

Classes & Objects

class: It is a user define type. Or class is a collection of fields & methods. Defining a class will not occupy any space in memory. It is like a template or a blue print. Collectively fields & methods are called members of the class.

General Form:

```
class <class-name>
{
    var-type var-name // instance variable or field
    var-type method-name(parameter[s]) {
    }
}
```

Instance variable: Means every object will contain its own copy.

Object: Object will be created from class. It occupies space in memory. It will contain all the members define by that class. To create an object we use new operator.

new: The new operator dynamically allocates memory for an object. It has this general form:
class-var = new classname();

Here, class-var is a variable of the class type being created. The classname is the name of the class that is being instantiated. The class name followed by parentheses specifies the constructor for the class. A constructor defines what occurs when an object of a class is created.

Example: BoxDemo.java

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

class BoxDemo {
    public static void main(String args[])
    {
        Box b1; // creating reference variable of Box
        // creating a Box object and assigning it to b1
        b1=new Box();
        b1.width=2; b1.height=3; b1.depth=4;
        double vol;
        vol = b1.width * b1.height * b1.depth;
        System.out.println("Volume of b1 :"+vol);
        Box b2 = b1; // assigning b1 reference to b2.
        vol = b2.width * b2.height * b2.depth;
        System.out.println("Volume of b2 :"+vol);
    }
}
```

Effect

The diagram illustrates the state of memory. On the left, there are two boxes labeled b1 and b2. Initially, b1 contains 'null'. Then, b1 points to a 'Box object' which has three fields: Width, Height, and Depth. Subsequently, b2 also points to the same 'Box object'. Red arrows indicate the sequence of these changes.

Save : BoxDemo.java

compile : javac BoxDemo.java → Box.class, BoxDemo.class

Run : java BoxDemo

Introducing Methods: Methods will be used to access class variables. It can contain complicated logic which can be accessed with the help of method. Methods can have access outside the class.

Example: BoxDemo1.java

<pre>class Box{ double width, height, depth; void setData(double w, double h, double d) { width=w; height=h; depth=d; } }</pre>	<pre>double volume(){ return width*height*depth; } // Box</pre>
---	---

Summery of Box: **Fields** : double width, height, depth
 Methods: void setData(double, double, double)
 double volume()

```
class BoxDemo1{
    public static void main(String args[]) {
        Box b1= new Box();
        b1.setData(4.0, 5.0, 6.0);
        double vol = b1.volume();
        System.out.println("Volume of box b1 :"+vol);
    }
}
```

Output: Volume of box b1 :120.0

Constructors :

- Java allows objects to initialize themselves when they are created. This automatic initialization is performed through the use of a constructor.
- It has the same name as the class in which it resides and is syntactically similar to a method. Once defined, the constructor is automatically called immediately after the object is created, before the new operator completes.
- Constructors do not have return type even void.
- The implicit return type is a class type itself.
- Default constructor is added by java even if user doesn't add any constructor.
- Default constructor is not added when user provides constructor.

Example: BoxConst.java

```
// import java.lang
class Box {
    double width, height, depth;
    //Box(){ } // zero argument constructor or default constructor
    Box(double w, double h, double d) { // parameterized constructor
        System.out.println("Inside Constructor");
        width=w; height=h; depth=d;
    }
    void Box(double w, double h, double d) { // ← Note here void Box() is a method
        System.out.println("Inside Method");
        width=w; height=h; depth=d;
    }
    double volume(){ return width*height*depth; }
} // Box
```

```
class BoxConst {
    public static void main(String args[]) {
        Box b1= new Box(2,3,4);
        double vol = b1.volume();
        System.out.println("Volume of box b1 :"+vol);
        Box b2 = new Box(); // gives error Box(){ }
        b2.Box(5.0, 6.0,7.0);
        vol = b2.volume();
        System.out.println("Volume of box b2 :"+vol);
    }
} // BoxConst
```

Note if in above Box class if you don't provide default constructor then we get error at statement:
Box b2 = new Box();

Statement : b2.Box(5.0, 6.0, 7.0);
We are calling method not constructor.

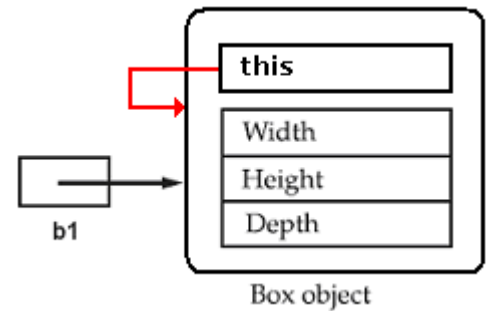
Assignment: Create a class Student with following members:
1) Instance variable : String name, result; double eng, math, science, tmarks, per;
2) Constructor : Student(String, double, double, double)
3) methods : void setData(String, double, double, double)
void calcResult(), void showResult()

Note: result should be like : First Div., Second Div, Third Div., or Fail. Assume marks of each subject out of 100. If marks in any subject less than 40 then the result should be fail.

The 'this' Keyword: this is always a reference to the object on which the method was invoked. You can use 'this' anywhere a reference to an object of the current class type is permitted.

Example:

```
Box(double width, double height, double depth) {
    this.width=width;
    this.height=height;
    this.depth=depth;
}
double volume(){
    return this.width* this.height * this.depth;
}
```



Instance Variable Hiding: when a local variable has the same name as an instance variable, the local variable hides the instance variable.

```
Box(double width, double height, double depth) {
    width=width; height=height; depth=depth;
}
```

Parameterized Constructors : Constructor with parameter.

Overloading Constructors: constructors overloading is done when you require creating objects in different ways. For Example

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

Now object can be created in 3 different ways:

```
Box b1 = new Box();
Box b2 = new Box(5);
Box b3 = new Box(2,3,4);
```

Garbage Collection: In language like C, C++ user allocates memory dynamically either using malloc() in c or use new in C++. To de-allocate memory after you finish with those variables program must explicitly de-allocate memory using delete() in c++. Memory management is done explicitly by the programmer. In Java memory management is done automatically (de-allocation). For this Java provides a mechanism called garbage collection. Garbage collection will work in background look for objects which do not have any reference. Such objects will be removed by garbage collection and memory is returned to system. We can also call garbage collection explicitly using System.gc(). This can be done for the server application where we cannot wait for garbage collection to be called automatically.

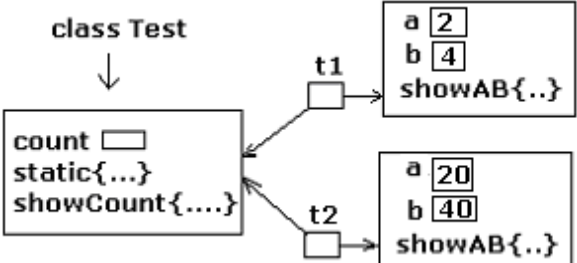
Overloading Methods: In Java it is possible to define two or more methods within the same class that share the same name, as long as their parameter declarations are different. When this is the case, the methods are said to be overloaded, and the process is referred to as method overloading. Method overloading is one of the ways that Java implements polymorphism. Parameters can differ either in number, order or type.

Understanding static : Normally a class member must be accessed only in conjunction with an object of its class. However, it is possible to create a member that can be used by itself, without reference to a specific instance. To create such a member, precede its declaration with the keyword static. When a member is declared static, it can be accessed before any objects of its class are created, and without reference to any object. You can declare both methods and variables to be static. The most common example of a static member is main(). main() is declared as static because it must be called before any objects exist. Instance variables declared as static are, essentially, global variables. When objects of its class are declared, no copy of a static variable is made. Instead, all instances of the class share the same static variable.

Example: TestStatic.java

<pre>class Test { static int count; // class variable int a, b; // instance var created with Object // static block called only once when class is loaded static { count = 100; // a=10; } Test(int a, int b){ this.a=a; this.b=b; count++; } //instance method void showAB(){ System.out.println("a="+a+" b="+b); } // static method can be use along with class name static void showCount(){ System.out.println("count="+count); //a++; } }</pre>	<pre>class TestStatic { public static void main(String args[]) { System.out.println("Inside main "); Test t1 = new Test(2,4); // count=101 Test t2 = new Test(20,40); // count=102 t1.showAB(); // 2 , 4 t1.showCount(); // 102 Test.showCount(); //102 t2.showAB(); // 20,40 t2.showCount(); // 102 Test.showCount(); // 102 } }</pre> <p>O/P:</p> <p>Inside main a=2 b=4 count=102 count=102 a=20 b=40 count=102 count=102</p>
--	--

Working: java TestStatic → JVM looks for class TestStatic.class, if found class gets loaded into memory. In memory main() gets created because it is defined as static. Next program execution begins from main().

<ul style="list-style-type: none">- "Inside main" gets printed. Next statement executes: Test t1 = new Test(2 ,4);- JVM looks for Test.class, if found class gets loaded.- static variables are created : count=0- static block gets executed : count =100- showCount() is created into the memory- object t1 is created using new Test(2,4); now count =100+1, a=2, b=4- object t2 is created using Test t2 = new Test(20,40);	<p>Note Test is already in memory. count =101+1, a=20, b=40 → Now the final value of count =102</p> 
---	---

Command-Line Arguments: In Java it is possible to pass arguments from the command prompt. Arguments will be sent as String array object to the main().

Example: CommandLine.java

<pre>class CommandLine { public static void main(String args[]) { for(int i=0;i<args.length; i++) { System.out.println(args[i]); } } }</pre>	<p>Run: java CommandLine hello hi fine 33</p> <p>Output: hello hi fine 33</p>
---	---

Factory Methods

<pre>class FactMethod { int a, b; private FactMethod(int a, int b){ this.a=a; this.b=b; } void display(){ System.out.println("a="+a + " b="+b); } }</pre>	<pre>class TestFactMethod { public static void main(String args[]){ FactMethod ob1 =new FactMethod(10,20); ob1.display(); } }</pre> <p>Output: Error at FactMethod ob1 =new FactMethod(10,20);</p>
---	--

Factory method:
When a class doesn't contain a visible constructor (i.e. class contains private constructor) in such a case object cannot be created using new outside the class. For this you can use factory method. Factory method is one which can be accessed from outside the class. Should be defined as static so that it can be used along with class name. Must return same class type object.

Above program modified using factory method:

<pre>class FactMethod { static FactMethod getObject(int a, int b) { // factory method FactMethod temp= new FactMethod(a, b); return temp; } } class TestFactMethod { public static void main(String args[]) { //FactMethod ob1 =new FactMethod(10,20); FactMethod ob1 = FactMethod.getObject(10,20); ob1.display(); } }</pre>

Inheritance

A class that is inherited is called a **superclass**. The class that does the inheriting is called a **subclass**. Therefore, a subclass is a specialized version of a superclass. It inherits all of the instance variables and methods defined by the superclass and adds its own, unique elements.

Example: TestInherit.java

<pre>class Test { int a, b; Test(){ } Test(int a, int b){ this.a=a; this.b=b; } void showAB(){ System.out.println("a="+a+" b="+b); } }</pre>	<pre>class TestNew extends Test { // int a, b comes from super class int c; TestNew(int a, int b, int c){ this.a=a; this.b=b; this.c=c; } void showC(){ System.out.println("c="+c); } void showABC(){ System.out.println("a="+ a+" b="+ b+" c="+c); } }</pre>
--	---

Summary of Test :

Fields : int a, b
Method: void showAB()

Summary of TestNew :

Fields : int a, b, c
Method: void showAB()
void showC()
void showABC()

<pre>class InheritDemo{ public static void main(String args[]) { TestNew ob1 = new TestNew(2,3,4); // sub-class object ob1.showAB(); ob1.showC(); ob1.showABC(); Test ob2 = new Test(10,20); // super-class object ob2.showAB(); //ob2.showC(); } }</pre>	Output: a=2 b=3 c=4 a=2 b=3 c=4 a=10 b=20
--	---

Example: BoxInheritance.java

<pre>class Box { double width, height, depth; void setData(double w, double h, double d){ width=w; height=h; depth=d; } Box(double w, double h, double d) { width=w; height=h; depth=d; } }</pre>	<pre>Box() { } double volume(){ return width*height*depth; } } // Box</pre>
---	---

<pre>class BoxWeight extends Box { double weight; BoxWeight(double w, double h, double d, double m) { width=w; height=h; depth=d; weight=m; } void showWeight(){ System.out.println("Weight of Box :"+weight); } } class BoxInheritance { public static void main(String args[]){ BoxWeight bw = new BoxWeight(2,3,4,100); double vol= bw.volume(); System.out.println("Volume of bw :"+vol); bw.showWeight(); } }</pre>	Output: Volume of bw :24.0 Weight of Box :100.0
--	--

Problem with BoxWeight class is that there is a **repetition of code for initialization**. The width, height, depth are initialized in super class constructor. Same code is written in sub-class. This repetition should not be there. Also if width, height, depth were defined as private, then we cannot access them from sub-class. The above class can be modified as under using super().

Using super : super is a keyword that can be used to access super class members (variables, methods & constructor). Note: Immediate super class

- **accessing super class variable:** super.variable_name
- **calling super class constructor:** super(parameter-list);
- **calling super class method :** super.method-name();

<pre>class BoxWeight extends Box { double weight; BoxWeight(double w, double h, double d, double m){ //width=w; height=h; depth=d; weight=m; super(w, h, d); // super() statement is used to call the super class constructor weight=m; } void showWeight(){ System.out.println("Weight of Box :"+weight); } }</pre>
--

Note: super() should be the first statement inside the sub-class constructor.

Types of Inheritance:

- 1) Single Level : e.g. class A{ } ← class B extends A { }
 - 2) Multi Level : e.g. class A{ } ← class B extends A { } ← class C extends B{ }
 - 3) Multiple : e.g. class A{ } class B { }
class C extends A, B <- wrong
- Java doesn't support multiple inheritances directly. That means Java supports inheritance that contains only one Single Super class. However similar structure is possible using interface.
- 4) Hierarchical : e.g. class A{ }
class B extends A { }
class C extends A{ }
 - 5) Hybrid : is also not supported.

Method Overriding: Method overriding is a concept of defining two methods with same name and same argument, one in the super class and another in the sub class.

- In a class hierarchy, when a method in a subclass has the same name and type signature as a method in its superclass, then the method in the subclass is said to override the method in the superclass and the process is called method overriding.
- When an overridden method is called from within a subclass, it will always refer to the version of that method defined by the subclass. The version of the method defined by the superclass will be hidden.

Creating a Multilevel Hierarchy

<pre>class A { int i=10; A(){ System.out.println("Inside A"); } void show(){ System.out.println("i="+i); } } class B extends A { int i=100; B(){ //super(); System.out.println("Inside B"); } void show(){ super.show(); System.out.println("super.i in B= "+super.i); //10 System.out.println("i in B = "+i); // 100 } }</pre>	<pre>class C extends B { int j=200; C(){ // super(); System.out.println("Inside C"); } void show() { super.show(); System.out.println("j="+j); } } class MultiLevel { public static void main(String args[]){ C obj = new C(); obj.show(); System.out.println("-----"); A supref; supref=obj; supref.show(); } }</pre>
--	---

Output: Inside A
Inside B
Inside C
i=10
super.i in B= 10
i in B = 100
j=200

i=10
super.i in B= 10
i in B = 100
j=200

Q. How Constructors get executed in multilevel hierarchy?

Ans: From Super class to sub-class.

Q. Difference between Method Overloading & Method Overriding

Ans:

Assignment: Create a class Employee :

Instance variables: empno, ename, basic, netsalary

Constructors: use necessary constructors to initialize.

Method: calcSalary() to calculate netsalary.

Next create a class Manager inheriting Employee class:

Instance variables: hra, da, ta assume da = 50% of basic.

hra, ta <- pass from command prompt.

Method: override method calcSalary() to calculate netsalary for Manager class.

Example: VehicleDemo.java

<pre>class Vehicle { String col=""; Vehicle(String c){ col=c; } Vehicle(){ } int cost(){ return 0; } int cc(){ return 0; } int avg(){ return 0; } String showColor(){ return col; } } class Splendor extends Vehicle{ Splendor(String c){ col=c; } int cost(){ return 50000; } int cc(){ return 125; } int avg(){ return 65; } }</pre>	<pre>class Pulsar extends Vehicle { Pulsar(String c){ col=c; } int cost(){ return 80000; } int cc(){ return 150; } int avg(){ return 45; } void startType(){ System.out.println("Self Start"); } } class VehicleDemo { public static void main(String args[]){ Vehicle v = new Vehicle(); System.out.println("Cost of Vehicle :"+v.cost()); System.out.println("CC of Vehicle :"+v.cc()); System.out.println("Average of Vehicle :"+v.avg()); System.out.println("Vehicle color:"+v.showColor()); Splendor s = new Splendor("Red"); System.out.println("Cost of Splendor :"+s.cost()); System.out.println("CC of Splendor :"+s.cc()); System.out.println("Average of Splendor :"+s.avg()); System.out.println("Splendor color: "+s.showColor()); Vehicle vref = new Pulsar("Black"); System.out.println("Cost of Pulsar :"+vref.cost()); System.out.println("CC of Pulsar :"+vref.cc()); System.out.println("Average of Pulsar "+vref.avg()); System.out.println("Pulsar color: "+vref.showColor()); //vref.startType(); ((Pulsar)vref).startType(); } }</pre>
---	---

A Superclass Variable Can Reference a Subclass Object :

- A reference variable of a superclass can refer a sub-class object. However using superclass reference variable we can access only those methods which are known to the superclass. Any new methods added by the subclass will not be accessed using super class reference variable
- It is important to understand that it is the type of the reference variable not the type of the object that it refers to that determines what members can be accessed.

