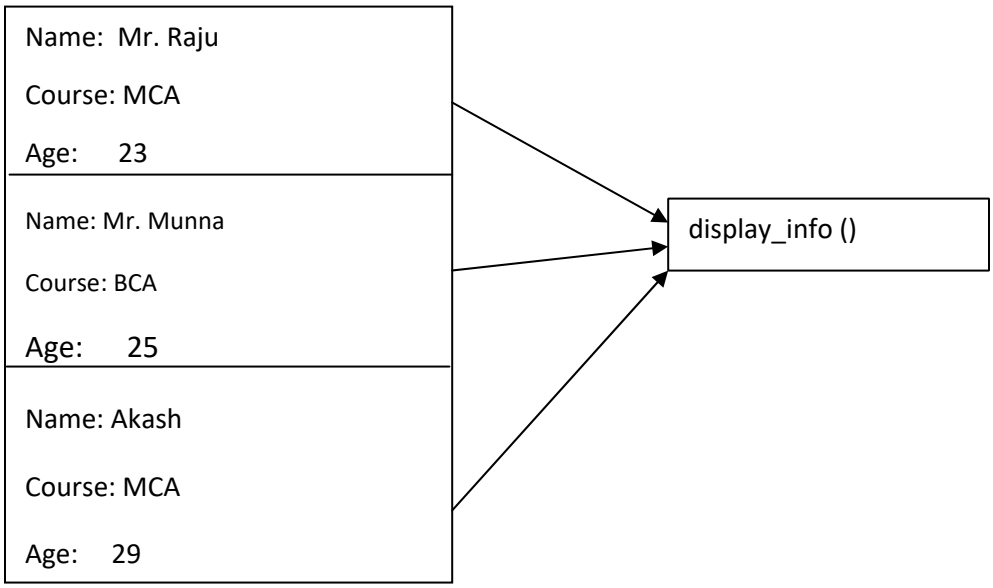
**Class and Object**

In object-oriented programming, a **class** is an extensible program-code-template for creating objects.

A **class** is nothing but a blueprint or a template for creating different objects which defines its properties and behaviors. **Java class** objects exhibit the properties and behaviors **defined** by its **class**. A **class** can contain variables and methods to describe the behavior of an object.

```
//class definition
class Student
 {
String name;
String course ;
int age;
      void display_info( )                          // function for displaying basic information
      {
            System.out.println(" Student Information");
            System.out.println("Name:"+name);
            System.out.println("Course:"+course);
            System.out.println("Age:"+age);
       }
}
// end of Student class
```

The above code will take memory and work like




Creating object

An object of a class can be created by performing these two steps.
1. Declare a object of class.
2. Acquire space for and bind it to object.

Why are two steps needed?

In the first step only a reference to an variable is created, no actual object's exists. For actual existence of an object memory is needed. That is acquired by using **new** operator.

The **new** operator in Java dynamically allocates memory for an object returns a reference to object and binds the reference with the object.

Declaring Student object
Step 1:
Student student1; // declaring reference to object
Step 2:
Student1 = new Student( ); // allocating memory

```
class Student
 {
String name;
String course ;
int age;
     void display_info( )                          // function for displaying basic information
     {
          System.out.println(" Student Information");
          System.out.println("Name:"+name);
          System.out.println("Course:"+course);
          System.out.println("Age:"+age);
      }
}

class  Display_Test

{

public static void main( String para[])

 {

          Student  student1;

          student1 = new Student();
          sudent1.name = "Mr.Raju";          //assigning value to name variable of  student1 object
          student1.course = "MCA";          //assigning value to course variable of  student1 object
          student1.age  =  23;                //assigning value to age variable of  student1 object
          student1.display_info();           // invoking display_info( ) method on student1 object
 }

}
```

## Assigning Object Reference Variables

One object can be assigned to another object, of same type but it works very different than a normal assignment. Let us see it with the help of a program.

```
 class Person
{
String name; int age ;
String address;
void Display( )
{
System.out.println("Person Information:"+name +" ("+age +")"+"\n"+address);
}
}

class Reference_Test
{
public static void main(String[] args)
{
        Person p = new Person();
        Person q = new Person();
        p.name= "Mr.Brijesh Kumar";
        p.age= 24;
        p.address = "B2B Block B Dass Garden, Baprola";
        p.Display();
        q = p;// q refer to p
        q.name = "Mr.Suresh";
        q.address = "22,Mahanadi,IGNOU,Maidan Garhi";
        p.Display();
        q.Display();
}
}
```

INTRODUCING METHODS

A Java class is a group of values with a set of operations. The user of a class manipulates object of that class only through the methods of that class.
General form of a method is:

```
type name_of_method (argument_list)
{
        // body of method
}
```

type specifies the type of data that will be returned by the method. This can be any data type including class types.

For Example

class Complex

```
{
    double real;
    double imag;
    void assignReal( double r)
    {
       real = r;
    }
    void assignImag( double i)
    {
       imag= i;
     }
void showComplex ( )
{
System.out.println("The Complex Number is :"+ real +"+i"+imag); }
}
class Complex_Test
{
public static void main(String[] args)
{
Complex R1 = new Complex();
R1.assignReal(5);
R1.assignImag(2);
R1.showComplex();
}
}
```

Output of this program is:
The Complex Number is :5.0+i2.0

Static Methods

Methods and variables that are not declared as *static* are known as instance methods and instance variables. Static variables and methods are also known as class variables or class methods since each class variable and each class method occurs once per class. A static method is a characteristic of a class, not of the objects it has created.

Let us see one example program:

```
import Java.util.Date;
 class DateApp
{
        public static void main(String args[])
        {
                Date today = new Date();
        System.out.println(today);
        }
}
```

Constructors

By Brijesh Singh Mob. 9891428723

A **constructor** is a special method of a class or structure in object-oriented programming that initializes an object of that type. A **constructor** is an instance method that usually has the same name as the class, and can be used to set the values of the members of an object, either to default or to user-defined values.

```
class Complex
{
double real;
double imag;
        Complex( double p, double q)
        {
        System.out.println("Constructor in process...");
        real = p; imag = q;
        }
        void showComplex ( )
        {
        System.out.println("The Complex Number is :"+ real +"+i"+imag); }
        }
class Complex_Test
{
        public static void main(String[] args)
        {
                Complex R1 = new Complex(5,2);
                R1.showComplex();
        }
}
```

The output of this program is:
Constructor in process…
The Complex Number is :5.0+i2.0

Constructors can be defined in two ways.
First , A constructor may not have parameter. This type of constructor is known as non-parameterized constructor.
Second, A constructor may take parameters. This type of constructor is known as parameterized constructor.
Examples
A constructor may not have parameter

```
class Point
{
 int x;
int y;
        Point()
        {
        x= 2;
        y= 4;
        }
        void Display_Point()
        {
```

```
                System.out.println("The Point is: ("+x+","+y+")");
        }
}
class Point_Test
{
        public static void main( String args[])
        {
        Point p1 = new Point();
        Point p2 = new Point();
        p1.Display_Point();
        p2.Display_Point();
        }
}
```

A constructor may take parameters

```
class Point
{
 int x;
int y;
        Point(int a, int b)
        {
        x= a;
        y= b;
        }
        void Display_Point()
        {
                System.out.println("The Point is: ("+x+","+y+")");
        }
}
class Point_Test
{
        public static void main( String args[])
        {
        Point p1 = new Point(2,4);
        Point p2 = new Point(3,5);
        p1.Display_Point();
        p2.Display_Point();
        }
}
```

Overloading Constructors

**Constructors Overloading** are used to increase the flexibility of a class by having more number of constructor for a single class.

Just like method **overloading**, **constructors** also can be **overloaded**. Same **constructor** declared with different parameters in the same class is known as **constructor overloading**. Compiler differentiates

which **constructor** is to be called depending upon the number of parameters and their sequence of data types.

```
class Student
{
String name;
int roll_no;
String course;
Student( String n, int r, String c)
{ name = n; roll_no  = r; course =
c;
}
Student(String n, int r )
{ name  = n; roll_no = r;
course = "MCA";
}
void Student_Info( )
{
System.out.println("********** Student Information ***********");
System.out.println("Name:"+name);
System.out.println("Course:"+course);
System.out.println("Roll Number:"+roll_no);
} }
class Student_Test
{
public static void main(String[] args)
{
Student s1 = new Student("Ravi Prakash", 987770012,"BCA");
Student s2 = new Student("Rajeev ", 980070042); s1.Student_Info();
s2.Student_Info();
}
}
```

Output of this program is:

```
********** Student Information ***********
Name:Ravi Prakash
Course:BCA
Roll Number:987770012
********** Student Information ***********
Name:Rajeev
Course:MCA
Roll Number:980070042
```

**this** keyword

If a method wants to refer the object through which it is invoked, it can refer to by using **this** keyword.

There are 6 usage of java this keyword.

1. this can be used to refer current class instance variable.
2. this can be used to invoke current class method (implicitly)
3. this() can be used to invoke current class constructor.
4. this can be passed as an argument in the method call.
5. this can be passed as argument in the constructor call.
6. this can be used to return the current class instance from the method.

Program without this keyword

```
class Student
{
int rollno;
String name;
float fee;
    Student(int rollno,String name,float fee)
    {
    rollno=rollno;
    name=name;
    fee=fee;
}
void display(){System.out.println(rollno+" "+name+" "+fee);
}
}
class TestThis1
{
public static void main(String args[])
{
Student s1=new Student(111,"ankit",5000f);
Student s2=new Student(112,"sumit",6000f);
s1.display();
s2.display();
}}
```

Program with this keyword

```
class Student
{
    int rollno;
```

```
        String name;
    float fee;
    Student(int rollno,String name,float fee)
    {
    this.rollno=rollno;
    this.name=name;
    this.fee=fee;
    }
    void display(){System.out.println(rollno+" "+name+" "+fee);}
    }

    class TestThis2
    {
        public static void main(String args[])
        {
        Student s1=new Student(111,"ankit",5000f);
        Student s2=new Student(112,"sumit",6000f);
        s1.display();
        s2.display();
        }
    }
```

OBJECTS AS PARAMETERS

Objects can be based as parameter and can also be returned from the methods.

```
class Marks
 {
String name;
int percentage;
String grade;
Marks(String n, int m)
{
name = n;
percentage = m;
}
void Display()
{
System.out.println("Student Name :"+name);
System.out.println("Percentage Marks:"+percentage);
System.out.println("Grade : "+grade);
System.out.println("*******************************");
}
```

```
}
class Object_Pass
 {
 public static void main(String[] args)
        {
         Marks ob1 = new Marks("Naveen",75);
        Marks ob2 = new Marks("Neeraj",45);
        Set_Grade(ob1);
        System.out.println("*******************************");
        ob1.Display();
        Set_Grade(ob2);
        ob2.Display();
        }
        static void Set_Grade(Marks m)
         {
            if (m.percentage >= 60)
               m.grade ="A";
             else if( m.percentage >=40)
                        m.grade = "B";
             else
                   m.grade="F";
         }
        }
```

Returning Objects

Like other basic data types, methods can return data of class type, i.e. object of class. An object returned from a method can be stored in any other object of that class like as values of basic type returned are stored in variables of that data type.

```
class Salary.
{
int basic ;
String E_id;
Salary( String a, int b)
{
E_id = a;
basic = b;
}
Salary  incr_Salary ( Salary s )
{
s.basic = basic*110/100;
return s;
}
 }
class Ob_return_Test
{
public static void main(String[] args)
```

```
        {
                Salary s1 = new Salary("I100",5000);
                Salary s2;// A new salary object s1 =
                s1.incr_Salary( s1);
                System.out.println("Current Basic  Salary
                is : "+ s2.basic);
          }
         }
```

METHOD OVERLOADING

**Method Overloading** is a feature that allows a class to have two or more **methods** having same name, if their argument lists are different. It means it should have different types of or different no of arguments.

```
class Test
 {
 int Area( int i)
   {
     return i*i;

   }

       int Area(int a,int b)
        {
             return a*b;
         }
}
class Area_Overload
{
public static void main(String args[])
{
Test t = new Test();
int area; area = t.Area(5);
System.out.println("Area of Squire is : "+area);
  area = t.Area(5,4);
System.out.println("Area of Rectangle is : "+area);
}
}
```

Output of this program is:
Area of Squire is: 25
Area of Rectangle is: 20.