```
Vehicle vref = new Pulsar("Black");
System.out.println("Cost of Pulsar :"+vref.cost());
System.out.println("CC of Pulsar :"+vref.cc());
System.out.println("Average of Pulsar "+vref.avg());
System.out.println("Pulsar color: "+vref.showColor());
//vref.startType();
((Pulsar)vref).startType();
```

Note: Here cost of Pulsar will be called and not of Vehicle.
 // vref.startType(); commented

Here startType() cannot be accessed using super class reference variable. If you think about it, this makes sense, because the superclass has no knowledge of what a subclass adds to it.

Dynamic Method Dispatch :

- Method overriding forms the basis for one of Java's most powerful concept dynamic method dispatch.
- Dynamic method dispatch is the mechanism by which a call to an overridden method is resolved at run time, rather than compile time.
- Dynamic method dispatch is important because this is how Java implements run-time polymorphism.
- We know an important principle: a superclass reference variable can refer to a subclass object. Java uses this fact to resolve calls to overridden methods at run time.

Here is how.

When an overridden method is called through a superclass reference, Java determines which version of that method to execute based upon the type of the object being referred to at the time the call occurs. Thus, this determination is made at run time. When different types of objects are referred to, different versions of an overridden method will be called.

```
Vehicle vref = new Pulsar("Black");
System.out.println("Cost of Pulsar :"+vref.cost());
System.out.println("CC of Pulsar :"+vref.cc());
System.out.println("Average of Pulsar "+vref.avg());
System.out.println("Pulsar color: "+vref.showColor());
```

Assignment: on applying Method Overriding

```
class Figure {
    double dim1, dim2;
    Figure(double d1, double d2){
        dim1=d1;
        dim2=d2;
    }
    double area(){
        return 0;
    }
}
```

class: Rectangle, Triangle by extending Figure class.

Abstract Classes:

- That is, sometimes you will want to create a superclass that only defines a generalized form that will be shared by all of its subclasses, leaving it to each subclass to fill in the details. Such a class determines the nature of the methods that the subclasses must implement.
- One way this situation can occur is when a superclass is unable to create a meaningful implementation for a method. This is the case with the class Vehicle used in the preceding example. The definition of cost() is simply a placeholder. It will not compute and display the cost of any type of object.
- You may have methods which must be overridden by the subclass in order for the subclass to have any meaning. Consider the class Splendor. It has no meaning if cost() is not defined
- If you require that certain methods be overridden by subclasses then specify this by using the abstract type modifier.

abstract type name(parameter-list);

- Any class that contains one or more abstract methods must also be declared abstract. To declare a class abstract, you simply use the abstract keyword in front of the class keyword at the beginning of the class declaration. There can be no objects of an abstract class. That is, an abstract class cannot be directly instantiated with the new operator. Such objects would be useless, because an abstract class is not fully defined. Also, you cannot declare abstract constructors, or abstract static methods.
- Although it is not possible to create an object of type Vehicle, you can create a reference variable of type Vehicle. The variable 'supref' is declared as a reference to Vehicle, which means that it can be used to refer to an object of any class derived from Vehicle. As explained, it is through superclass reference variables that overridden methods are resolved at run time.

Example : VehicleAbstractDemo.java

<pre>abstract class Vehicle { String col=""; Vehicle(String c){ col=c; } Vehicle(){ } abstract int cost(); abstract int cc(); abstract int avg(); String showColor(){ return col; } } class Splendor extends Vehicle{ Splendor(String c){ col=c; } int cost(){ return 50000; } int cc(){ return 125; } int avg(){ return 65; } }</pre>	<pre>class Pulsar extends Vehicle { Pulsar(String c){ col=c; } int cost(){ return 80000; } int cc(){ return 150; } int avg(){ return 45; } void startType(){ System.out.println("Self Start"); } } class VehicleAbstractDemo { public static void main(String args[]){ //Vehicle v = new Vehicle(); Vehicle vref; vref = new Pulsar("Black"); System.out.println("Cost of Pulsar :"+vref.cost()); System.out.println("CC of Pulsar :"+vref.cc()); System.out.println("Average of Pulsar "+vref.avg()); System.out.println("Pulsar color: "+vref.showColor()); //vref.startType(); ((Pulsar)vref).startType(); } }</pre>
---	--

Note: Abstract class can contain some methods with body. It is not necessary that abstract class should contain abstract method. Only thing is you cannot instantiate this class.

In below example if you uncomment then program **gives error** : Test is abstract; cannot be instantiated :- new Test();

```
abstract class Test{
    public static void main(String args[]){
        System.out.println("Hello how r u");
        //new Test();
    }
}
```

Using keyword final: In java final keyword can be used to declare variable, method & class.

- When **final** used with **variable** then such variable will **become constant**. Constants are generally declared in **Upper case** alphabet. You must also **initialize constant** variable. Initialization can be done while declaring or through constructor. e.g. final double PI=3.14;
- when **final** used along with method then such **method cannot be overridden** by the derived class.
- when **final** used along with class declaration then such **class cannot be super class** or a subclass from this class cannot be created.

Conclusion: final can be used to create constant, prevent overriding & prevent inheriting.

Example : FinalDemo.java

```
final class A { // to prevent Inheritance use final
    final double PI=3.14;
    int a, b;
    A(){
        //PI=3.14;
    }
    final void show() { // prevent overriding
        System.out.println("a="+a + " b="+b);
        //PI=3.14;
    }
}
class B //extends A
{
    int c;
    /*
    void show(){
        System.out.println("a="+a + " b="+b+ " c="+c);
    } */
}
class FinalDemo {
    public static void main(String args[]){
        A ob1 = new A();
        ob1.show();
    }
}
```

Final methods do early bindings :

Methods declared as final can sometimes provide a performance enhancement: The compiler is free to inline calls to them because it "knows" they will not be overridden by a subclass. When a small final method is called, often the Java compiler can copy the bytecode for the subroutine directly inline with the compiled code of the calling method, thus eliminating the costly overhead associated with a method call. In lining is only an option with final methods.

Normally, Java resolves calls to methods dynamically, at run time. This is called **late binding**. However, since final methods cannot be overridden, a call to one can be resolved at compile time. This is called **early binding**.

The Object Class: Object is a superclass of all other classes in Java. This means that a reference variable of type Object can refer to an object of any other class.

```
Pulsar p = new Pulsar();
Object o =p;
System.out.println("Hash code :"+ o.hashCode());
// System.out.println("CC :"+ o.cc());
```

Note: Every user define class automatically inherits Object class.

Assignment:

1) Modify class Figure as an abstract class by adding abstract method area().

Interfaces

- Using the keyword interface, you can fully abstract a class interface from its implementation.
- Interfaces are syntactically similar to classes, but they lack instance variables, and their methods are declared without any body.
- Once it is defined, any number of classes can implement an interface. Also, one class can implement any number of interfaces.
- To implement an interface, a class must create the complete set of methods defined by the interface. However, each class is free to determine the details of its own implementation.
- By providing the interface keyword, Java allows you to fully utilize the “one interface, multiple methods” aspect of polymorphism.
- Interfaces are designed to support dynamic method resolution at run time.
- Variables can be declared inside of interface declarations. They are implicitly final and static, meaning they cannot be changed by the implementing class. They must also be initialized with a constant value.

Defining an Interface

```
access interface name {
    return-type method-name1(parameter-list); // public abstract
    return-type method-name2(parameter-list);
    // ...
    return-type method-nameN(parameter-list);
    type final varname1 = value; // by default static final
    type final-varname2 = value;
    type final-varnameN = value;
}
```

Notice that the methods which are declared have no bodies. They end with a semicolon after the parameter list. They are, essentially, abstract methods.

Implementing Interfaces

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

Example : VehicleInterface.java

- 1) declare interface using keyword interface.
- 2) declare methods without body
- 3) create a class implementing interface by using implements keyword
- 4) override methods define in interface. Declare these methods as public because by default methods in interface are public abstract.

Note: Interface doesn't have instance variable, constructor and method with body.

```
interface Vehicle {
    int cost(); // by default public abstract int cost()
    int cc();
    int avg();
}
```

<pre>class Splendor implements Vehicle { String col; Splendor(String c){ col=c; } public int cost(){ return 50000; } public int cc(){ return 125; } public int avg(){ return 65; } void showColor(){ System.out.println("Color is :"+col); } }</pre>	<pre>class Pulsar implements Vehicle { String col; Pulsar(String c){ col=c; } public int cost() { return 80000; } public int cc() { return 150; } public int avg() { return 45; } String startType(){ return "Self Start"; } }</pre>
<pre>class VehicleInterface{ static void show(Vehicle v){ System.out.println("Cost :"+v.cost()); System.out.println("CC :"+v.cc()); System.out.println("Average :"+v.avg()); if (v instanceof Pulsar) System.out.println("Start type:"+ ((Pulsar)v).startType()); } static void showSplendor(Splendor s){ System.out.println("Splendor:"); System.out.println("Cost :"+s.cost()); } }</pre>	<pre> System.out.println("CC :"+s.cc()); System.out.println("Average :"+s.avg()); } public static void main(String args[]){ Pulsar p = new Pulsar("Black"); Splendor s = new Splendor("Red"); show(p); show(s); showSplendor(s); //showSplendor(p); } } // VehicleInterface</pre>

Note: `//showSplendor(p);` gives Error: `showSplendor(Splendor)` cannot be applied to `(Pulsar)`

Accessing Implementations Through Interface References

An interface reference can refer a sub-class object. The correct version will be called based on the actual instance of the interface being referred to. This is one of the key features of interfaces. The method to be executed is looked up dynamically at run time.

Variables in Interfaces :

<pre>interface I1 { double PI=3.14; // static final } abstract class Shape { abstract double area(); } class Circle extends Shape implements I1 { double radius; Circle(double radius){ this.radius=radius; } double area(){ return PI*radius*radius; } }</pre>	<pre>class CircleDemo { public static void main(String args[]) { Circle c1 = new Circle(2.5); System.out.println("Area of circle c1 "+ c1.area()); } }</pre>
---	--

Interfaces Can Be Extended

<pre>interface I1 { void show(); } interface I2 extends I1 { void display(); }</pre>	<pre>class InterfaceExtend implements I2 { public void show(){ System.out.println("From show :"); } public void display(){ System.out.println("From display :"); } public static void main(String args[]) { InterfaceExtend o1 = new InterfaceExtend(); o1.show(); o1.display(); } }</pre>
---	--

Packages

Definition: A package is a group of similar types of class, interfaces and sub-packages. Package 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.

- Packages are containers for classes that are used to keep the class name space compartmentalized.
- Java uses file system directories to store packages. For example, the **.class** files for any classes you declare to be part of **MyPackage** must be stored in a directory called **MyPackage**. Remember that case is significant, and the directory name must match the package name exactly.
- More than one file can include the same **package** statement.
- You can create a hierarchy of packages. To do so, simply separate each package name from the one above it by use of a period.

Advantage of Package

Package is used to categorize the classes and interfaces so that they can be easily maintained. Package provides access protection. Package removes naming collision.

Defining a Package

Package can be defined by using a **package** command as the first statement in a Java source file. If you omit the **package** statement, the class names are put into the default package, which has no name. This is the general form of the **package** statement:

```
package pkg;
```

e.g. package MyPackage;

General form of a multileveled package statement is shown here:

```
package pkg1[.pkg2[.pkg3]];
```

For example, a package declared `java.awt.image;` needs to be stored in `java\awt\image`,

Finding Packages and CLASSPATH

Packages are mirrored by directories. This raises an important question: How does the Java run-time system know where to look for packages that you create? The answer has two parts. First, by default, the Java run-time system uses the current working directory as its starting point. Thus, if your package is in the current directory, or a subdirectory of the current directory, it will be found. Second, you can specify a directory path or paths by setting the **CLASSPATH** environmental variable.

For example, consider the following package specification.

```
package MyPack;
```

In order for a program to find **MyPack**, one of two things must be true. Either the program is executed from a directory immediately above **MyPack**, or **CLASSPATH** must be set to include the path to **MyPack**. The first alternative is the easiest (and doesn't require a change to **CLASSPATH**), but the second alternative lets your program find **MyPack** no matter what directory the program is in.

A Short Package Example

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

To Compile: javac -d . Simple.java
To Run: java mypack.Simple

Output: Welcome to package

The -d is a switch that tells the compiler where to put the class file i.e. it represents destination. The dot (.) represents the current folder.

Access Protection

	Private	No modifier	Protected	Public
Same class	Yes	Yes	Yes	Yes
Same package subclass	No	Yes	Yes	Yes
Same package non-subclass	No	Yes	Yes	Yes
Different package subclass	No	No	Yes	Yes
Different package non-subclass	No	No	No	Yes

An Access Example :

// Save as A.java package p1; public class A{ class B extends A{ class C { }	// Save as p2.java package p2; class D extends p1.A { } class E { }
--	--

Rule: There can be only one public class in a java source file and it must be saved by the public class name.

Importing Packages : The **import** statements occur immediately following the **package** statement (if it exists) and before any class definitions.

General form: import pkg1[.pkg2].(classname)*;
Example: import java.util.Date;
import java.io.*;

Using Import	Without import
import java.util.*; class MyDate extends Date{ }	class MyDate extends java.util.Date{ }

Subpackage
Package inside the package is called the subpackage. It should be created to categorize the package further. Example: java.lang, java.io, java.net, java.awt. java.awt.event etc.

Static Import:
The static import feature of Java 5 facilitate the java programmer to access any static member of a class directly. There is no need to qualify it by the class name.

Simple Example of static import

import static java.lang.System.*; class StaticImportExample{ public static void main(String args[]){ out.println("Hello"); // Now no need of System.out out.println("Java"); } }
--

Output: Hello
Java

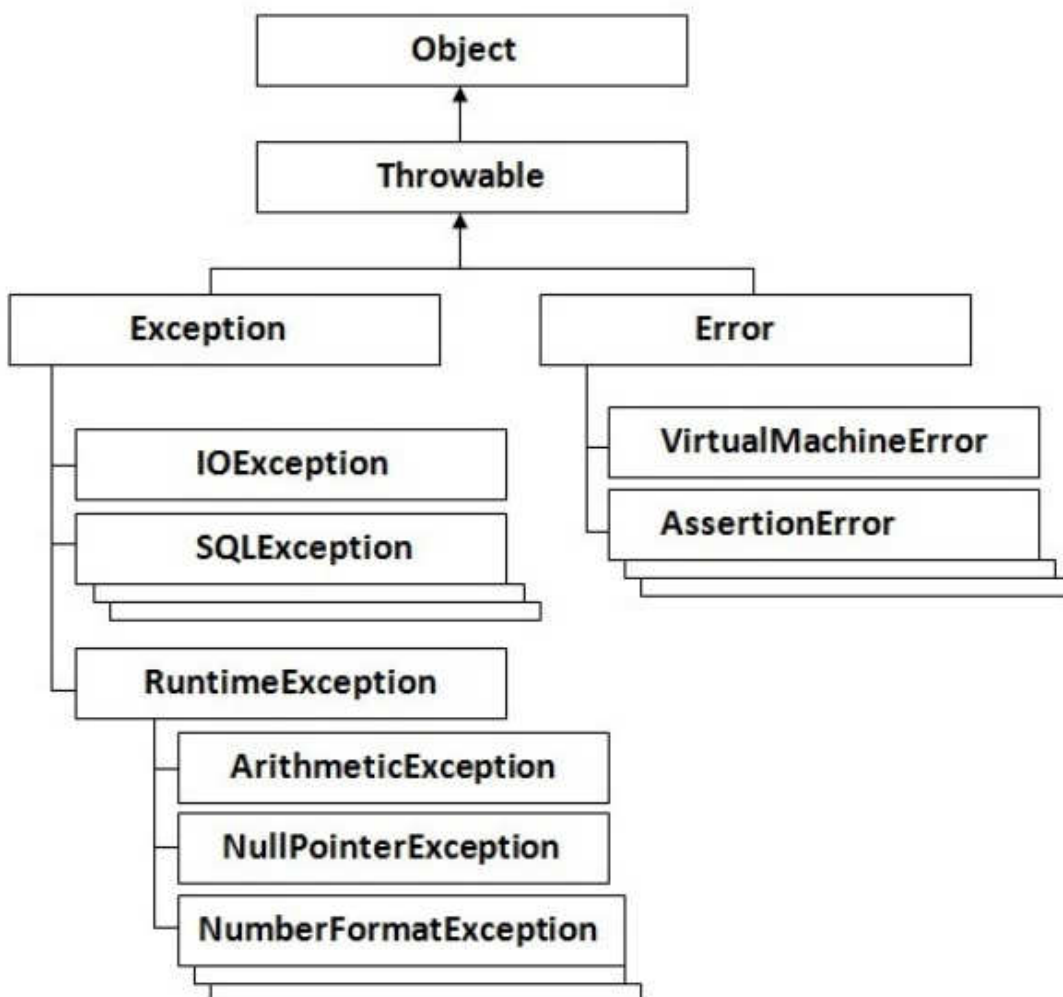
Exception Handling

- An exception is an abnormal condition that arises in a code sequence at run time.
- An exception is a run-time error
- When an exceptional condition arises, an object representing that exception is created and thrown in the method that caused the error.
- Exceptions can be generated by the Java run-time system, or they can be manually generated by your code
- Java exception handling is managed via five keywords: try, catch, throw, throws, and finally.
- To manually throw an exception, use the keyword throw.
- Any exception that is thrown out of a method must be specified as such by a throws clause.
- Any code that absolutely must be executed before a method returns is put in a finally block.

This is the general form of an exception-handling block:

```
try {
    // block of code to monitor for errors
}
catch (ExceptionType1 exOb) {
    // exception handler for ExceptionType1
}
catch (ExceptionType2 exOb) {
    // exception handler for ExceptionType2
}
finally {
    // block of code to be executed before try block ends
}
```

Throwable :All exception types are subclasses of the built-in class Throwable. Thus, Throwable is at the top of the exception class hierarchy.



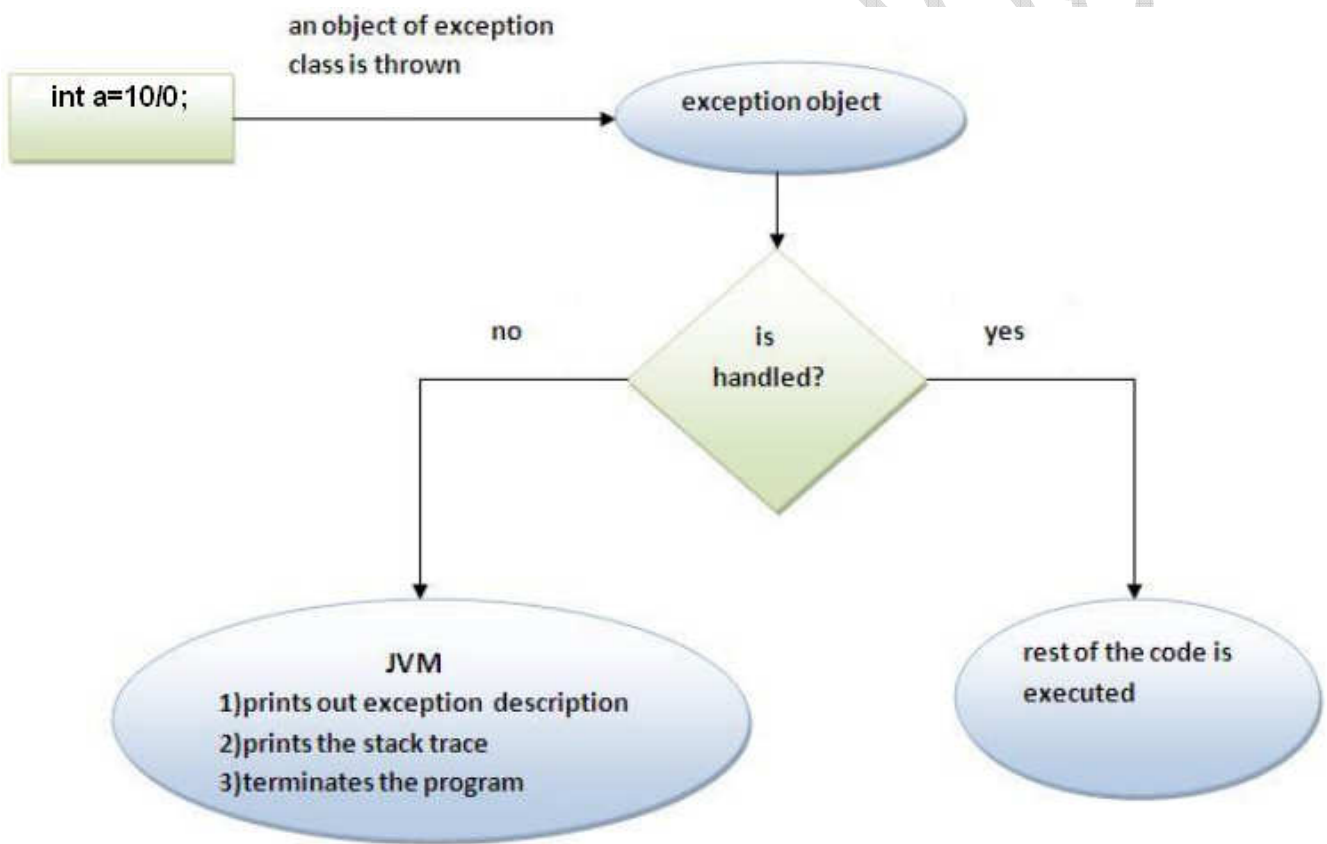
Uncaught Exceptions: When an exception is raised at run time, object representing that exception will be created by Java Runtime. This object is thrown inside the method where exception was raised. This exception object can be handled by the user by providing try, catch. If not then it will be handled by the default exception handler provided by Java runtime. This default exception handler catches exception object, displays error message and program terminates immediately. No further statements will be executed.

Example: Test.java

```
class Test {
    public static void main(String args[]){
        System.out.println("Inside main");
        int a=10/0; // ← ArithmeticException Object
        System.out.println("a="+a);
        System.out.println("Program Exiting....");
    }
}
```

Output: Inside main
Exception in thread "main" java.lang.ArithmeticException: / by zero
at Test.main(Test.java:4)

What happens behind the code int c=50/0;



Using try and catch: TestException.java

<pre>class TestException { public static void main(String args[]){ System.out.println("Inside main"); try{ int a=10/0; System.out.println("a="+a); }catch(ArithmeticException e){ System.out.println("Error divide by zero"); } System.out.println("Program exiting...."); } }</pre>	<p>Output: Inside main Error divide by zero Program exiting....</p>
--	--

Multiple catch Clauses : A try block can be followed by either single catch or multiple catch. However you should arrange them from child to parent. If you arrange from parent to child then program may not compile and you get error : Exception already caught.

Example :MultiCatch.java

```
class MultiCatch {
    public static void main(String args[]){
        int a, b, c;
        a=10;
        b=args.length;
        System.out.println("a="+a+" b="+b);
        try {
            c=a/b; // <-- ArithmeticException
            int d=Integer.parseInt(args[1]); // <-ArrayIndexOutOfBoundsException
            c=c+d;
            System.out.println("c="+c);
        }
        catch(ArrayIndexOutOfBoundsException e) {
            System.out.println("Array Index out of bounds ");
        }
        catch(Exception e) {
            System.out.println("Error due to divide by zero ");
            System.out.println("e = "+e);    // toString() is called
        }
        System.out.println("Program Exiting....");
    }
}
```

Output:

java MultiCatch 10 20 a=10 b=2 c=25 Program Exiting....	java MultiCatch 10 a=10 b=1 Array Index out of bounds Program Exiting....	java MultiCatch a=10 b=0 Error due to divide by zero e = java.lang.ArithmeticException: / by zero Program Exiting....
---	---	--

throw : So far, you have only been catching exceptions that are thrown by the Java run-time system. However, it is possible for your program to throw an exception explicitly, using the throw statement. The general form of throw is shown here:

```
throw ThrowableInstance;
```

Here, ThrowableInstance must be an object of type Throwable or a subclass of Throwable. The flow of execution stops immediately after the throw statement; any subsequent statements are not executed.

Example: ThrowDemo.java

```
class ThrowDemo{
    public static void main(String args[]){
        try{
            throw new ArithmeticException("Divide by zero");
            //System.out.print("Hello");
        }
        catch(Exception e) { System.out.println("error :"+e); }
        System.out.println("Exiting main");
    }
}
```

Output: error :java.lang.ArithmeticException: Divide by zero
Exiting main

- Note:-
1. statements after throw are commented.
 2. argument that is passed while creating object is the error message (i.e. "Divide by zero")

Exception propagation:

An exception is first thrown from the top of the stack and if it is not caught, it drops down the call stack to the previous method, If not caught there, the exception again drops down to the previous method, and so on until they are caught or until they reach the very bottom of the call stack. This is called exception propagation.

Rule: By default Unchecked Exceptions are forwarded in calling chain (propagated).

```
class Simple {
    void m(){    int data=50/0;    }
    void n(){    m();                }
    void p(){
        try{
            n();
        }
        catch(Exception e){
            System.out.println("exception handled");
        }
    }
    public static void main(String args[]) {
        Simple obj=new Simple();
        obj.p();
        System.out.println("normal flow...");
    }
}
```

```
graph TD
    m[m()]
    n[n()]
    p[p()]
    main[main()]
    m --- n
    n --- p
    p --- main
    m -- "exception occurred" --> out[ ]
    subgraph CallStack [Call Stack]
        m
        n
        p
        main
    end
```

Output: exception handled
normal flow...

In the above example exception occurs in m() method where it is not handled, so it is propagated to previous n() method where it is not handled, again it is propagated to p() method where exception is handled. Exception can be handled in any method in call stack either in main() method p() method n() method or m() method.

throws:

- If a method is capable of causing an exception that it does not handle, it must specify this behavior so that callers of the method can guard themselves against that exception.
- You do this by including a throws clause in the method’s declaration.
- A throws clause lists the types of exceptions that a method might throw.

This is the general form of a method declaration that includes a throws clause:

```
type method-name(parameter-list) throws exception-list {
    // body of method
}
```

Que) Which exception should we declare?

Ans) checked exception only, because:

- **unchecked Exception:** under your control so correct your code.
- **error:** beyond your control e.g. you are unable to do anything if VirtualMachineError or StackOverflowError occurs.

Example: ThrowsDemo.java

```
class ThrowsDemo {
    static void throwOne() throws IllegalAccessException, ArithmeticException {
        System.out.println("Inside throwOne");
        throw new ArithmeticException("Divide by Zero");
    }
    public static void main(String args[]) {
        try{
            throwOne(); // <-
        }catch(Exception e) { System.out.println("Caught inside main e : "+e); }
    }
}
```

Output: Inside throwOne
Caught inside main e : java.lang.ArithmeticException: Divide by Zero

Types of Exception: There are mainly two types of exceptions: checked and unchecked where error is considered as unchecked exception. The Sun Microsystems says there are three types of exceptions:

- 1. Checked Exception
- 2. Unchecked Exception
- 3. Error

What is the difference between checked and unchecked exceptions?

1) Checked Exception

The classes that extend Throwable class except RuntimeException and Error are known as checked exceptions e.g. IOException, SQLException etc. Checked exceptions are checked at compile-time.

```
class ThrowsDemo1 {
    static void throwOne(){
        System.out.println("Inside throwOne");
        throw new IllegalAccessException("Illegal Error");
    }
    public static void main(String args[]) {
        try{ throwOne(); // <-
        }catch(Exception e) { System.out.println("Caught inside main e : "+e); }
    }
}
```

Output: ThrowsDemo1.java:4: unreported exception java.lang.IllegalAccessException; must be caught or declared to be thrown throw new IllegalAccessException("Demo");

2) Unchecked Exception

The classes that extend RuntimeException are known as unchecked exceptions e.g. ArithmeticException, NullPointerException, ArrayIndexOutOfBoundsException etc. Unchecked exceptions are not checked at compile-time rather they are checked at runtime.

```
class ThrowsDemo2 {
    static void throwOne(){
        System.out.println("Inside throwOne");
        throw new ArithmeticException("Divide by zero");
    }
    public static void main(String args[]) {
        try{ throwOne(); // <-
        }catch(Exception e) { System.out.println("Caught inside main e : "+e); }
    }
}
```

Output: Inside throwOne
Caught inside main e : java.lang.ArithmeticException: Divide by zero

Thus IllegalAccessException is of type checked & ArithmeticException is unchecked type.

3) Error: Error is irrecoverable e.g. OutOfMemoryError, VirtualMachineError, AssertionError etc.

finally: The finally block will execute whether or not an exception is thrown. If an exception is thrown, the finally block will execute even if no catch statement matches the exception.

Example: FinallyDemo.java

<pre>class FinallyDemo { static void procA() { try{ System.out.println("inside procA"); throw new RuntimeException("Demo....."); } finally { System.out.println("procA's finally"); } } } //procA</pre>	<pre>static void procB() { try { System.out.println("Inside ProcB"); return; } finally { System.out.println("procB's Finally"); } } //procB</pre>
---	---

```
static void procC() {
    try {
        System.out.println("Inside procC");
    }
    finally {
        System.out.println("procC finally");
    }
} //procC
```

```
public static void main(String args[]) {
    try { procA(); }
    catch(Exception e) {
        System.out.println("Exception caught "
                           +e);
    }
    procB();
    procC();
} // main
} // FinallyDemo
```

Output: inside procA
procA's finally
Exception caught java.lang.RuntimeException: Demo.....
Inside ProcB
procB's Finally
Inside procC
procC finally

Creating Your Own Exception Subclasses

1. create a class by extends <subclass of Throwable>
2. define instance variables. Use constructor to initialize variables.
3. override toString() general form : public String toString(){ }

Example: OwnException.java