GARBAGE COLLECTION

The name “garbage collection" implies that objects that are no longer needed by the program are “garbage” and can be thrown away and collects back into the heap.

The **garbage collector** is a program which runs on the **Java** Virtual Machine which gets rid of objects which are not being used by a **Java** application anymore. It is a form of automatic memory management. ... This means that in every iteration, a little bit of memory is being allocated to make a String object.

## Advantages and Disadvantages of garbage collection

Advantages

First, it can make programmers more productive. Programming in non-garbage-collected languages, the programmer has to spend a lot of time in keeping track of the memory de-allocation problem.

A second advantage of garbage collection is that it ensure program integrity. Garbage collection is an important part of Java's security strategy. Java programmers feel free from the fear of accidental crash of the system because JVM is there to take care of memory allocation and de–allocation.

Disadvantages

The major disadvantage of a garbage-collected heap is that it adds an overhead that can adversely affect program performance. The JVM has to keep track of objects that are being referenced by the executing program, and free unreferenced objects on the fly. This activity will likely take more CPU time than would have been required if the program explicitly freed unrefined memory. The second disadvantage is the programmers in a garbage-collected environment have less control over the scheduling of CPU time devoted to freeing objects that are no longer needed.

THE FINALIZE () METHOD

The **java**.lang.Object.**finalize**() is called by the garbage collector on an object when garbage collection determines that there are no more references to the object. A subclass overrides the **finalize method** to dispose of system resources or to perform other cleanup.

Sometimes it is required to take independent resources from some object before the garbage collector takes the object. To perform this operation, a method named finalized () is used. Finalized () is called by the garbage collector when it determines no more references to the object exist.

Finalized()method has the following properties:

1. Every class inherits the **finalize()** method from **Java.lang.Object**.
2. The garbage collector calls this method when it determines no more references to the object exist.
3. The **Object class** finalize method performs no actions but it may be overridden by any derived class.
4. Normally it should be overridden to clean-up **non-Java** resources, i.e. closing a file, taking file handle, etc.

```
public class TestGarbage1
{
 public void finalize()
  {
        System.out.println("object is garbage collected");
  }
```

```
    public static void main(String args[])
    {
     TestGarbage1 s1=new TestGarbage1();
     TestGarbage1 s2=new TestGarbage1();
     s1=null;
     s2=null;
     System.gc();
    }
    }
```

**INHERITANCE**

Inheritance id the methodology to create a new class with the help of an existing class. In **object-oriented programming**, **inheritance** enables new objects to take on the properties of existing objects. A class that is used as the basis for **inheritance** is called a superclass, parent class or base class. A class that inherits from a superclass is called a subclass, child class or derived class. The extends keyword is used to inherit the properties of a class.

The extends keyword indicates that you are making a new class that derives from an existing class. The meaning of "extends" is to increase the functionality.

Example

```
class  Student
{
String name;
String address;
int age;
Student( String a, String b, int c)
{
 name = a;
address = b;
age = c;
}
void display( )
{
System.out.println("*** Student Information***");
Sstem.out.println("Name : "+name+"\n"+"Address:"+address+"\n"+"Age:"+age);
 }
}
class PG_Student extends Student
{
int age;
int  percentage;
String course;
PG_Student(String a, String b, String c, int d , int e)
```

```
{
super(a,b,d);
course = c;
percentage = e;
age = super.age;
}
void display()
{
 super.display();
System.out.println("Course:"+course);
}
}
class Test_Student
{
public static void main(String[] args)
{
Student std1 = new Student("Mr. Amit Kumar" , "B-34/2 Saket J Block",23);
PG_Student pgstd1=new  PG_Student("Mr.Ramjeet ", "241- Near Fast Lane Raipur" ,"MCA", 23, 80);
std1.display(); pgstd1.display();
}
}
```

# Types of inheritance

## 1) Single Inheritance

**Single inheritance** is initial inheritance where we can extends only one class to derive a new class. The below flow diagram shows that class B extends only one class which is A. Here A is a **parent class** of B and B would be a **child class** of A.


(a) Single Inheritance

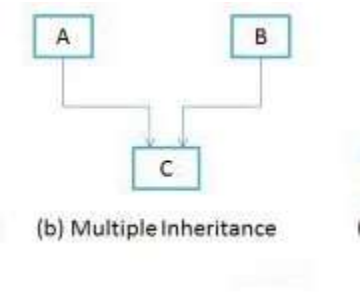
```
Class A
{
  public void methodA()
  {
    System.out.println("Base class method");
  }
}

Class B extends A
```

```
{
  public void methodB()
  {
    System.out.println("Child class method");
  }
  public static void main(String args[])
  {
    B obj = new B();
    obj.methodA();                    //calling super class method
    obj.methodB();                   //calling local method
  }
}
```

# 2) Multiple Inheritance

Multiple inheritance is the methodology to create a new class with the help of more than one classes. It means when we need to use the properties of more than one class to derive a class multiple inheritance exists there.

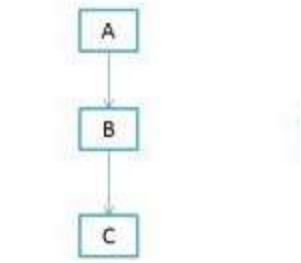
Here java do not supports multiple inheritance to use such features we need to interface.



(b) Multiple Inheritance

# 3) Multilevel Inheritance

**Multilevel inheritance** refers to a mechanism in OO technology where classes can be derives in level wise format. It means inheritance can be done using level by level.



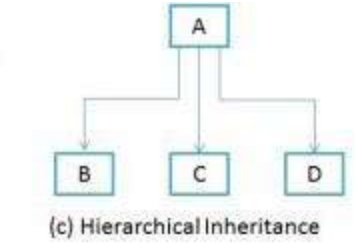
(d) Multilevel Inheritance

In the above example we can see that class A in inherited to create class B while class b is inherited to create class c.

```
Class X
{
  public void methodX()
  {
    System.out.println("Class X method");
  }
}
Class Y extends X
{
public void methodY()
{
System.out.println("class Y method");
}
}
Class Z extends Y
{
  public void methodZ()
  {
    System.out.println("class Z method");
  }
  public static void main(String args[])
  {
    Z obj = new Z();
    obj.methodX();          //calling grand parent class method
    obj.methodY();          //calling parent class method
    obj.methodZ();          //calling local method
  }
}
```

# 4) Hierarchical Inheritance

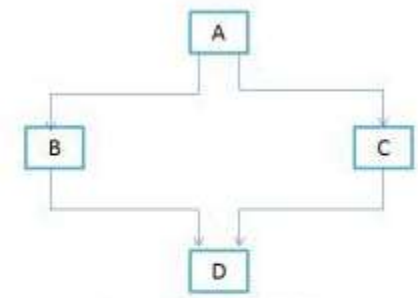
In such kind of inheritance one class is inherited by many **sub classes**. In below example class B,C and D **inherits** the same class A. A is **parent class (or base class)** of B,C & D.



(c) Hierarchical Inheritance

# 5) Hybrid Inheritance

In simple terms you can say that Hybrid inheritance is a combination of **Single** and **Multiple inheritance.** A typical flow diagram would look like below. A hybrid inheritance can be achieved in the java in a same way as multiple inheritance can be!! Using interfaces.



(e) Hybrid Inheritance

METHOD OVERRIDING

**Method overriding**, in object-oriented programming, is a language feature that allows a subclass or child class to provide a specific implementation of a **method** that is already provided by one of its super classes or parent classes.

Some important points that must be taken care while overriding a method:

i. An overriding method (largely) replaces the method it overrides.
ii. Each method in a parent class can be overridden at most once in any one of the subclass.
iii. Overriding methods must have exactly the same argument lists, both in type and in order.
iv. An overriding method must have exactly the same return type as the method it overrides.
v. Overriding is associated with inheritance.

```
class Figure
{
double sidea;
double sideb;
```

```
Figure(double a, double b)
{
sidea = a;
sideb = b;
}
Figure(double a)
{
sidea = a;
sideb = a;
}
 double area( )
{
System.out.println("Area inside figure is Undefined."); return 0;
}
}
class Rectangle extends Figure
{
Rectangle( double a , double b)
{
super ( a, b);
}
double area ( )
{
System.out.println("The Area of Rectangle:");
return sidea*sideb;
}
}
class Squre extends Figure
{
Squre( double a )
{
super (a);
}
double area( )
{
System.out.println("Area of Squre: ");
return sidea*sidea;
}
}
class Area_Overrid
{
public static void main(String[] args)
{
Figure f = new  Figure(20.9, 67.9);
Rectangle r = new  Rectangle( 34.2, 56.3);
Squre s = new Squre( 23.1);
System.out.println("***** Welcome to Override Demo ******");
```

```
f.area();
System.out.println(" "+r.area());
System.out.println(" "+s.area());
}
 }
```

ABSTRACT CLASSES

A superclass has common features that are shared by subclasses. In some cases you will find that superclass cannot have any instance (object) and such of classes are called abstract classes. Abstract classes usually contain abstract methods. Abstract method is a method signature (declaration) without implementation. Basically these abstract methods provide a common interface to different derived classes. Abstract classes are generally used to provide common interface derived classes.
For example,

```
public abstract class Player  // class is abstract
 {
private String name;
public Player(String  vname)
{ name=vname;  }
public String getName()         // regular method
{  return (name);
}
public abstract void Play(); //  abstract method: no implementation
}
```

Example -1

```
abstract class Bike
{
  abstract void run();
}
class Honda4 extends Bike
{
void run(){System.out.println("running safely..");
}
public static void main(String args[])
{
 Bike obj = new Honda4();
 obj.run();
}
}
```

Example -2

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

Example -3

```
abstract class Bank
{
abstract int getRateOfInterest();
}
class SBI extends Bank
{
int getRateOfInterest()
{return 7;
}
}
class PNB extends Bank
{
int getRateOfInterest(){return 8;}
```

```
}

class TestBank
{
public static void main(String args[])
{
Bank b;
b=new SBI();
System.out.println("Rate of Interest is: "+b.getRateOfInterest()+" %");
b=new PNB();
System.out.println("Rate of Interest is: "+b.getRateOfInterest()+" %");
}
}
```

FINAL KEYWORD

In the **Java** programming language, the **final keyword** is used in several different contexts to define an entity that can only be assigned once. Once a **final** variable has been assigned, it always contains the same value. It is used to make a variable as a constant, Restrict method overriding, Restrict inheritance. It is used at variable level, method level and class level. In java language final keyword can be used in following way.

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

## Final at variable level

Final keyword is used to make a variable as a constant. This is similar to const in other language.

```
public class Circle
{
public  static final double PI=3.14159;

public static void main(String[] args)
{
System.out.println(PI);
}
}
```

## Final at method level

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

```
class Employee
{
final void disp()
{
System.out.println("Hello Good Morning");
}
}
class Developer extends Employee
{
void disp()
{
System.out.println("How are you ?");
}
}
class FinalDemo
{
public static void main(String args[])
{
Developer obj=new Developer();
obj.disp();
}
}
```

## Final at class level

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

```
final class Employee
{
int salary=10000;
}
class Developer extends Employee
{
void show()
{
System.out.println("Hello Good Morning");
}
}
class FinalDemo
{
```

```
public static void main(String args[])
{
Developer obj=new Developer();
Developer obj=new Developer();
obj.show();
}
}
```

**Package**

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.

## Purpose of package

The purpose of package concept is to provide common classes and interfaces for any program separately.

## Advantage of package

- Package is used to categorize the classes and interfaces so that they can be easily maintained
- Application development time is less, because reuse the code
- Application memory space is less (main memory)
- Application execution time is less
- Application performance is enhance (improve)
- Redundancy (repetition) of code is minimized
- Package provides access protection.
- Package removes naming collision.

### built-in **package**

These are the package which are already designed by the Sun Microsystem and supply as a part of java API, every predefined package is collection of predefined classes, interfaces and sub-package.

## User defined package

If any package is design by the user is known as user defined package. User defined package are those which are developed by java programmer and supply as a part of their project to deal with common requirement.

## Rules to create user defined package

- package statement should be the first statement of any package program.
- Choose an appropriate class name or interface name and whose modifier must be public.
- Any package program can contain only one public class or only one public interface but it can contain any number of normal classes.
- Package program should not contain any main class (that means it should not contain any main())
- modifier of constructor of the class which is present in the package must be public. (This is not applicable in case of interface because interface have no constructor.)
- The modifier of method of class or interface which is present in the package must be public (This rule is optional in case of interface because interface methods by default public)
- Every package program should be save either with public class name or public Interface name

**Example**

Example-1

```
package mypack;
public class A
{
public void show()
{
System.out.println("Sum method");
}
}
```

Example-2

```
import mypack.A;
public class Hello
{
```

```
public static void main(String arg[])
{
A a=new A();
a.show();
System.out.println("show() class A");
}
}
```

Exampe-3

```
package pack;

public class A

{

        public void msg()

        {

                System.out.println("Hello");

        }

}
import pack. *;

class B

{

  public static void main(String args[])

  {

   A obj = new A();

   obj.msg();

  }

}
```

# INTERFACE

Interfaces in Java look similar to classes but they don't have instance variables and provides methods that too without implementation. It means that every method in the interface is strictly a declaration.. All methods and fields of an interface must be public. Interfaces are used to provide methods to be implemented by class(es).A class implements an interface using implements clause.

an interface is used to define a protocol of behavior that can be implemented by any class anywhere in the class hierarchy. As Java does not support multiple inheritance, interfaces are seen as the alternative of multiple inheritance, because of the feature that multiple interfaces can be implemented by a class.

Interfaces are useful because they:

• Provide similarities among unrelated classes without artificially forcing a class relationship. • Declare methods that one or more classes are expected to implement. • Allow objects from many different classes which can have the same type. This allows us to write methods that can work on objects from many different classes, which can even be in different inheritance hierarchies.

## Defining an Interface

Though an interface is similar to a class, there are several restrictions to be followed:

• An interface does not have instance variable.

• Every method of an interface is abstract.

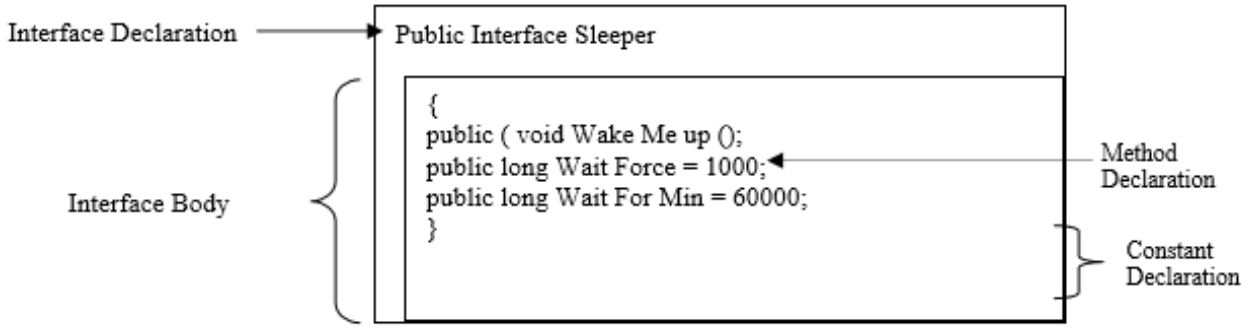
• All the methods of an interface are automatically public.

An interface definition has two components:

i. The interface declaration

ii. ii. The interface body.

The interface declaration declares various attributes of the interface such as its name and whether it extends another interface. The interface body contains the constant and method declarations within that interface.




# Example

public interface Calculate

```
{
 public double calculateInterest();
}
```

In the following program interface Calculate is implemented by SavingAccount class.

```
//program
interface Calculate
{
public double calculateInterest();
}

class SavingAccount implements Calculate
{
int Ac_No;
int Amount;
double Rate_of_Int;
int time;
SavingAccount( int a , double b, int c, int d)
{
Ac_No = a;
Rate_of_Int = b;
time = c;
Amount = d;
}
void DisplayInt()
{
System.out.println("Interest for Account No. "+Ac_No+" is Rs "+calculateInterest());
}
public double calculateInterest( )
{
return( Amount*Rate_of_Int*time/100);
}
}
public class TestIterface
{
public static void main( String args[])
```

```
{
SavingAccount S = new SavingAccount( 1010,4.5,5,5000); S.DisplayInt();
}
}
```

# DIFFERENCE BETWEEN INTERFACE AND ABSTRACT CLASSES

Abstract classes and interfaces carry similarity between them that methods of both should be implemented but there are many differences between them. These are:

- A class can implement more than one interface, but an abstract class can only subclass one class.
- An abstract class can have non-abstract methods. All methods of an interface are implicitly (or explicitly) abstract.
- An abstract class can declare instance variables; an interface cannot.
- An abstract class can have a user-defined constructor; an interface has no constructors.
- Every method of an interface is implicitly (or explicitly) public.
- An abstract class can have non-public methods.

**Exception**

In **java**, when any kind of abnormal conditions occurs with in a method then the **exceptions** are thrown in form of **Exception** Object i.e. the normal program control flow is stopped and an **exception** object is created to handle that exceptional condition. The method creates an object and hands it over to the runtime system.

Exception is some unusual condition that sometimes generates during the execution of our code. The reason may be an invalid input, no input and even violation of some security features. If we don't handle the exception ourselves then the Java Run Time automatically handles and reports the exception, finally terminating the program. This can be embarrassing if you are writing a professional program (Take your own project as an example). Thus handling some exceptional condition is really important. Using try and catch The two keywords "try and catch" are extremely useful for handling run time exception. If we suspect a part of code to be prone to exceptions, we can enclose it within the try block. If an exception occurs, it is handled by the catch block.

Now the same program is written using try and catch to handle exception.

```
class airtexc
{
        public static void main(String args[])
        {
        int no,no2,ans;
        no=10; no2=0;
        try
        {
        System.out.println("Try block begins"); ans=no/no2;
```

```
        }
        catch(ArithmeticException e)
        {
        System.out.println(" Sorry!!An Exception occured");
        System.out.println(" Exception Details"+e);
        }
        System.out.println("EXCEPTION DETAIL PRINTED");
        }
}
```

throw and throws keywords---

The “throw” keyword is used to manually throw an exception. As soon as the interpreter encounters the line with the throw keywords, it searches for the catch statement of the try block in which the “throw” keyword is enclosed. If no catch block is found then the default exception handler handles the exception and terminates the program.

```
class thrw
{
        public static void main(String args[])
        {
        try
                {
                System.out.println("Try block's execution begins");
                throw new ArithmeticException ();
                }
                catch(ArithmeticException obj)
                {
                System.out.println("Inside the catch block"); System.out.println("Exception detail---"+obj);
                }
        }
}
```

throws If we construct a method that throws an exception which it does not handle, the keyword “throws” warns the calling method of the exception.

```
return_type method( parameter list) throws exception
{
// Method’s body //
Other statements
}
```

A straightforward and clear example will be a simple program that takes some input from the user.

```
import java.io.*;
class trws
{
        public static void main(String args[]) throws IOException /. Use of “throws” keyword
        {
        int no;
        InputStreamReader rd=new InputStreamReader(System.in);
        BufferedReader inp=new BufferedReader(rd);
        System.out.println("Enter a number");
        String v1=inp.readLine();
        no=Integer.parseInt(v1);
        System.out.println("Multiplying a number");
        no=no*no; System.out.println("Square of the no. is "+no);
        }
}
```

## MULTITHREADING

Multithreading is the ability of a program or an operating system process to manage its use by more than one user at a time and to even manage multiple requests by the same user without having to have multiple copies of the programming running in the computer. Each user request for a program or system service (and here a user can also be another program) is kept track of as a thread with a separate identity. As programs work on behalf of the initial request for that thread and are interrupted by other requests, the status of work on behalf of that thread is kept track of until the work is completed.

The advantage of multithreaded programs is that they support logical concurrency. Many programming languages support multiprogramming, as it is the logically concurrent execution of multiple programs. For example, a program can request the operating system to execute programs A, B and C by having it spawn a separate process for each program. These programs can run in a concurrent manner, depending upon the multiprogramming features supported by the underlying operating system.