```
class MyException extends Exception {
    int a;
    MyException(int a){
        this.a=a;
    }
    public String toString(){
        return "Invalid Age :"+a;
    }
}
class OwnException {
    static void chkAge(int a) throws MyException {
        if(a>18 && a <36)
            System.out.println("Age Entered is correct :");
        else
            throw new MyException(a);    // ->
    }
    public static void main(String args[]) {
        try{
            chkAge(25);
            chkAge(15);    // <- MyException
        }
        catch(MyException e) {
            System.out.println("Error : "+e);
        }
        System.out.println("Program exiting .....");
    }
}
```

Output: Age Entered is correct :
Error : Invalid Age :15
Program exiting

- Assignment:**
1. Create your own exception class BankException with message Insufficient funds.
 2. Create a Bank class : with instance variable balance. Using constructor initialize with some amount say 50000.
 3. Define methods : void withdraw(double) , withdraw method should update balance. if balance is sufficient then display message: Collect your cash else: throw new BankException(double, double)
 4. Create BankDemo class with main() to test your application.

Multithreaded Programming

- Java provides built-in support for multithreaded programming.
- A multithreaded program contains two or more parts that can run concurrently. Each part of such a program is called a thread, and each thread defines a separate path of execution.
- Thus, multithreading is a specialized form of multitasking.
- Multithreading enables you to write very efficient programs that make maximum use of the CPU, because idle time can be kept to a minimum.
- There are two distinct types of multitasking: process-based and thread-based.

1) Process-based Multitasking (Multiprocessing)

Each process has its own address in memory i.e. each process allocates separate memory area. Process is heavyweight. Cost of communication between the processes is high. Switching from one process to another require some time for saving and loading registers, memory maps, updating lists etc.

2) Thread-based Multitasking (Multithreading)

Threads share the same address space. Thread is lightweight. Cost of communication between the thread is low.

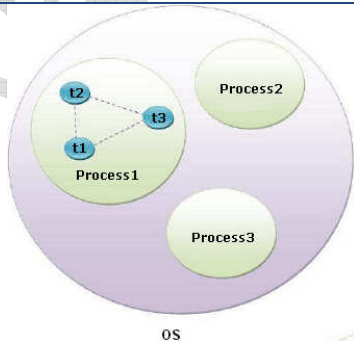
Note: At least one process is required for each thread.

What is Thread?

A thread is a lightweight subprocess, a smallest unit of processing. It is a separate path of execution. It shares the memory area of process.

As shown in figure, the thread is executed inside the process. There is context-switching between the threads. There can be multiple processes inside the OS and one process can have multiple threads.

Note: At a time only one thread is executed



Threads exist in several states:

- A thread can be running.
- It can be ready to run as soon as it gets CPU time.
- A running thread can be suspended, which temporarily suspends its activity.
- Suspended thread can then be resumed, allowing it to pick up where it left off.
- A thread can be blocked when waiting for a resource.
- At any time, a thread can be terminated, which halts its execution immediately. Once terminated, a thread cannot be resumed

Thread Priorities

Java assigns to each thread a priority that determines how that thread should be treated with respect to the others. Thread priorities are integers that specify the relative priority of one thread to another. A thread with more priority will be given more CPU clock cycles. Thread priority is from 1-10. Default is 5.

Synchronization:

When we want two threads to communicate and share a complicated data structure, such as a linked list, you need some way to ensure that they don't conflict with each other. That is, you must prevent one thread from writing data while another thread is in the middle of reading it. Java implements this using a mechanism called Monitor. A monitor as a very small box that can hold only one thread. Once a thread enters a monitor, all other threads must wait until that thread exits the monitor. In this way, a monitor can be used to protect a shared asset from being manipulated by more than one thread at a time.

The Thread Class and the Runnable Interface:

```
Runnable [ Interface]
|
|----- method : public abstract void run();
Thread [class]
```

Thread Class :- Thread class is derived from Runnable interface. It will implement run(). It also defines some more methods.

Constructors:

- **Thread():** creates a new thread with auto generated name
- **Thread(Runnable target) :** new thread with target whose run method is called.
- **Thread(Runnable target, String name) :**
- **Thread(String name):** creates a new thread with name you provide.

Methods:

1. **static Thread currentThread():** Returns a reference to the currently executing thread object.
2. **String getName() :** Returns this thread's name.
3. **int getPriority() :** Returns this thread's priority.
4. **boolean isAlive() :** Tests if this thread is alive.
5. **void join():** Waits for this thread to die.
6. **public void run():** If this thread was constructed using a separate Runnable run object, then that Runnable object's run method is called; otherwise, this method does nothing and returns.
7. **void setName(String new-name):** Changes the name of this thread to a new-name.
8. **void setPriority(int new-Priority):** Changes the priority of this thread.
9. **static void sleep(long Millis):** Causes the currently executing thread to sleep (temporarily cease execution) for the specified number of milliseconds.
10. **void start():** Cause this thread to begin execution; the Java Virtual Machine calls the run method of this thread.

The Main Thread:

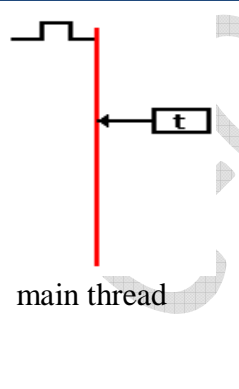
When a Java program starts up, one thread begins running immediately. This is usually called the *main thread* of your program, because it is the one that is executed when your program begins. The main thread is important for two reasons:

- It is the thread from which other "child" threads will be spawned.
- Often it must be the last thread to finish execution because it performs various shutdown actions.

The main thread can be controlled through a **Thread** object. To do so, you must obtain a reference to it by calling the method **currentThread()**, which is a **public static** member of **Thread**.

General form: static Thread currentThread()

Example: CurrentThread.java

	<pre>class CurrentThread { public static void main(String args[]){ System.out.println("Inside main():"); Thread t = Thread.currentThread(); System.out.println("Name of thread :"+t.getName()); System.out.println("Priority :"+t.getPriority()); System.out.println("t="+t); t.setName("Demo"); t.setPriority(8); System.out.println("t="+t); } }</pre>
---	--

Output: Inside main():
Name of thread :main
Priority :5
t=Thread[main,5,main]
t=Thread[Demo,8,main]

Creating a Thread:

1. By implement the Runnable interface.
2. By extending the Thread class, itself.

1) Implementing Runnable:

To implement Runnable, a class need only implement a single method called run(), which is declared like this: **public void run()**

Inside run(), you will define the code that constitutes the new thread. It is important to understand that run() can call other methods, use other classes, and declare variables, just like the main thread can. The only difference is that run() establishes the entry point for another, concurrent thread of execution within your program. This thread will end when run() returns.

- After you create a class that implements Runnable, you will instantiate an object of type Thread from within that class: **Thread(Runnable threadOb)**
- After the new thread is created, it will not start running until you call its start() method, which is declared within Thread: **start() executes a call to run().**

Example: On creating a new thread using Runnable interface

```
class ChildThread implements Runnable {
    Thread t;
    ChildThread(String s) {
        t = new Thread(this, s); // child thread gets created
        t.start();
    }
    public void run(){
        String child=t.getName();
        System.out.println("Thread Name :"+child);
        try{
            for(int i=5; i>0; i--) {
                System.out.println("Child Thread :"+child+"->" +i);
                t.sleep(500);
            }
        }catch(Exception e){ }
        System.out.println("Child thread Exiting ...");
    }
}

class RunnableDemo {
    public static void main(String args[]) throws InterruptedException {
        Thread t=Thread.currentThread();
        System.out.println(t);
        new ChildThread("One");
        for(int i=5; i>0; i--) {
            System.out.println("Thread : "+t.getName() + " " +i);
            t.sleep(1000);
        }
        System.out.println("Main Thread exiting ....");
    }
}
```

Output: Thread[main,5,main]
 Thread : main 5
 Thread Name :One
 Child Thread :One->5
 Child Thread :One->4
 Thread : main 4
 Child Thread :One->3
 Child Thread :One->2
 Thread : main 3
 Child Thread :One->1
 Child thread Exiting ...
 Thread : main 2
 Thread : main 1
 Main Thread exiting

Q. Can you call the run() method directly?
A. Yes, but run() is put onto same stack.

Extending Thread & Creating Multiple Threads :

```
class ChildThread extends Thread {
    ChildThread(String s) {
        setName(s); // super(s);
        start();
    }
    public void run(){
        String child=getName();
        System.out.println("Thread Name : " +child);
        try{
            for(int i=5; i>0; i--) {
                System.out.println("Child  : " +child+"->" +i);
                sleep(500); // Thread.sleep();
            }
        }catch(Exception e){ }
        System.out.println("Child thread Exiting ...");
    }
}
```

Output: Thread[main,5,main]
Thread : main 5
Thread Name :One
Child :One->5
Thread Name :Two
Child :Two->5
Child :One->4
Child :Two->4
Thread : main 4
Child :One->3
Child :Two->3
Child :One->2
Child :Two->2
Thread : main 3
Child :One->1
Child :Two->1
Child thread Exiting ...
Child thread Exiting ...
Thread : main 2
Thread : main 1
Main Thread exiting

```
class ExtendMultiDemo {
    public static void main(String args[]) {
        Thread t=Thread.currentThread();
        System.out.println(t);
        ChildThread one=new ChildThread("One");
        ChildThread two= new ChildThread("Two");
        try{
            for(int i=5; i>0; i--) {
                System.out.println("Thread : "+t.getName() + " " +i);
                t.sleep(1000);
            }
        }catch(Exception e){ }
        System.out.println("Main Thread exiting ....");
    }
}
```

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

However, this is hardly a satisfactory solution, How can one thread know when another thread has ended? Fortunately, Thread provides a means by which you can answer this question.

Using isAlive() and join()

- final boolean isAlive() :The isAlive() method returns true if the thread upon which it is called is still running.
- final void join() throws InterruptedException: This method waits until the thread on which it is called terminates.

IsAliveDemo.java & JoinDemo.java

Previous example can be modified by removing the for loop from main() method as under:

<pre>class IsAliveDemo { public static void main(String args[]) { Thread t=Thread.currentThread(); System.out.println(t); ChildThread one=new ChildThread("One"); ChildThread two= new ChildThread("Two"); while(one.isAlive() two.isAlive()) { } System.out.println("Main Thread exiting.."); } }</pre>	<pre>class JoinDemo { public static void main(String args[]) { Thread t=Thread.currentThread(); System.out.println(t); ChildThread one=new ChildThread("One"); ChildThread two= new ChildThread("Two"); try{ one.join(); two.join(); }catch(Exception e){ } System.out.println("Main Thread exiting.."); } }</pre>
---	--

Thread Priorities:

- Thread priorities are used by the thread scheduler to decide when each thread should be allowed to run.
- higher-priority threads get more CPU time than lower priority threads.
- A higher-priority thread can also preempt a lower-priority one. For instance, when a lower-priority thread is running and a higher-priority thread resumes (from sleeping or waiting on I/O, for example), it will preempt the lower-priority thread.

Example: HiLoPri.java

<pre>class clicker implements Runnable { long click = 0; Thread t; private boolean running = true ; public clicker(int p) { t = new Thread(this); t.setPriority(p); } public void run() { while(running) { click++; } } public void stop() { running = false; } public void start() { t.start(); } } // class clicker</pre>	<pre>class HiLoPri { public static void main(String args[]) { clicker hi = new clicker(Thread.NORM_PRIORITY +4); clicker lo = new clicker(Thread.NORM_PRIORITY - 2); lo.start(); hi.start(); try{ Thread.sleep(10000); } catch(InterruptedException e) { System.out.println("Main thread interrupted."); } lo.stop(); hi.stop(); try { hi.t.join(); lo.t.join(); } catch(InterruptedException e) { System.out.println("InterruptedException caught"); } System.out.println("Low priority thread :" + lo.click); System.out.println("High priority thread :" + hi.click); } } // HiLoPri</pre>
---	---

Synchronization

- When two or more threads need access to a shared resource, they need some way to ensure that the resource will be used by only one thread at a time. The process by which this is achieved is called synchronization.
- Key to synchronization is the concept of the monitor (also called a semaphore). A monitor is an object that is used as a mutually exclusive lock, or mutex. Only one thread can own a monitor at a given time. When a thread acquires a lock, it is said to have entered the monitor. All other threads attempting to enter the locked monitor will be suspended until the first thread exits the monitor. These other threads are said to be waiting for the monitor.

This is the general form of the synchronized statement:

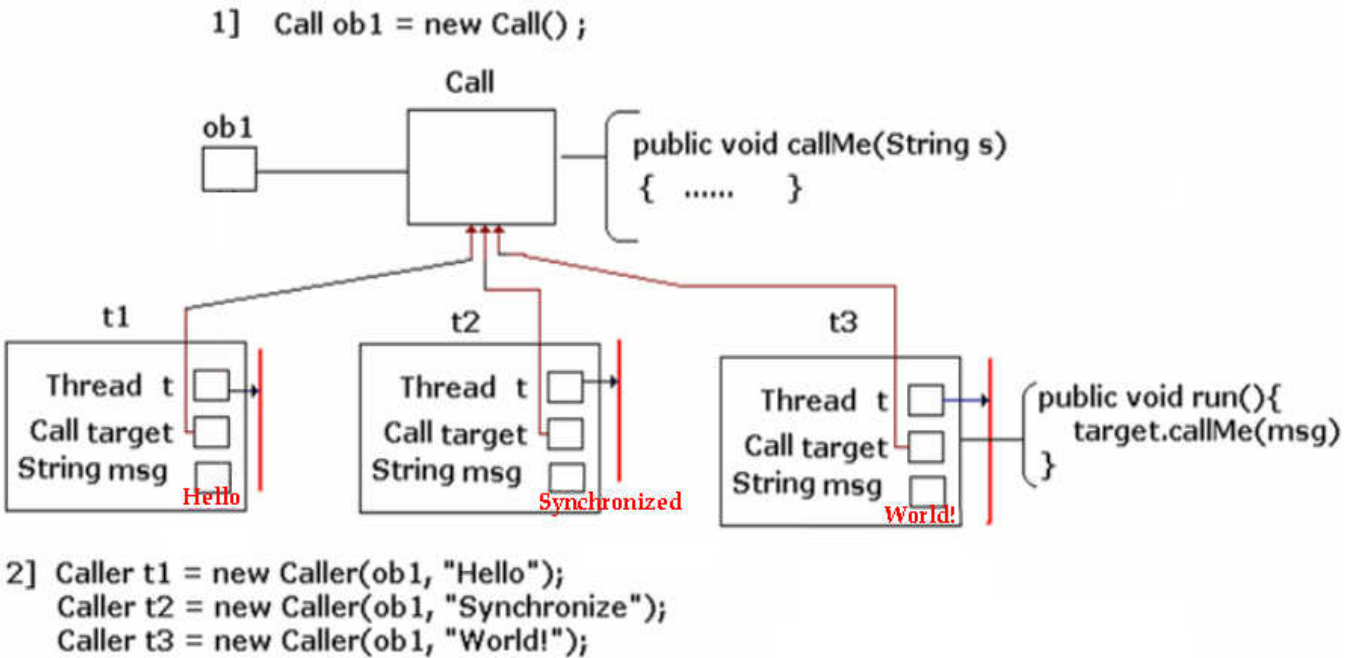
```
synchronized(object) { // statements to be synchronized }
```

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

Example: CalleMe.java

<pre>class Call { public void callme(String s) { try{ System.out.print("["+ s); // [Hello Thread.sleep(2000); }catch(Exception e) { } System.out.println("]"); //] } }</pre>	<pre>class Caller implements Runnable { Thread t; Call target; // target holds ref to Call object String msg; Caller(Call target, String s){ t = new Thread(this); this.target=target; msg=s; t.start(); } public void run(){ target.callme(msg); } }</pre>
<pre>class CallMe { public static void main(String args[]){ Call ob1 = new Call(); Caller t1 = new Caller(ob1,"Hello"); Caller t2 = new Caller(ob1,"Synchronize"); Caller t3 = new Caller(ob1,"World!"); } }</pre>	

Output: [Hello[Synchronize[World!]
]
]



The above example is not synchronized. To Synchronize we have 2 ways:

- 1) put **synchronized** before method name which you want to synchronize
- 2) synchronized block :