Difference between multithreading and multiprogramming

**Multiprogramming:**

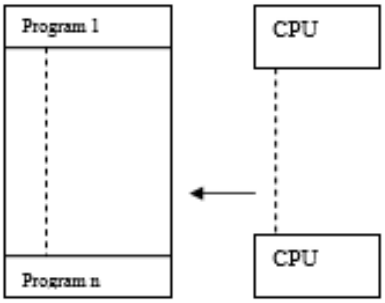
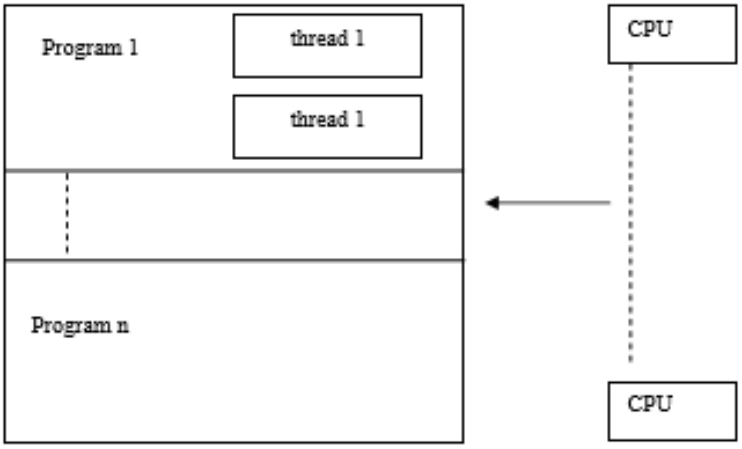
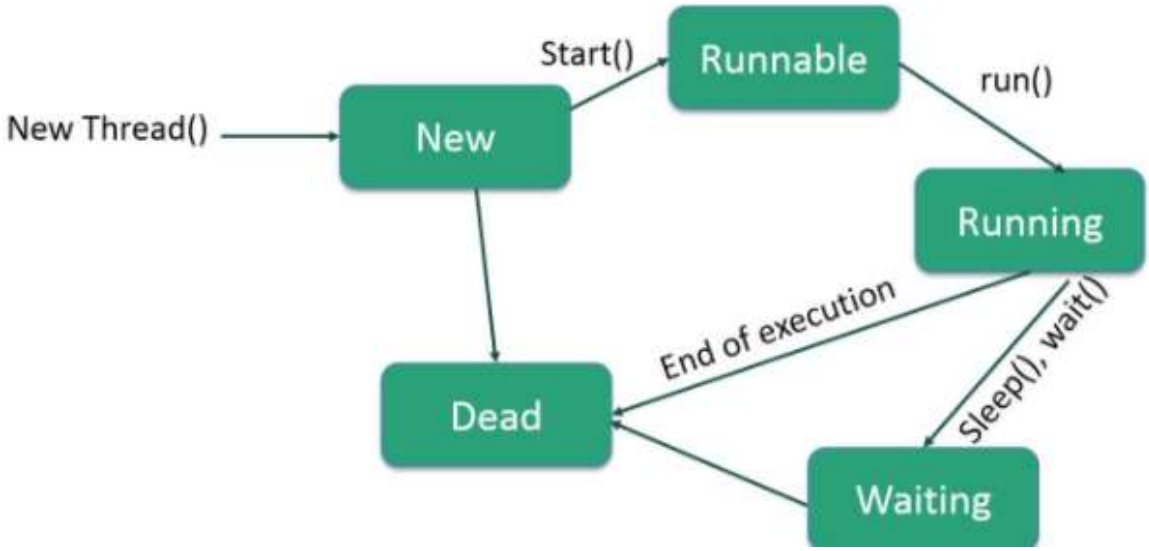



Figure 1a: Multiprogramming




Thread Life Cycle

A thread goes through various stages in its life cycle. For example, a thread is born, started, runs, and then dies. The following diagram shows the complete life cycle of a thread.



- **New** – A new thread begins its life cycle in the new state. It remains in this state until the program starts the thread. It is also referred to as a **born thread**.
- **Runnable** – After a newly born thread is started, the thread becomes runnable. A thread in this state is considered to be executing its task.
- **Waiting** – Sometimes, a thread transitions to the waiting state while the thread waits for another thread to perform a task. A thread transitions back to the runnable state only when another thread signals the waiting thread to continue executing.
- **Timed Waiting** – A runnable thread can enter the timed waiting state for a specified interval of time. A thread in this state transitions back to the runnable state when that time interval expires or when the event it is waiting for occurs.
- **Terminated (Dead)** – A runnable thread enters the terminated state when it completes its task or otherwise terminates.

### *Commonly used methods of Thread class*

1. *getName()* method returns the name of thread.
   *public String getName();*
2. *setName()* method is used to change the name of a thread.
   *public void setName (String name);*
3. *getPriority()* method is used to find out the priority of a thread.
   *public int getPriority();*
4. *setPriority()* method is used to change the priority of a thread.
   *public void setPriority(int priority);*
5. *sleep()* method is used to suspend a thread for the specified time.
   *public static void sleep (long milliseconds) throws InterruptedException;*
6. *start()* method is used to start a thread i.e. this method loads the run method as a thread.
   *public void start();*
7. *isAlive()* method is used to find out whether a thread is completed or not.
   *public boolean isAlive();*
8. *currentThread()* method returns the reference of the thread object for the current thread.

*public static Thread currentThread();*

*etc.*

**//Simple program**

```
class MyThreadDemo extends Thread
{
        public String MyMessage [ ]= {"Java","is","very","good","Programming","Language"};
        MyThreadDemo(String s)
        {
        super(s);
        }
        public void run( )
        {
        String name = getName( );
        for ( int i=0; i < MyMessage.length;i++)
        {
        Wait( );
        System.out.println (name +":"+ MyMessage [i]);
        }
        }
        void Wait( )
        {
        try
        {
        sleep(1000);
        }
        catch (InterruptedException e)
        {
        System.out.println (" Thread is Interrupted");
        }
        }
        }
        class ThreadDemo
        {
        public static void main ( String args [ ])
        {
        MyThreadDemo Td1= new MyThreadDemo("thread 1:");
        MyThreadDemo Td2= new MyThreadDemo("thread 2:");
```

```
Td1.start ( );
Td2.start ( );
boolean isAlive1 =  true;
 boolean  isAlive2 = true;
do
{
if (isAlive1 && ! Td1.isAlive( ))
{
isAlive1= false;
System.out.println ("Thread 1 is dead");
}
if (isAlive2 && !Td2.isAlive( ))
{
isAlive2= false;
System.out.println ("Thread 2 is dead");
}
}  while(isAlive1 || isAlive2);
}
}

//program
class Thread_Priority
{
public static void main (String args [ ])
{
try
{
Thread Td1  = new  Thread("Thread1");
Thread Td2  = new  Thread("Thread2");
System.out.println ("Before any change in default priority:");
System.out.println("The  Priority  of  "+Td1.getName() +"  is  "+  Td1.getPriority());
System.out.println("The Priority of "+Td1.getName() +" is "+ Td2.getPriority()); /
/change in priority
Td1.setPriority(7);
Td2.setPriority(8);
System.out.println ("After changing in Priority:");
```

```
System.out.println("The Priority of "+Td1.getName() +" is "+ Td1.getPriority());
System.out.println("The Priority of "+Td1.getName() +" is "+ Td2.getPriority());
}
catch ( Exception e)
{
System.out.println("Main thread interrupted");
}
}
}
```

# Synchronization in Java

Synchronization in java is the capability *to control the access of multiple threads to any shared resource*.

Java Synchronization is better option where we want to allow only one thread to access the shared resource.

## Why use Synchronization

The synchronization is mainly used to

1. To prevent thread interference.
2. To prevent consistency problem.

## Types of Synchronization

There are two types of synchronization

1. Process Synchronization
2. Thread Synchronization

## Thread Synchronization

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

1. Mutual Exclusive
    1. Synchronized method.
    2. Synchronized block.

3. static synchronization.

2. Cooperation (Inter-thread communication in java)

Write a Java program to show interprocess synchronization.

Cooperation (**Inter-thread communication**) is a mechanism in which a **thread** is paused running in its critical section and another **thread** is allowed to enter (or lock) in the same critical section to be executed.It is implemented by following methods of Object class: wait().

A **synchronized method** synchronizes on the object instance or the class. A thread only executes a **synchronized method** after it has acquired the lock for the**method's** object or class. static **synchronized methods synchronize** on the class object.

**You must prevent one thread from writing data while another thread is in the middle of reading it. For this purpose, Java implements model of interprocess synchronizations.**

**When a synchronized method is invoked for a given object, it tries to acquire the lock for that object. If it succeeds, no other synchronized method may be invoked for that object until the lock is released. A lock is automatically released when the method completes its execution and returns. A lock may also be released when a synchronized method executes certain methods, such as wait( ).**

```
class Table
{
 synchronized void printTable(int n)
{//synchronized method
  for(int i=1;i<=5;i++){
    System.out.println(n*i);
    try{
     Thread.sleep(400);
    }catch(Exception e){System.out.println(e);}
  }

 }
}

class MyThread1 extends Thread
{
```

```
Table t;
MyThread1(Table t)
{
this.t=t;
}
public void run()
{
t.printTable(5);
}

}
class MyThread2 extends Thread
{
Table t;
MyThread2(Table t)
{
this.t=t;
}
public void run()
{
t.printTable(100);
}
}

public class TestSynchronization2
{
    public static void main(String args[])
    {
    Table obj = new Table();//only one object
    MyThread1 t1=new MyThread1(obj);
    MyThread2 t2=new MyThread2(obj);
    t1.start();
    t2.start();
    }
}
```

INTERTHREAD COMMUNICATION

To avoid wastage of precious time of CPU, or to avoid polling, Java includes an interprocess communication mechanism via the wait( ), notify( ), and notifyAll( ) methods. These methods are implemented as final methods, so all classes have them. These three methods can be called only from within a synchronized method.

wait( ): tells the calling thread to give up the monitor and go to sleep until some other thread enters the same monitor and calls notify( ) or notifyAll().
notify( ): Wakes up a single thread that is waiting on this object's monitor.
notifyAll( ): Wakes up all threads that are waiting on this object's monitor, the highest priority Thread will be run first.

```
//program
class WaitNotifyTest implements Runnable
{
WaitNotifyTest ()
{
Thread th = new Thread (this);
th.start();
}
synchronized void notifyThat ()
{
System.out.println ("Notify the threads waiting");
this.notify();
 }
synchronized public void run()
{
    try
   {
    System.out.println("Thead is waiting....");
    this.wait ();
    }
    catch (InterruptedException e)
    {}
    System.out.println ("Waiting thread notified");
    }
    }
    class runWaightNotify
```

```
{
    public static void main (String args[])
    {
    WaitNotifyTest wait_not = new WaitNotifyTest();
    Thread.yield ();
    wait_not.notifyThat();
    }
    }
    Output: Thead is waiting....
    Notify the threads waiting
    Waiting thread notified
```

- **Input/Output** in java is stream based. A stream represents sequence of bytes or characters. Stream-based input/output has following advantages over conventional I/O.
    1. Abstraction
    2. Flexibility
    3. Performance
- *In Java, two types of input output streams are defined :-*
    - Byte oriented streams(*8-bit sequence*)
    - Character oriented streams(*16-bit sequence*)
- ***Commonly used byte oriented streams-***

## InputStream

is an abstract class that is extended by all *byte oriented input* streams.