```
synchronized(target){  
    // method  
}
```

2nd option when you don't have source to modify.

CallMeSync.java : using synchronized

```
1) synchronized public void callme(String s) { .... }
2) public void run() {
    synchronized(target) {
        target.callme(msg);
    }
}
```

Assignment:

- Create a class Bank with instance variables: balance. Use necessary constructor to initialize balance. Define a synchronized method : withdraw(int amt) { }
- The withdraw method should check if balance is sufficient. If sufficient then display message "Your Transaction is under process" and call sleep method for 2 seconds.
- Next update balance (i.e. balance=balance-withdraw-amt)
- Next display message "Your Transaction is completed : Avail balance is :"
- Call sleep for 1 second and then display Thank You message!
- Next create a BankThread class. From its run() call the Bank withdraw().
- Next create BankDemo with public static void main().
- Inside main create Bank Object. Next Try to perform 4 types of transactions: Cash, Cheque, DD, and ATM
- (Cash, Cheque, DD, ATM should be the names passed as thread names to the BankThread class.

Inter-thread Communication

- **void wait():** Causes current thread to wait until another thread invokes the notify() method or the notifyAll() method for this object.
- **void notify() :** Wakes up a single thread that is waiting on this object's monitor.
- **void notifyAll():** Wakes up all threads that are waiting on this object's monitor.

All the above methods are defined by the class Object and must only be called from within synchronized method.

Refer Example: PC.java

<pre>class Q{ int n; synchronized int get(){ System.out.println("Got : "+ n); return n; } synchronized void put(int n) { this.n = n; System.out.println("Put : " +n); } } //Q</pre>	<pre>class Producer implements Runnable { Q q; Producer (Q q) { this.q = q; new Thread(this, "Producer").start(); } public void run() { int i=0; while (true){ q.put(i++); } } } // Producer</pre>
<pre>class Consumer implements Runnable { Q q; Consumer(Q q){ this.q = q; new Thread(this, "Consumer").start(); } public void run() { while(true){ q.get(); } } } // Consumer</pre>	<pre>class PC { public static void main(String args[]) { Q q = new Q(); new Producer(q); new Consumer(q); System.out.println("Press ctrl-C to stop"); } } // PC</pre>

Correct Implementation of Producer & Consumer using wait() & notify()

Refer Example: PCFixed.java

```
class Q {
    int n;
    boolean valueSet=false;
    synchronized int get() {
        if(!valueSet) try { wait(); } catch(InterruptedException e){ }
        System.out.println("Got : "+ n);
        valueSet=false;
        notify();
        return n;
    }

    synchronized void put(int n) {
        if(valueSet) try{ wait(); } catch(InterruptedException e){ }
        this.n = n;
        valueSet=true;
        System.out.println("Put : " +n);
        notify();
    }
}
```

STRING

In Java a string is a sequence of characters. But, unlike many other languages that implement strings as character arrays, Java implements strings as objects of type String. Objects of type **String are immutable**; once a String object is created, its contents cannot be altered. When you create a String object, you are creating a string that cannot be changed. Java defines a peer class of String, called StringBuffer, which allows strings to be altered, so all of the normal string manipulations are still available in Java.

How to create String object?

There are two ways to create String object:

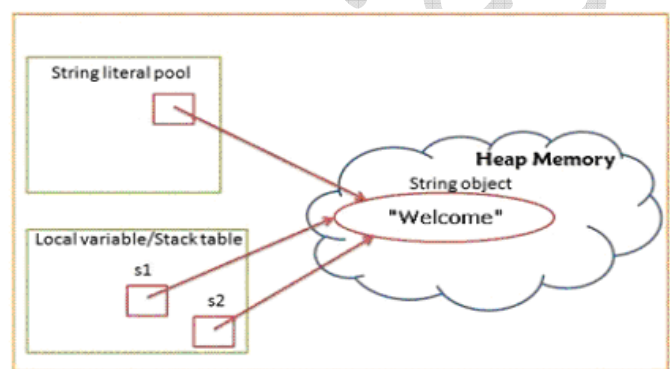
1. By string literal e.g. String s1 = "hello";
2. By new keyword e.g. String s = new String("hello");

Both s & s1 are objects

1) String literal:

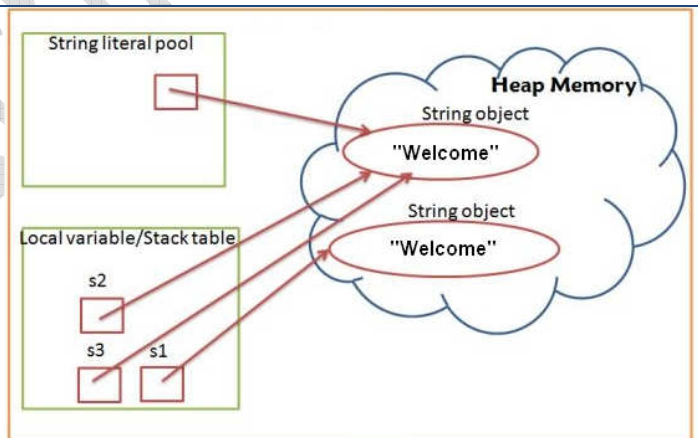
Each time you create a string literal, the JVM checks the string constant pool first. If the string already exists in the pool, a reference to the pooled instance returns. If the string does not exist in the pool, a new String object instantiates then is placed in the pool. For example :

```
String s1="Hello";  
String s2="Hello"; // now no new object will  
be created
```



String objects created with the new operator do not refer to objects in the string pool but can be made to using String's intern() method. The intern() returns an interned String, that is, one that has an entry in the global String literal pool. If the String is not already in the global String literal pool, then it will be added. For example:

```
String s1 = new String("Welcome");  
String s2 = s1.intern();  
String s3 = "Welcome";  
System.out.println(s2 == s3);
```



String Constructors

1. **String():** Initializes a newly created String object with an empty character sequence.
2. **String(char chars[]):** Initializes string from char array.
3. **String(char chars[], int startIndex, int numChars):** Here, startIndex specifies the index at which the subrange begins, and numChars specifies the number of characters to use.

```
Example: char chars[] = { 'a', 'b', 'c', 'd', 'e', 'f', 'g' };  
        String s1 = new String(chars);           // abcdefg  
        String s2 = new String(chars, 3, 3);     // def
```

4. **String(String strObj) :** Here, strObj is a String object.

```
Example: char c[] = { 'J', 'a', 'v', 'a' };  
        String s1 = new String(c);  
        String s2 = new String(s1);  
        System.out.println(s1); //Java  
        System.out.println(s2); //Java
```

5. **String(byte asciiChars[])** : Initializes string from byte type of array
6. **String(byte asciiChars[], int startIndex, int numChars)**

```
byte ascii[] = {65, 66, 67, 68, 69, 70 };
String s1 = new String(ascii);
System.out.println(s1); // ABCDEF
```

METHODS:

- 1) **length()**: Returns string length e.g. : int a = "Java".length(); // 4
- 2) **char charAt(int where)** : To extract a single character from a String.

String s="Hello"; char ch=s.charAt(0); System.out.println(ch) //H	String email="anil@hotmail.com" char ch=email.charAt(4); System.out.println(ch) //@
---	---

- 3) **public void getChars(int srcBegin, int srcEnd, char[] dst, int dstBegin)**: Copies characters from this string into the destination character array.
- srcBegin - index of the **first** character in the string to copy.
srcEnd - index **after** the last character in the string to copy.
dst - the destination array.
dstBegin - the start offset in the destination array.

String s="This is from InetSoft Solution"; char ch[]=new char[30]; s.getChars(13, 21, ch, 0); System.out.println(ch)//InetSoft	String email="anil@yahoo.com"; char ch[]=new char[30]; email.getChars(5, 10, ch, 0); System.out.println(ch)//yahoo
---	---

- 4) **public byte[] getBytes()**: Encodes this String into a sequence of bytes using the platforms default charset, storing the result into a new byte array.
- 5) **public char[] toCharArray()**:
- 6) **public boolean equals(Object anObject)** : Compares this string to the specified object. The result is true if and only if the argument is not null and is a String object that represents the same sequence of characters as this object.
- 7) **public boolean equalsIgnoreCase(String anotherString)**:
- 8) **public boolean regionMatches(boolean ignoreCase, int toffset, String other, int ooffset, int len)**: Tests if two string regions are equal. A substring of this String object is compared to a substring of the argument other
e.g.:
String s1="This is from InetSoft Solution";
String s2="This is from inetSoft Solution";
boolean b=s1.regionMatches(true,13, s2, 13, 8); // true
boolean b=s1.regionMatches(false,13, s2, 13, 8); // false
- 9) **public boolean startsWith(String prefix)**: Tests if this string starts with the specified prefix.
e.g. System.out.println("InetSoft".startsWith("In")); // true
- 10) **public boolean endsWith(String suffix)** : Tests if this string ends with the specified suffix.
e.g. String email="abc@yahoo.com";
System.out.println(email.endsWith(".com")) ; //true
- 11) **equals()** versus **==** :
equals() : checks for their contents. If contents are same it returns true else false.
== : checks for their references. If both the variables point to the same object then it will return true else false

<pre>String s1="Hello"; String s2=new String(s1); String s3="Good Bye"; String s4="Hello"; String s5=s1; System.out.println(s1+"equals " +s2+"--->" +s1.equals(s2)); System.out.println(s1+"== " +s2+"--->" +(s1==s2)); System.out.println(s3+"equals " +s4+"--->" +s3.equals(s4)); System.out.println(s1+"== " +s5+"--->" +(s1==s5)); System.out.println(s1+" == " + s4 + "--->" +(s1==s4));</pre>	<pre>graph LR s1[s1] --> H1[Hello] s2[s2] --> H2[Hello] s4[s4] --> H1 s5[s5] --> H1 s3[s3] --> G[Good Bye]</pre> <p>Note: s4 doesn't create new object. Instead it point to existing object s1. Also s5.</p>
<p>O/p: Hello equals Hello → true Hello == Hello → false Good Bye equals Hello → false Hello equals Hello → true Hello equals Hello → true</p>	<p>Content are same Points to different object Contents are different Points to same object Points to same object</p>

12) public int compareTo(String str) : Compares two strings and returns integer value as under :

Value	Meaning
Less than zero	The invoking string is less than s2
Greater than zero	The invoking string is greater than s2.
Zero	The two strings are equal.

e.g.

<pre>String s1 = new String("Hello"); String s2= new String("Hello"); String s3= new String("hello");</pre>	<pre>System.out.println("s1 , s2 " +s1.compareTo(s2); // 0 System.out.println("s1 , s2 " +s1.compareTo(s3); // -32 System.out.println("s1 , s2 " +s3.compareTo(s1); // +32</pre>
---	--

13) indexOf(int ch):

- a) int indexOf(int ch):**Returns the index within this string of the first occurrence of the specified character.