- **ByteArrayInputStream** is an input stream that is used to read bytes from a byte array.
- **BufferedInputStream** is an input stream used to read bytes from a buffer.
- **FileInputStream** is an input stream used to read bytes from a file.

**InputStream** class contains an abstract **read()** method that is defined by all its subclasses.

- *public int read() throws IOException;*
- *public int read(byte[ ]) throws IOException;*

## OutputStream

is an abstract class that is extended by all *byte oriented output* streams.

- **ByteArrayOutputStream** is an output stream that is use to write bytes to a byte array.
- **BufferedOutputStream** is an output stream use to write bytes to a buffer.

- **FileOutputStream** is an output stream used to write bytes to a file.
- **PrintStream** is an output stream use to write primitive types, characters & strings to another stream. This stream provides **print()** and **println()** methods.

**OutputStream** class contains an abstract **write()** method that is defined by all its subclasses. It is used to write a byte or block of bytes to **OutputStream**.

- *public void write(int byte) throws IOException;*
- *public void write(byte[ ]) throws IOException;*

### General Signature of creating an Input/output stream –

*public TypeInputStream(Object source);*
*public TypeOutputStream(Object sink);*

*Example -*

- *Create an InputStream to read bytes from a byte array.*

```
byte a[ ] = {1, 2, 3, 4, 5};
ByteArrayInputStream b = new ByteArrayInputStream(a);
b.read();
```

- *Stream to read bytes from a file named "a.txt".*

```
FileInputStream f = new FileInputStream(a);
f.read();
```

- *Stream to read bytes from keyboard*

```
BufferedInputStream b = new BufferedInputStream(System.in);
b.read();
```

- *Commonly used character oriented streams-*

## Reader

is an abstract class that is extended by all *character oriented input* streams.

- **CharArrayReader** is an input stream use to read characters from a character array.
- **BufferedReader** is an input stream use to read characters and strings from a buffer.
- **FileReader** is an input stream use to read characters from a file.
- **InputStreamReader** is an input stream use to convert bytes to characters.

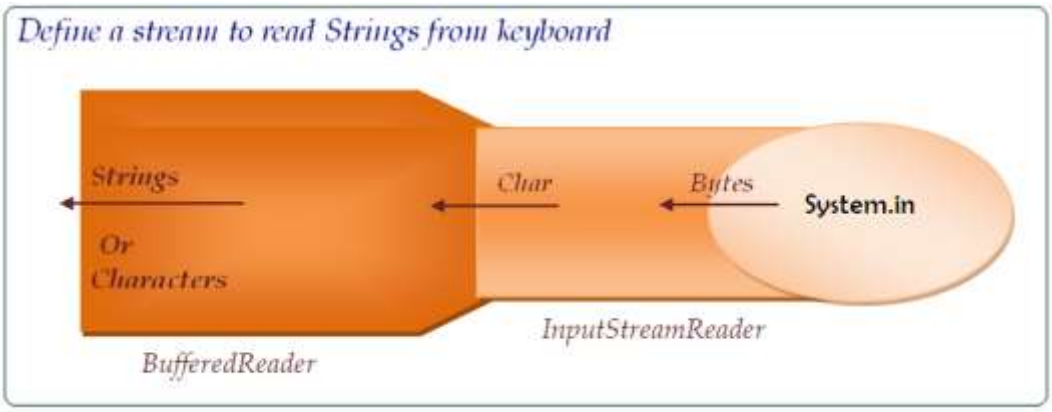
## Writer

is an abstract class that is extended by all *character oriented output* streams.

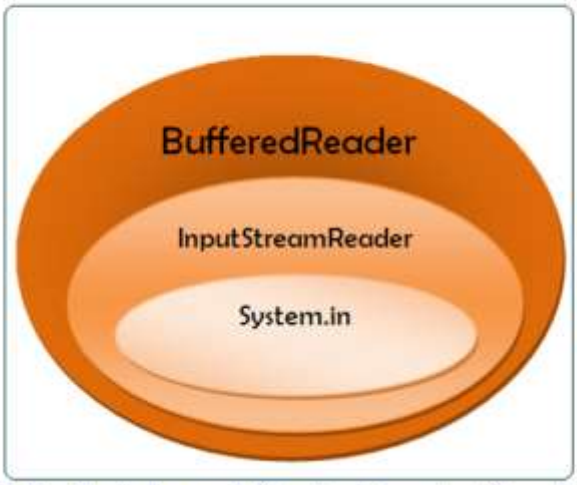
- **CharArrayWriter** is an output stream use to write characters to a character array.
- **BufferedWriter** is an output stream use to write characters to a buffer.
- **FileWriter** is an output stream use to write characters to a file.
- **OutputStreamWriter** is an output stream use to convert characters to bytes.
  - **PrintWriter** is the character oriented version of **PrintStream**.
  - etc.

**Reader** class provides an abstract **read()** method that is defined by all its subclasses. Apart from **read()** method **BufferedReader** class defines an additional method named **readLine()** method that reads a complete line of text from a stream & returns it as a string.

*public String readLine() throws IOException;*



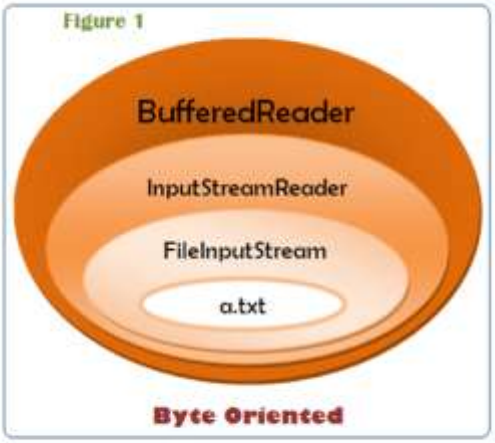
*The same diagram can also be represented as shown below. Diagram shown below will be used everywhere further in the notes ahead.*




```
BufferedReader b = new BufferedReader(new InputStreamReader(System.in));
```

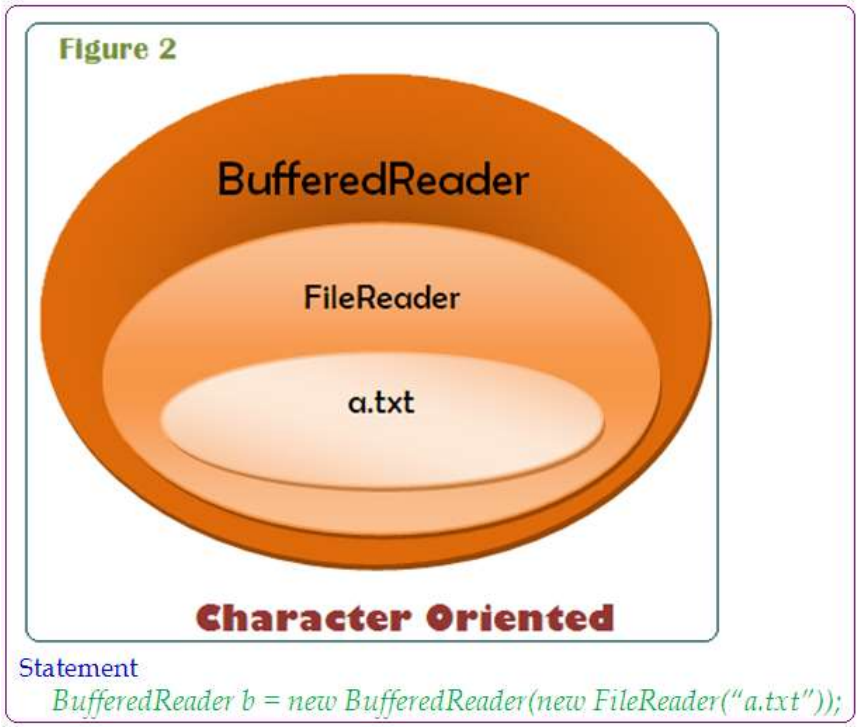
*Define two input streams to read data from a file named "a.txt" line by line.*

Figure 1




Statement

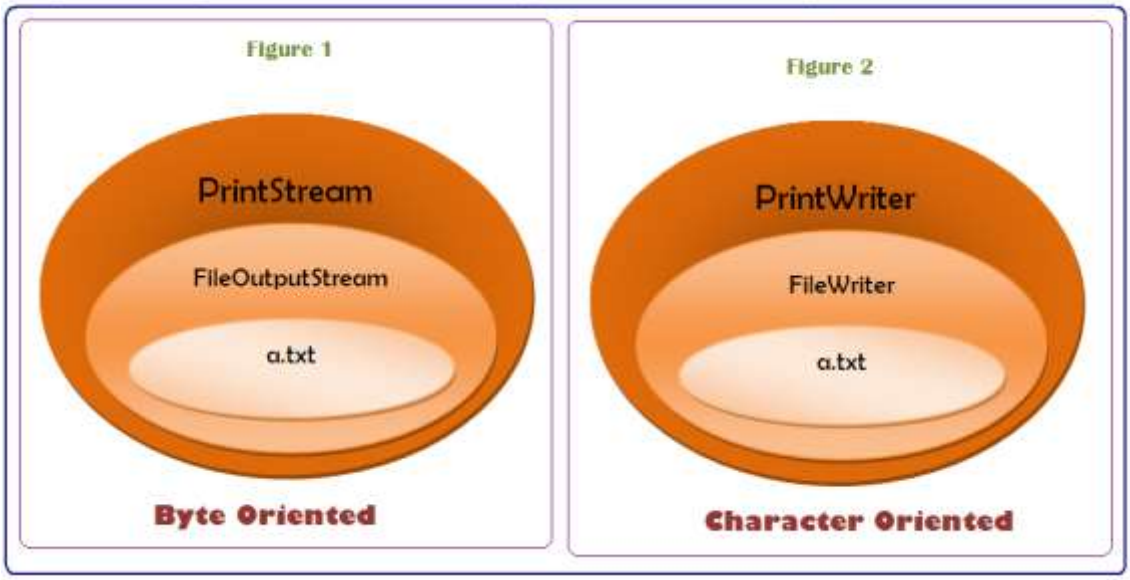
```
BufferedReader b = new BufferedReader(new InputStreamReader(new FileInputStream("a.txt")));
```




Statement

*BufferedReader b = new BufferedReader(new FileReader("a.txt"));*

- *Define two output streams to write data to a file named "a.txt", line by line.*




  - Statements

**NOTE** : All these I/O streams are defined in java.io package.

- *Example1*

```
//to display the contents of a text file on console.
// name of file is given as command line argument.
// data is read from the file byte by byte.

import java.io.*;

class Display
{
public static void main(String[] args)
{
        try
        {
                FileInputStream f = new FileInputStream(args[0]);
                int ch;
                while(true)
                {
                        ch=f.read();
                        if(ch==1)
                                break;
                        System.out.println((char) ch);
                }
        }
        catch(Exception e)
        {
                System.out.println(e);
        }
}
}
```

- *Example2*

```
//to display the contents of a text file on console.
// name of file is given as command line argument.
// data is read from the file line by line.

import java.io.*;

class Display1
{
public static void main(String[] args)
{
        try
        {
                BufferedReader f = new BufferedReader(new InputStreamReader(new
FileInputStream(args[0])));
                while(true)
                {
                        String line=f.readLine();
                        if(line==null)
                                break;
                        System.out.println(line);
```

```
                }
        }
        catch(Exception e)
        {
                System.out.println(e);
        }
}
}
```

- *Example3*

```
        //to display the contents of a text file on console.
        // name of file is given as command line argument.
        // data is read from the file in one go.

import java.io.*;

class Display3
{
public static void main(String[] args)
{
        try
        {
                FileInputStream f = new FileInputStream(args[0]);
                byte a[] = new byte[f.available()];
                f.read(a);
                String s = new String(a);
                System.out.println(s);
        }
        catch(Exception e)
        {
                System.out.println(e);
        }
}
}
```

```
//program
import Java.io.*;
public class FileOperation
{
        public static void main(String[] args) throws IOException
        {
        // the file must be called 'input.txt'
        String s = "input.txt" File f = new File(s);
        //check if file exists
        if (!f.exists())
        { System.out.println("\'" + s + "\' does not exit!"); return; }
        // open disk file for input
        BufferedReader inputFile = new BufferedReader(new FileReader(s));
        // read lines from the disk file, compute stats
        String line;
        int nLines = 0;
        int nCharacters = 0;
        while ((line = inputFile.readLine()) != null)
        {
        nLines++; nCharacters += line.length();
        } // output file statistics
        System.out.println("File statistics for \'" + s + "\'...");
        System.out.println("Number of lines = " + nLines);
        System.out.println("Number of characters = " + nCharacters); inputFile.close();
        }
        }
```

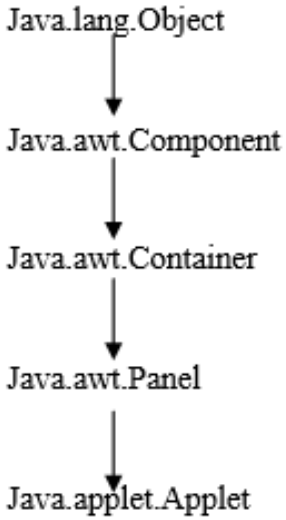
Output: File statistics for 'input.txt'…
Number of lines = 3
Number of characters = 7

## Applet

An **applet** is a small Internet-based program written in Java, a programming language for the Web, which can be downloaded by any computer. The **applet** is also able to run in HTML. The **applet** is usually embedded in an HTML page on a Web site and can be executed from within a browser.

Applet class is packed in the Java. Applet package which has several interfaces. These interfaces enable the creation of Applets, interaction of Applets with the browser, and playing audio clips in Applets. In Java 2, class Javax.swing. JApplet is used to define an Applet that uses the Swing GUI components.

in Java class hierarchy Object is the base class of Java.lang package. The Applet is placed into the hierarchy as follows:




Do's

• Draw pictures on a web page

• Create a new window and draw the picture in it.

• Play sounds.

• Receive input from the user through the keyboard or the mouse.

• Make a network connection to the server from where the Applet is downloaded, and send to and receive arbitrary data from that server.

Don'ts

• Write data on any of the host's disks.

• Read any data from the host's disks without the user's permission. In some environments, notably Netscape, an Applet cannot read data from the user's disks even with permission.

• Delete files

• Read from or write to arbitrary blocks of memory, even on a non-memoryprotected operating system like the MacOS

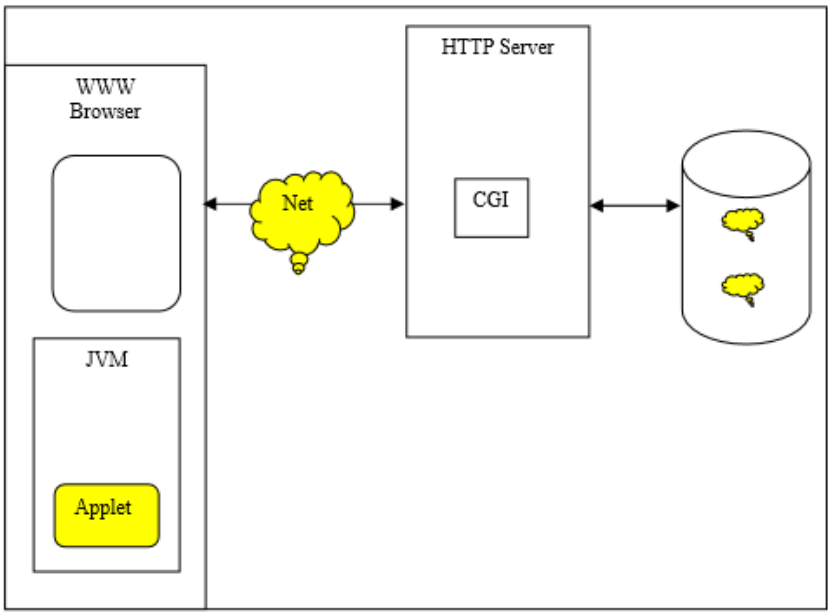
• Make a network connection to a host on the Internet other than the one from which it was downloaded.

• Call the native API directly (though Java API calls may eventually lead back to native API calls).

• Introduce a virus or Trojan horse into the host system

APPLET ARCHITECTURE

Java Applets are essentially Java programs that run within a web page. Applet programs are Java classes that extend the Java.Applet.Applet class and are embedded by reference within a HTML page. when Applets are combined with HTML, they can make an interface more dynamic and powerful than with HTML alone. While some Applets do nothing more than scroll text or play animations, but by incorporating these basic features in web pages you can make them dynamic. These dynamic web pages can be used in an enterprise application to view or manipulate data coming from some source on the server.




all Applets use their five following methods:

public void init ();
public void start();
public void paint();
public void stop();
public void destroy();

**init () method**: The init() method is called exactly once in an Applet's life, when the Applet is first loaded. It's normally used to read PARAM tags, start downloading any other images or media files you need, and to set up the user interface. Most Applets have init () methods.

**start () method:** The start() method is called at least once in an Applet's life, when the Applet is started or restarted. In some cases it may be called more than once. Many Applets you write will not have explicit start () method and will merely inherit one from their super class. A start() method is often used to start any threads the Applet will need while it runs.

**paint () method:** The task of paint () method is to draw graphics (such as lines, rectangles, string on characters on the screen).

**stop() method:** The stop () method is called at least once in an Applet's life, when the browser leaves the page in which the Applet is embedded. The Applet's start () method will be called if at some later point the browser returns to the page containing the Applet. In some cases the stop () method may be called multiple times in an Applet's life. Many Applets you write will not have explicit stop () methods and will merely inherit one from their super class. Your Applet should use the stop () method to pause any running threads. When your Applet is stopped, it should not use any CPU cycles.

**destroy () method:** The destroy() method is called exactly once in an Applet's life, just before the web browser unloads the Applet. This method is generally used to perform any final clean-up. For example, an Applet that stores state on the server might send some data back to the server before it is terminated. Many Applet programs generally don't have explicit destroy () methods and just inherit one from their super class.

**Simple example of applet programme**

**File Name: HelloWorldApplet.java**

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

**File name: HelloWorldApplet.html**

```
<HTML>
<HEAD>
<TITLE> Hello World </TITLE>
```

```
</HEAD>

<BODY>

<Applet code="HelloWorldApplet" width="150" height="50"> </Applet>

</BODY>

</HTML>
```

## What is the event handler in Java?

A function or method containing program statements that are executed in response to an event. An event handler typically is a software routine that processes actions such as keystrokes and **mouse** movements. With Web sites, event handlers make Web content dynamic.

## What is an event listener in Java?

**Event listeners** represent the interfaces responsible to handle **events**. **Java**provides various **Event listener** classes, however, only those which are more frequently used will be discussed. Every **method** of an **event listener method** has a single argument as an object which is the subclass of EventObject class.

## What is the use of ActionListener in Java?

Interface **ActionListener**. ... The class that is interested in processing an action event implements this interface, and the object created with that class is registered with a component, using the component's addActionListener **method**. When the action event occurs, that object's actionPerformed **method** is invoked.

Components of an Event

1. Event Object: When the user interacts with the application by clicking a mouse button or pressing a key an event is generated. The Operating System traps this event and the data associated with it. For example, info about time at which the event occurred, the event types (like keypress or mouse click etc.). This data is then passed on to the application to which the event belongs. You must note that, in Java, objects, which describe the events themselves, represent events. Java has a number of classes that describe and handle different categories of events.

3. Event Source: An event source is the object that generated the event, for example, if you click a button an ActionEvent Object is generated. The object of the ActionEvent class contains information about the event (button click).

3. Event-Handler: Is a method that understands the event and processes it. The event-handler method takes the Event object as a parameter. You can specify the objects that are to be notified when a specific event occurs. If the event is irrelevant, it is discarded.

The semantic listener interfaces defined by AWT semantic events are:

• ActionListener
• AjdustmentListener
• ItemListener
• TextListener

The low-level event listeners are as follows:

• ComponentListener
• ContainerListener
• FocusListener
• KeyListener
• MouseListener
• MouseMotionListener
• WindowsListener.