```
String s="abc@hotmail.com";
int i=s.indexOf('@');
System.out.println(i); //3
```
- b) int indexOf(int ch, int fromIndex):** Returns the index within this string of the first occurrence of the specified character, starting the search at the specified index.
- c) int indexOf(String str):** Returns the index within this string of the first occurrence of the specified substring.
- d) int indexOf(String str, int fromIndex):**Returns the index within this string of the first occurrence of the specified substring, starting at the specified index.
- e) int lastIndexOf(int ch):** Returns the index within this string of the last occurrence of the specified character.
- f) int lastIndexOf(int ch, int fromIndex):** Returns the index within this string of the last occurrence of the specified character, searching backward starting at the specified index.
- g) int lastIndexOf(String str):** Returns the index within this string of the rightmost occurrence of the specified substring.
- h) int lastIndexOf(String str, int fromIndex) :** Returns the index within this string of the last occurrence of the specified substring, searching backward starting at the specified index.

14) substring():

- a) String substring(int beginIndex):** Returns a new string that is a substring of this string from beginIndex to end of String.
- b) String substring(int beginIndex, int endIndex):** Returns a new string from beginIndex to endIndex that is a substring of this string.

```
String s="abc@hotmail.com";
String s1=s.substring(indexOf('@')+1,indexof('.'));
```

```
System.out.println(s1); //hotmail
```

15) String concat(String str):Concatenates the specified string to the end of this string.
String s1="Have a ";
String s2=s1.concat("Nice day");
System.out.println(s2); //Have a Nice day

16) String replace(char oldChar, char newChar): Returns a new string resulting from replacing all occurrences of oldChar in this string with newChar.
String s="Hello".replace('l', 'w');
System.out.println(s); // Hewwo

17) String replace(String target, String replacement): Replaces each substring of this string that matches the literal target string with the specified literal replacement string.

18) String trim(): Returns a copy of the string, with leading and trailing white space omitted.
String s=" Hello World ".trim();

19) String toLowerCase() & String toUpperCase()

- a) String toUpperCase():Converts all of the characters in this String to upper case using the rules of the default locale.
- b) String toUpperCase(Locale locale):Converts all of the characters in this String to upper case using the rules of the given Locale.

```
String s="Hello";  
String s1=s.toUpperCase() ;  
System.out.println(s1); //HELLO
```

StringBuffer

StringBuffer is a peer class of String that provides much of the functionality of strings. String represents fixed-length, immutable character sequences. In contrast, StringBuffer represents growable and writeable character sequences. StringBuffer may have characters and substrings inserted in the middle or appended to the end.

Constructors:

- 1) StringBuffer(): initial capacity of 16 characters.
- 2) StringBuffer(CharSequence seq):Constructs a string buffer that contains the same characters as the specified CharSequence.
- 3) StringBuffer(int capacity):Constructs a string buffer with specified initial capacity.
- 4) StringBuffer(String str):Constructs a string buffer initialized to the contents of the specified string.

Methods:

- 1) void ensureCapacity(int minimumCapacity):**Ensures that the capacity is at least equal to the specified minimum.
- 2) charAt() and setCharAt():** The value of a single character can be obtained from a StringBuffer via the charAt() method. You can set the value of a character within a StringBuffer using setCharAt().

Syntax: char charAt(int where) Syntax: void setCharAt(int where, char ch)

<pre>StringBuffer sb = new StringBuffer("Hello"); System.out.println("buffer before = " + sb); System.out.println("charAt(1) before = " + sb.charAt(1)); sb.setCharAt(1, 'i'); sb.setLength(2); System.out.println("buffer after = " + sb); System.out.println("charAt(1) after = " + sb.charAt(1));</pre>	<p>O/P</p> <p>buffer before = Hello charAt(1) before = e buffer after = Hi charAt(1) after = i</p>
--	---

3) append :- Appends string at the end of the string.

```
StringBuffer append(String str)
StringBuffer append(int i)
StringBuffer append(Object obj)
```

4) insert:-> Inserts string:

```
StringBuffer insert(String str)
StringBuffer insert(int i)
StringBuffer insert(Object obj)
```

StringBuffer sb = new StringBuffer("I Java!"); sb.insert(2, "like "); System.out.println(sb);	o/p I like Java!
---	---------------------

5) StringBuffer reverse(): Causes this character sequence to be replaced by the reverse of the sequence.

StringBuffer s = new StringBuffer("abcdef"); System.out.println(s); s.reverse(); System.out.println(s);	o/p abcdef fedcba
--	-------------------------

6) delete() and deleteCharAt():

StringBuffer delete(int start, int end): Removes the characters in a substring of this sequence.

StringBuffer deleteCharAt(int index): Removes the character at the specified position in this sequence.

StringBuffer sb = new StringBuffer("This is a test."); sb.delete(4, 7); System.out.println("After delete: " + sb); sb.deleteCharAt(0); System.out.println("After deleteCharAt: " + sb);	O/p: After delete: This a test. After delete CharAt: his a test.
---	--

7) StringBuffer replace(int start, int end, String str):

Replaces the characters in a substring of this sequence with characters in the specified String.

StringBuffer sb = new StringBuffer("This is a test."); sb.replace(5, 7, "was"); System.out.println("After replace: " + sb);	O/p: After replace: This was a test.
---	---

Wrapper classes

```
int i=33;  
Integer ob=new Integer(i);
```

- 1) Number
- 2) Double and Float
- 3) Byte, Short, Integer and Long
- 4) Character
- 5) Boolean
- 6) BigDecimal
- 7) BigInteger

java.lang.Object

java.lang.Number

java.math.BigDecimal

1) Number :- The abstract class Number is the super class of classes

- 1) byte byteValue() : Returns the value of the specified number as a byte.
- 2) abstract double doubleValue()
- 3) abstract float floatValue()
- 4) abstract int intValue()
- 5) abstract long longValue()
- 6) short shortValue()

```
Integer ob1=new Integer(3);  
Integer ob2=new Integer(3);
```

OB1+OB2 NOT ALLOWED
first convert object into primitive values

```
int a=(Integer(ob1)).intValue();  
int b=(Integer(ob2)).intValue();
```

Classes : Double and Float use for simple types =>double & float

Float: Constructors:

```
Float(double value) Float("3.33")  
Float(float value)  
Float(String s)
```

Double: Constructors:

```
Double(double value)  
Double(String s)
```

Methods Defined by Double & Float

- byte byteValue():
- static int compare(float f1, float f2):
- double doubleValue() :Returns the double value of this Float object.
- boolean equals(Object obj):
- float floatValue():
- int hashCode():unique code for every object
- int intValue(): Returns the value of this Float as an int (by casting to type int).
- boolean isInfinite():
- static boolean isInfinite(float v):
- boolean isNaN(): Returns true if this Float value is a Not-a-Number (NaN), false otherwise.
- long longValue():
- static float parseFloat(String s)
- short shortValue() : Returns the value of this Float as a short (by casting to a short).
- static String toHexString(float f) : Returns a hexadecimal string representation of the float argument.

DoubleDemo.java

```
class DoubleDemo
{
    public static void main(String arg[]) {
        Double d1=new Double(3.14159);
        Double d2=new Double("314159E-5");
        System.out.println(d1+"="+d2+"-->" +d1.equals(d2));
    }
} // Output      3.14159=3.14159-->true
```

Examples :->isInfinite() and isNaN() **InfNan.java**

```
class InfNan
{
    public static void main(String args[]) throws Exception {
        Double d1=new Double(1.0/0);
        Double d2=new Double(0.0/0);
        System.out.println(d1+"="+d1.isInfinite()+" , "+d1.isNaN());
        System.out.println(d2+"="+d2.isInfinite()+" , "+d2.isNaN());
    }
} /*      Output: Infinity=true, false
           NaN=false, true      */
```

Classes : Byte, Short, Integer and Long => byte, short, int long

Constructors:

Byte	Short	Integer	Long
Byte(byte value)	Short(short value)	Integer(int value)	Long(long value)
Byte(String s)	Short(String s)	Integer(String s)	Long(String s)

Methods:

```
static float parseFloat(String s)
static int  parseInt(String s)
static double parseDouble(String s)
static byte parseByte(String s)
```

Converting Numbers to and from Strings: ParseDemo.java

```
import java.io.*;
class ParseDemo {
    public static void main(String args[ ]) throws IOException {
        BufferedReader br = new  BufferedReader (new InputStreamReader(System.in));
        String str;
        int i=0;
        int sum=0;
        System.out.println("Enter numbers, 0 to quit.");
        do{
            str =br.readLine( );
            try{
                i = Integer.parseInt(str);
            }
            catch (NumberFormatException e) {
                System.out.println("Invalid format "+e);
            }
            sum +=i;
            System.out.println(" Current sum is: " + sum);
        }while(i !=0);
    } // main()
} /*      Output : Enter numbers, 0 to quit.
```

```
0
Current sum is: 0*/
```

Method: static String toBinaryString(int i)
 static String toHexString(float f)

```
System.out.println(Integer.toBinaryString(10)); //1010
System.out.println(Integer.toOctalString(10)); //12
System.out.println(Integer.toHexString(10)); //a
```

Class Character => char
Constructor: Character(char ch)

```
boolean b=Character.isDigit(10); //true
b=Character.isLetter(10); //false
b=Character.isLetter('c'); //true
```

Refer: Core and Advance java\All core examples\Lang\IsDemo.java

Boolean: boolean
Boolean(boolean value) Boolean(true)
Boolean(String s) Boolean("true")

Process :- public abstract class Process extends Object

Runtime:- The Runtime class encapsulates the run-time environment.

- You cannot instantiate a Runtime object. However, you can get a reference to the current Runtime object by calling the static method

```
Runtime r= Runtime.getRuntime( ).
```

- Once you obtain a reference to the current Runtime object, you can call several methods that control the state and behavior of the Java Virtual Machine.

Methods Defined by Runtime:

- void gc() : Runs the garbage collector.
- long totalMemory() : Returns the total amount of memory in the Java virtual machine.
- long freeMemory() : Returns the amount of free memory in the Java Virtual Machine(i.e run time system) in bytes.

MemoryDemo.java

```
class MemoryDemo {
    public static void main(String args[]) {
        Runtime r=Runtime.getRuntime();
        long mem1,mem2;
        Integer someints[]=new Integer[1000];
        System.out.println("Total memory is:"+r.totalMemory());
        mem1=r.freeMemory();
        System.out.println("Initial free memory:"+mem1);
        r.gc();
        mem1=r.freeMemory();
        System.out.println("free memory after garbage collection:"+mem1);
        for(int i=0;i<1000;i++)
            someints[i]=new Integer(i);
        mem2=r.freeMemory();
        System.out.println("free memory after garbage allocation:"+mem2);
        System.out.println("memory used by allocation:"+ (mem1-mem2));
        for(int i=0;i<1000;i++)
            someints[i]=null;
        r.gc();
        mem2=r.freeMemory();
    }
}
```



```
        System.out.println("free memory after collecting"+"discarded integers:"+mem2);
    }
}
```

Output: Total memory is:2031616
Initial free memory:1906592
free memory after garbage collection:1939688
free memory after garbage allocation:1923240
memory used by allocation:16448
free memory after collecting discarded integers:1939688 */

Executing Other Programs

Process exec(String command): Executes the specified string command in a separate process.

ExecDemo.java

```
class ExecDemo {
    public static void main(String args[]) {
        Runtime r=Runtime.getRuntime();
        Process p=null;
        try {
            p=r.exec("calc");
            p.waitFor();
        }
        catch(Exception e) {
            System.out.println("Error executing notepad");
        }
    }
}
```

Class System :-The System class holds a collection of static methods and variables.