**Write a program in which whenever you click the mouse on the frame, the coordinates of the point on which the mouse is clicked are displayed on the screen**.

```
import Java.Applet.*;
import Java.awt.*;
import Java.awt.event.*;
public class TestMouse1
{
public static void main (String[] args)
{
        Frame f = new Frame("TestMouseListener");
        f.setSize(500,500);
        f.setVisible(true);
        f.addMouseListener(new MouseAdapter()
        {
                public void mouseClicked(MouseEvent e)
                {
```

```
                System.out.println("Mouse clicked: ("+e.getX()+","+e.getY()+")");
                }
        }
        );
}
}
```

**Write an Applet program in which you place a button and a textarea. When you click on button, in text area Your name and address is displayed. You have to take your name and address using < PARAM>.**

```
import Java.Applet.*;
import Java.awt.*;
public class DrawStringApplet1 extends Applet
{
        public void paint(Graphics g)
        {
        String str1 = this.getParameter("Message1");
        g.drawString(str1, 50, 25);
        String str2 = this.getParameter("Message2");
        g.drawString(str2, 50, 50);
        }
}
```

DrawStringApplet1.html file:

```
<HTML>
<HEAD>
<TITLE> Draw String </TITLE>
</HEAD>
<BODY>
<APPLET code="DrawStringApplet1" width="300" height="250">
<PARAM name="Message1" value="M. P. Mishra">
```

<PARAM name="Message2" value="SOCIS, IGNOU, New Delhi-68"> </APPLET>

</BODY>

</HTML>

JAVA GRAPHICS

A Java graphics context enables drawing on the screen. A Graphics object manages a graphics context by controlling how objects are drawn. Graphics objects contain methods for drawing, font manipulation, color manipulation and the other related operations. It has been developed using the Graphics object g (the argument to the applet's paint method) to manage the applet's graphics context. In other words you can say that Java's Graphics is capable of:

• Drawing 2D shapes
• Controlling colors
• Controlling fonts
• Providing Java 2D API
• Using More sophisticated graphics capabilities
• Drawing custom 2D shapes
• Filling shapes with colors and patterns.

Before we begin drawing with Graphics, you must understand Java's coordinate system, which is a scheme for identifying every possible point on the screen. Let us start with Graphics Context and Graphics Class.

Graphics Context and Graphics Class
• Enables drawing on screen
• Graphics object manages graphics context
• Controls how objects is drawn
• Class Graphics is abstract
• Cannot be instantiated
• Contributes to Java's portability
• Class Component method paint takes Graphics object.

Drawing Lines

Drawing straight lines with Java can be done as follows:

call g.drawLine(x1,y1,x2,y2) method, where (x1, y1) and (x2, y2) are the endpoints of your lines and g is the Graphics object you are drawing with. The following program will result in a line on the applet.

```
import java.applet.*;
import java.awt.*;
public class SimpleLine extends Applet
{
```

```
public void paint(Graphics g)
{
g.drawLine(10, 20, 30, 40);
}

}
```

Drawing Rectangle

Drawing rectangles is simple. Start with a Graphics object g and call its drawRect() method:
public void drawRect(int x, int y, int width, int height)

Drawing Ovals and Circles

Java has methods to draw outlined and filled ovals. These methods are called drawOval() and fillOval() respectively. These two methods are:

public void drawOval(int left, int top, int width, int height)

public void fillOval(int left, int top, int width, int height)

BUILDING USER INTERFACE WITH AWT

| CONTROLS | FUNCTIONS |
|---|---|
| Textbox | Accepts single line alphanumeric entry. |
| TextArea | Accepts multiple line alphanumeric entry. |
| Push button | Triggers a sequence of actions. |
| Label | Displays Text. |
| Check box | Accepts data that has a yes/no value. More than one checkbox can be selected. |
| Radio button | Similar to check box except that it allows the user to select a single option from a group. |
| Combo box | Displays a drop-down list for single item selection. It allows new value to be entered. |
| List box | Similar to combo box except that it allows a user to select single or multiple items. New values cannot be entered. |

| CONTROLS | CLASS |
|---|---|
| Textbox | TextField |
| TextArea | TextArea |
| Push button | Button |
| Check box | CheckBox |
| Radio button | CheckboxGroup with CheckBox |
| Combo box | Choice |
| List box | List |

Example Program:

```
/* <Applet code= "ButtonTest.class" Width = 500 Height = 100>  </applet> */
 import java.awt.*;
import java.applet.Applet; public class ButtonTest extends Applet
{
        Button b1 = new Button ("Play");
        Button b2 = new Button ("Stop");
        public void init()
        {
        add(b1);
        add(b2);
        }
}
```

```
import java.awt.*;
public class CheckboxTest extends java.applet.Applet
{
Checkbox c1 = new Checkbox ("Java");
Checkbox c2 = new Checkbox ("XML");
Checkbox c3 = new Checkbox ("VB");
public void init()
{
add(c1);
add(c2);
add(c3);
}
}
```

```
import java.awt.*;
Public class ChoiceTest extends java.applet.Applet
{
Choice shoplist = new Choice();
Public void init()
{
shoplist.addItem("Bed And Bath");
shoplist.addItem("Furniture");
shoplist.addItem("Clothing");
shoplist.addItem("Home Appliance");
shoplist.addItem("Toys and Accessories");
add(shoplist);
}
}
```

LAYOUTS AND LAYOUT MANAGER

When you add a component to an applet or a container, the container uses its layout manager to decide where to put the component. Different LayoutManager classes use different rules to place components.

java.awt.LayoutManager is an interface. Five classes in the java packages implement it:

- FlowLayout
- BorderLayout
- CardLayout
- GridLayout
- GridBagLayout
- plus javax.swing.BoxLayout

For example the following applet uses a FlowLayout to position a series of buttons that mimic the buttons on a tape deck.

```
import java.applet.*;
import java.awt.*;
public class FlowTest extends Applet
{
public void init()
{
```

```
this.setLayout(new FlowLayout());
this.add( new Button("Add"));
this.add( new Button("Modify"));
this.add( new Button("Delete"));
this.add( new Button("Ok"));
this.add( new Button("CANCEL"));
}
}
```

a BorderLayout include the name of the section you wish to add them to do like done in the program given below.

```
this.add("South", new Button("Start"));
```

```
import java.applet.*;
 import java.awt.*;
public class BorderLayouttest extends Applet
{
public void init()
{
this.setLayout(new BorderLayout(2, 3));
this.add("South", new Button("Start"));
this.add("North", new Button("Stop"));
this.add("East", new Button("Play"));
this.add("West", new Button("Pause"));
}
}
```

GridbagLayout.

```
import java.awt.*;
public class GridbagLayouttest extends Frame
{
Button b1,b2,b3,b4,b5;
GridBagLayout gbl=new GridBagLayout();
```

```
GridBagConstraints gbc=new GridBagConstraints();
public GridbagLayouttest()
{
setLayout(gbl);
gbc.gridx=0;
gbc.gridy=0;
gbl.setConstraints(b1=new Button("0,0"),gbc);
gbc.gridx=4;   //4th column
gbc.gridy=3;   //3rd row
gbl.setConstraints(b2=new Button("4,3"),gbc);
gbc.gridx=8;   //8th column
gbc.gridy=5;   //5rd row
gbl.setConstraints(b3=new Button("8,5"),gbc);
gbc.gridx=10;   //10th column
gbc.gridy=3;   //3rd row
gbl.setConstraints(b4=new Button("10,3"),gbc);
gbc.gridx=20;   //20th column gbc.gridy=3;   //3rd row
gbl.setConstraints(b5=new Button("20,3"),gbc);
add(b1);
add(b2);
add(b3);
add(b4);
add(b5);
setSize(200,200);
setVisible(true);
}
public static void main(String a[])
{
GridbagLayouttest gb=  new GridbagLayouttest();
}
}
```

CONTAINER

A **container** is a component which can contain other components inside itself. It is also an instance of a subclass of **java**.awt.**Container** . **java**.awt.**Container**extends **java**.awt.Component so **containers** are themselves components. In general components are contained in a **container**. An applet is a**container**.

What are component and container?

Swing **components and container** objects. In Java, a component is the basic user interface object and is found in all Java applications. **Components** include lists, buttons, panels, and windows. To use **components**, you need to place them in a**container**. A **container** is a component that holds and manages other**components**.

A Container in AWT is a component itself and it provides the capability to add a component to itself. Following are certain noticable points to be considered. Sub classes of Container are called as Container. For example, **JPanel**, **JFrame** and JWindow. Container can add only a Component to itself.

# Socket

Socket is a data structure that maintains necessary information used for communication between client & server. Therefore both end of communication has its own sockets.
Java introduces socket based communications, which enable applications to view networking as if it were file I/O– a program which can read from socket or write to a socket with the simplicity as reading from a file or writing to a file. Java provides stream sockets and datagram sockets

Stream Sockets

Stream Sockets are used to provide a connection-oriented service (i.e. TCP- Transmission Control Protocol). With stream sockets a process establishes a connection to another process. Once the connection is in place, data flows between processes in continuous streams.

Datagram Sockets

This socket are used to provide a connection-less service, which does not guarantee that packets reach the destination and they are in the order at the destination. In this, individual packets of information are transmitted. In fact it is observed that packets can be lost, can be duplicated, and can even be out of sequence.

Write a program to show that from the client side you send a string to a server that reverse the string which is displayed on the client side
Client side Programming

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

```
public class Client
{
public static final int DEFAULT_PORT = 8000;
public static void usage()
{
System.out.println("Usage: java Client <hostname> [<port>]");
System.exit(0);
}
public static void main(String[] args)
{
        int port = DEFAULT_PORT;
        Socket s = null;
// Parse the port specification
if ((args.length != 1) && (args.length != 2))
usage();
 if (args.length == 1) port = DEFAULT_PORT;
else
{
try
{
port = Integer.parseInt(args[1]);
}
catch (NumberFormatException e)
{
usage();
}
}
try
{          // Here is  a socket to communicate to the specified host and port
s = new Socket(args[0], port);
 BufferedReader sin = new BufferedReader(new InputStreamReader(s.getInputStream()));
//stream for reading
PrintStream sout = new PrintStream(s.getOutputStream()); // stream for writing lines of text
// Here ise a stream for reading lines of text from the console
BufferedReader in = new BufferedReader(new InputStreamReader(System.in));
System.out.println("Connected to " + s.getInetAddress() + ":"+ s.getPort());
String line;
```

```
while(true)
{ // print a prompt
System.out.print("> ");
System.out.flush(); // read a line from the console; check for EOF
line = in.readLine();
if (line == null)
break; // Send it to the server
sout.println(line); // Read a line from the server.
line = sin.readLine(); // Check if connection is closed (i.e. for EOF)
if (line == null)
{
System.out.println("Connection closed by server.");
break;
} // And write the line to the console.
System.out.println(line);
}
}
catch (IOException e)
{

System.err.println(e);

}
finally

{
try
{
if (s != null)
s.close();
}
catch (IOException e2)
{
;
}
}
}
}
```

Describe different types of sockets.
:Types of Sockets

| Socket type | Protocol | Description |
|---|---|---|
| SOCK_STREAM | Transmission Control Protocol (TCP) | The stream socket (SOCK_STREAM) interface defines a reliable connectionoriented service. Data is sent without errors or duplication and is received in the same order as it is sent. |
| SOCK_DGRAM | User Datagram Protocol (UDP) | The datagram socket (SOCK_DGRAM) interface defines a connectionless service for datagrams, or messages. Datagrams are sent as independent packets. The reliability is not guaranteed, data can be lost or duplicated, and datagrams can arrive out of order. However, datagram sockets have improved performance capability over stream sockets and are easier to use |
| SOCK_RAW | IP, ICMP, RAW | The raw socket (SOCK_RAW) interface allows direct access to lower-layer protocols such as Internet Protocol (IP). |

Note: The type of socket you use is determined by the data you are transmitting: • When you are transmitting data where the integrity of the data is high priority, you must use stream sockets. • When the data integrity is not of high priority (for example, for terminal inquiries), use datagram sockets because of their ease of use and higher performance capability.

**What are Datagram and Stream Protocols?**

Datagram communication: The datagram communication protocol, known as UDP (user datagram protocol), is a connectionless protocol, meaning that each time you send datagrams, you also need to send the local socket descriptor and the receiving socket's address. As you can tell, socket address is required each time a communication is made. Stream communication: The stream communication protocol is known as TCP (Transfer Control Protocol). Unlike UDP, TCP is a connection-oriented protocol. In order to do communication over the TCP protocol, a

connection must first be established between the pair of sockets. While one of the sockets listens for a connection request (server), the other asks for a connection (client). Once two sockets have been connected, they can be used to transmit data in both (or either one of the) directions.

Proxy Servers

A proxy server is a kind of buffer between your computer and the Internet resources you are accessing. They accumulate and save files that are most often requested by thousands of Internet users in a special database, called “cache”. Therefore, proxy servers are able to increase the speed of your connection to the Internet. The cache of a proxy server may already contain information you need by the time of your request, making it possible for the proxy to deliver it immediately. The overall increase in performance may be very high. Proxy servers can help in cases when some owners of the Internet resources impose some restrictions on users from certain countries or geographical regions. In addition to that, a type of proxy server called anonymous proxy servers can hide your IP address thereby saving you from vulnerabilities concerned with it.

Anonymous Proxy Servers

Anonymous proxy servers hide your IP address and thereby prevent your data from unauthorized access to your computer through the Internet. They do not provide anyone with your IP address and effectively hide any information about you. Besides that, they don’t even let anyone know that you are surfing through a proxy server. Anonymous proxy servers can be used for all kinds of Web-services, such as WebMail (MSN Hot Mail, Yahoo mail), web-chat rooms, FTP archives, etc. ProxySite.com will provide a huge list of public proxies.

**CLIENT SERVER PROGRAM**

```
import java.net.Socket;
import java.io.OutputStream;
import java.io.DataOutputStream;

public class clientg
{
  public static void main(String args[]) throws Exception
  {
    Socket sock = new Socket("127.0.0.1", 5000);
    String message1 = "Accept Best Wishes, Serverji";
```

```
    OutputStream ostream = sock.getOutputStream();
    DataOutputStream dos = new DataOutputStream(ostream);
    dos.writeBytes(message1);
    dos.close();
    ostream.close();
    sock.close();
  }
}
```

**SERVER PROGRAM**

```
import java.net.ServerSocket;
import java.net.Socket;
import java.io.InputStream;
import java.io.DataInputStream;

public class server
{
  public static void main(String args[]) throws Exception
  {
    ServerSocket sersock = new ServerSocket(5000);
    System.out.println("server is ready");  //  message to know the server is running

    Socket sock = sersock.accept();

    InputStream istream = sock.getInputStream();
    DataInputStream dstream = new DataInputStream(istream);

    String message2 = dstream.readLine();
    System.out.println(message2);
    dstream .close(); istream.close(); sock.close(); sersock.close();
  }
}
```

## JAVA DATABASE CONNECTIVITY

During programming you may need to interact with database to solve your problem. Java provides JDBC to connect to databases and work with it. Using standard library routines, you can open a connection to the database. Basically JDBC allows the integration of SQL calls into a general programming environment by providing library routines, which interface with the database. In particular, Java's JDBC has a rich collection of routines which makes such an interface extremely simple and intuitive.

Establishing A Connection

Oracle driver is loaded using the following code snippet:

```
Class.forName("sun.jdbc.odbc.JdbcOdbcDriver")
```

Creating JDBC Statements

A JDBC Statement object is used to send the SQL statements to the DBMS. It is entirely different from the SQL statement. A JDBC Statement object is an open connection, and not any single SQL Statement. You can think of a JDBC Statement object as a channel sitting on a connection, and passing one or more of the SQL statements to the DBMS.

RMI APPLICATIONS

Java provides RMI (Remote Method Invocation), which is "a mechanism that allows one to invoke a method on an object that exists in another address space. The other address space could be on the same machine or a different one. The RMI mechanism is basically an object-oriented RPC mechanism."

RPC (Remote Procedure Call) organizes the types of messages which an application can receive in the form of functions. Basically it is a management of streams of data transmission.

RMI applications often comprised two separate programs: a server and a client. A typical server application creates some remote objects, makes references to them accessible, and waits for clients to invoke methods on these remote objects. A typical client application gets a remote reference to one or more remote objects in the server and then invokes methods on them.

There are three processes that participate in developing applications based on remote method invocation.

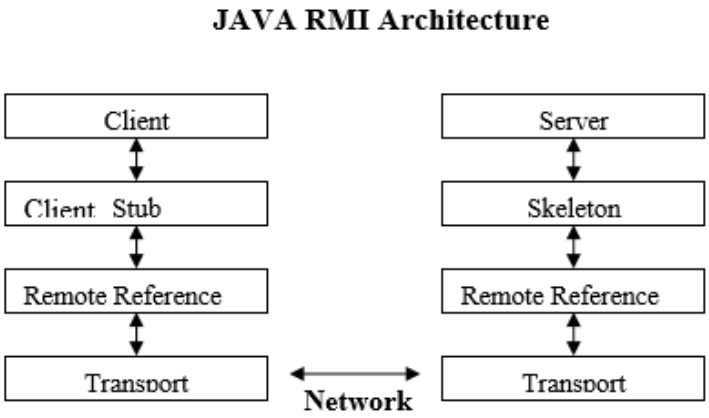
1. The Client is the process that is invoking a method on a remote object.
2. 2. The Server is the process that owns the remote object. The remote object is an ordinary object in the address space of the server process.

3. 3. The Object Registry is a name server that relates objects with names. Objects are registered with the Object Registry. Once an object has been registered, one can use the Object Registry to obtain access to a remote object using the name of the object.

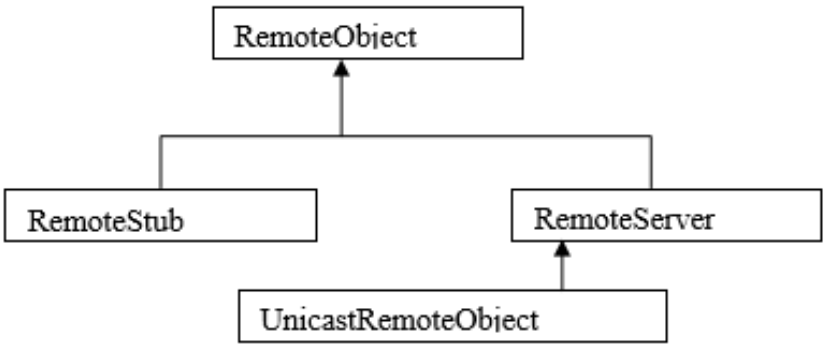
Remote Classes and Interfaces A Remote class is one whose instances can be used remotely. An object of such a class can be referenced in two different ways:

1. Within the address space where the object was constructed, the object is an ordinary object, which can be used like any other object. 2. Within other address spaces, the object can be referenced using an object handle While there are limitations on how one can use an object handle compared to an object, for the most part one can use object handles in the same way as an ordinary object.

RMI Architecture




Java RMI Architecture



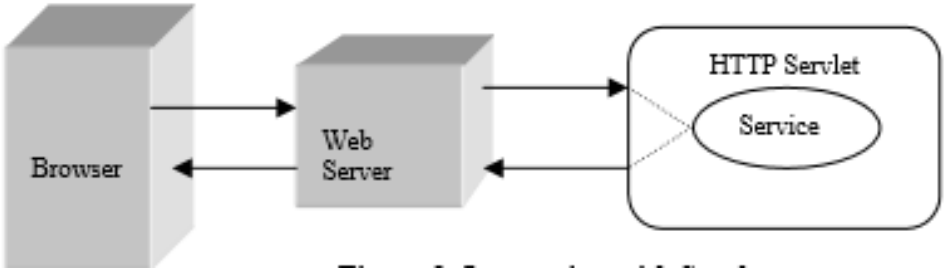

Remote Object Hierarchy

JAVA SERVLETS

A servlet is a class of Java programming language used to extend the capabilities of servers that host applications accessed via a request-response programming model. Although servlets can respond to any type of request, they are commonly used to extend the applications hosted by web servers. Java Servlet technology also defines HTTP-specific servlet classes. The javax.servlet and java.servlet.http packages provide interfaces and classes for writing servlets. All servlets must implement the Servlet interface, which defines life-cycle methods.

Servlet Life Cycle

The container in which the servlet has been deployed controls the life cycle of a servlet. When a request is mapped to a servlet, the container performs the following steps. Loads the servlet class. Creates an instance of the servlet class. Initializes the servlet instance by calling the init() method. When servlet is executed it invokes the service method, passing a request and response object. If the container needs to remove the servlet, it finalizes the servlet by calling the servlet's destroy method.




GET and POST Methods

The GET methods is a request made by browsers when the user types in a URL on the address line, follows a link from a Web page, or makes an HTML form that does not specify a METHOD. Servlets can also very easily handle POST requests, which are generated when someone creates an HTML form that specifies METHOD="POST".

The program code given below will give you some idea to write a servlet program:

```
import java.io.*;
import javax.servlet.*;
import javax.servlet.http.*;
public class SomeServlet extends HttpServlet
{
public void doGet(HttpServletRequest request, HttpServletResponse response)       throws
ServletException, IOException
```

```
{
// Use "request" to read incoming HTTP headers (e.g. cookies)
// and HTML form data (e.g. data the user entered and submitted)
// Use "response" to specify the HTTP response line and headers
// (e.g. specifying the content type, setting cookies).
PrintWriter out = response.getWriter();     // Use "out" to send content to browser
}
}
```

# Program to display data from database

```
import java.sql.*;
import java.io.*;
public class testconnection
{
public static void main(String a[]) throws Exception
{
Connection con;
ResultSet rs;
Class.forName("sun.jdbc.odbc.JdbcOdbcDriver");
con=DriverManager.getConnection("Jdbc:Odbc:mydb");
Statement stmt=con.createStatement();
rs=stmt.executeQuery("select * from login");
while(rs.next())
{
System.out.print(rs.getString(1));
System.out.print(rs.getString(2));
System.out.println();
}
}
}
```

# Program to insert data into database

```
import java.sql.*;
import java.io.*;
public class testconnection
{
public static void main(String a[]) throws Exception
{
Connection con;
ResultSet rs;
```

```
Class.forName("sun.jdbc.odbc.JdbcOdbcDriver");
con=DriverManager.getConnection("Jdbc:Odbc:mydb");
Statement stmt=con.createStatement();
rs=stmt.executeQuery("insert into login values(‘bk’,’singh’)");
}
}
```