```
class System {
    public static final PrintStream out;
    public static final PrintStream err;
    public static final InputStream in;
    ....
    ....
}
```

System.out.println(99)

The standard input, output, and error output of the Java run time are stored in the in, out, and err variables.

Methods defined by System :->

static String getProperty(String which) : Returns the property associated with which. A null object is returned if the desired property is not found.

ShowUserDir.java

```
class ShowUserDir{
    public static void main(String args[]) {
        System.out.println((System.getProperty("user.dir")));
    }
}
```

Refer: Environment Properties Page 412 Complete Reference Version 5

e.g.: java.class.version, java.compiler, java.home, os.arch, os.name, os.version, java.class.path, user.home, user.dir

class Class : Class encapsulates the run-time state of an object or interface. Objects of type Class are created automatically, when classes are loaded. You cannot explicitly declare a Class object. Generally, you obtain a Class object by calling the getClass() method defined by Object.

Instances of the class "Class" represent classes and interfaces in a running Java application.

Method forName():

```
public static Class forName(String className)
```

```
Class t = Class.forName("java.lang.Thread")
```

```
Class t = Class.forName("X")
```

```
or    X ob=new X();
```

A call to forName("X") causes the class named X to be initialized.

RTTI.java

```
class A {  
    int a;  
    float b;  
}  
  
class B extends A {  
    double c;  
}  
  
class RTTI {  
    public static void main(String arg[ ]) {  
        A superob=new A( );  
        B subob=new B( );  
        Class clObj;  
        clObj=superob.getClass( );  
        System.out.println("superob is object of type : "+clObj.getName( ));  
        clObj=subob.getClass( );  
        System.out.println("subob is object of type : "+clObj.getName( ));  
        clObj=clObj.getSuperclass( );  
        System.out.println("B is sub class of :"+clObj.getName( ));  
    }  
}
```

Output: superob is object of type : A
subob is object of type : B
B is sub class of :A

I/O

Most real applications of Java are not text-based, console programs. Rather they are graphically oriented applets that rely upon Java's Abstract Window Toolkit (AWT) for interaction with the user.

- Java does provide strong, flexible support for I/O as it relates to files and networks.
- Support for both I/O comes from Java's core API libraries, not from language keywords.

Stream:

A stream is a sequence of data. In Java a stream is composed of bytes. It's called a stream because it's like a stream of water that continues to flow.

The Predefined input Stream: Three streams are created for us automatically:

- 1) System.out: standard output stream
- 2) System.in: standard input stream
- 3) System.err: standard error

Java implements streams within class hierarchies defined in the java.io package. Java 2 defines 2 types of streams.

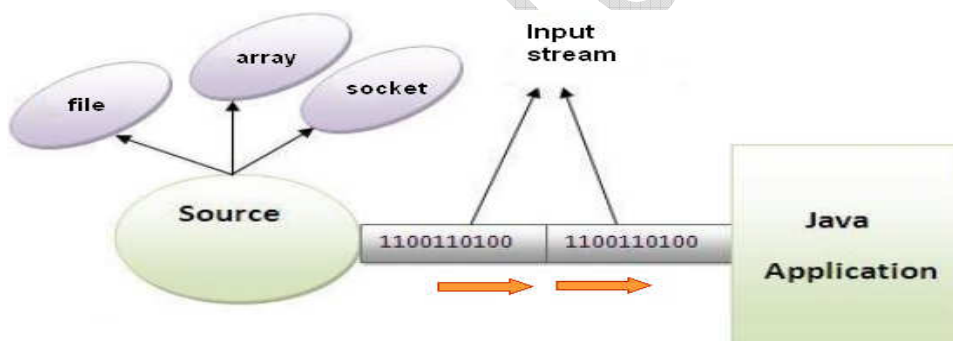
Byte Streams and Character Streams:

Byte streams: It provides a convenient means for handling input and output of bytes. Byte streams are used, for example, when reading or writing binary data. Byte streams are defined by using two class hierarchies. At the top are two abstract classes: **InputStream and OutputStream.**

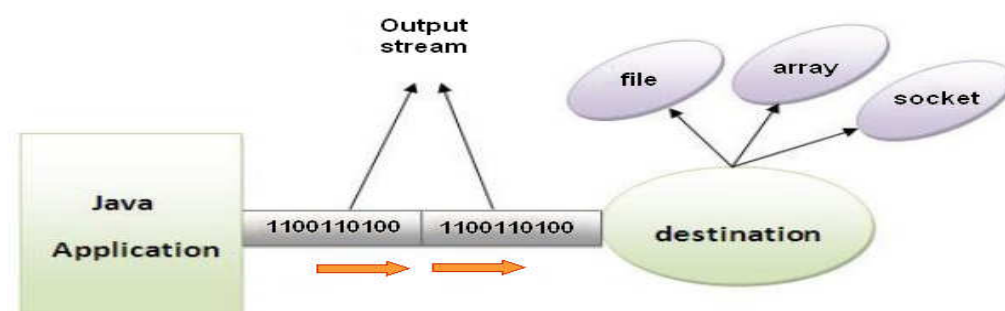
Character streams: It provides a convenient means for handling input and output of characters. Character streams are defined by using two class hierarchies. At the top are two abstract classes, **Reader and Writer.**

The Byte Stream Classes:

InputStream: Java application uses an input stream to read data from a source, it may be a file, an array, peripheral device or socket.



OutputStream: Java application uses an output stream to write data to a destination, it may be a file, an array, peripheral device or socket. OutputStream class is an abstract class.

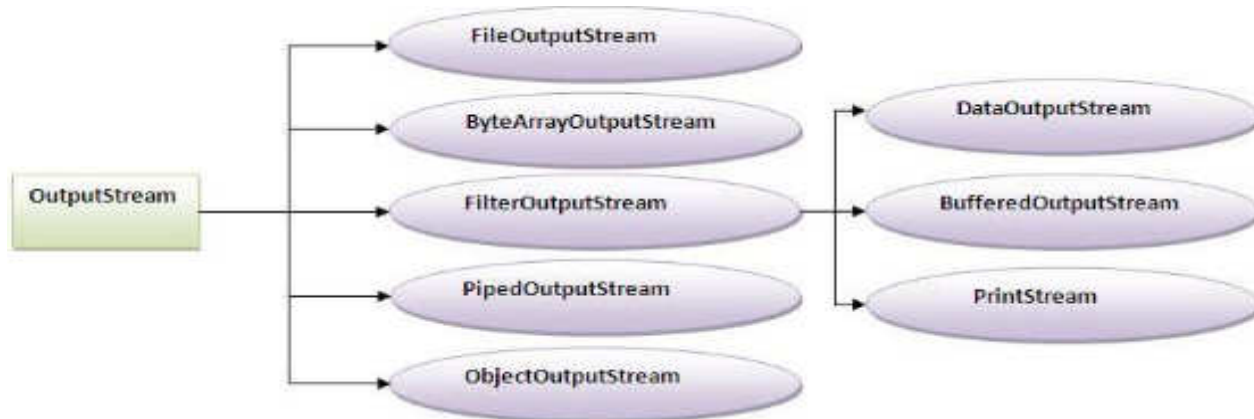


OutputStream class:

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

Commonly used methods of OutputStream class:

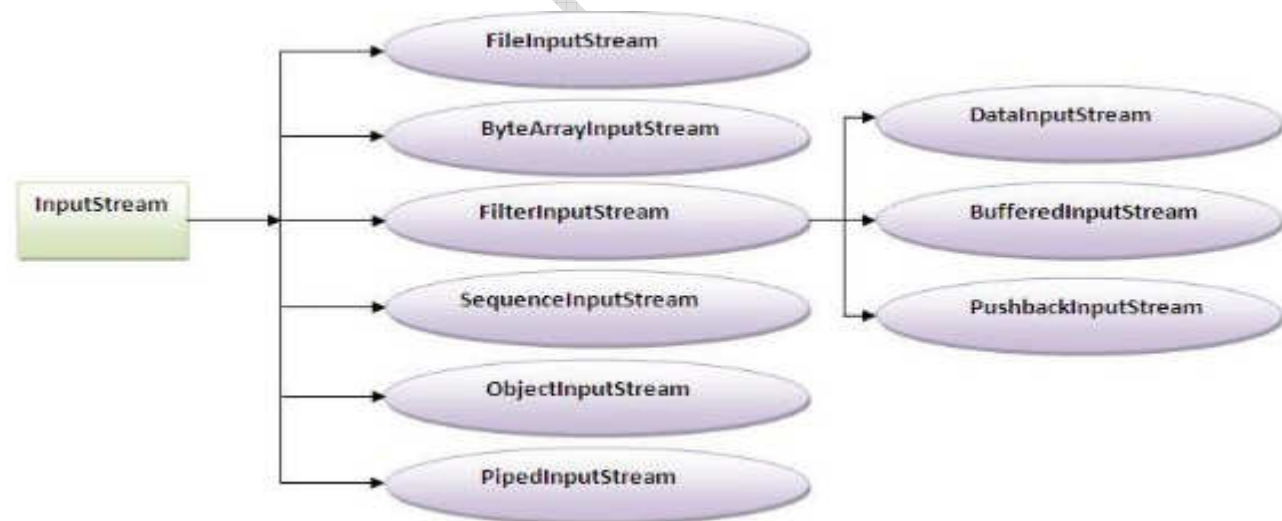
1. **public void write(int) throws IOException:** is used to write a byte to the current output stream.
2. **public void write(byte[])throws IOException:** is used to write an array of byte to the current output stream.
3. **public void flush()throws IOException:** flushes the current output stream.
4. **public void close()throws IOException:** is used to close the current output stream.

**InputStream class:**

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

Commonly used methods of InputStream class:

1. **public abstract int read()throws IOException:** reads the next byte of data from the input stream. It returns -1 at the end of file.
2. **public int available()throws IOException:** returns an estimate of the number of bytes that can be read from the current input stream.
3. **public void close()throws IOException:** is used to close the current input stream

**Reading data from keyboard:**

There are many ways to read data from the keyboard. For example:

- InputStreamReader
- Console
- Scanner
- DataInputStream etc.

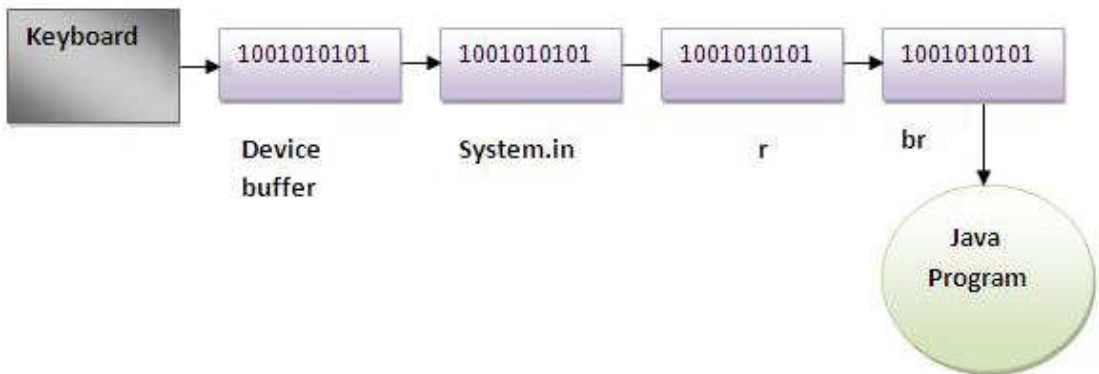
InputStreamReader class: InputStreamReader class can be used to read data from keyboard. It is a bridge from byte streams to character streams: It reads bytes and decodes them into characters.

```
public class InputStreamReader extends Reader { ... }
```

Constructor:
InputStreamReader(InputStream in): Create an InputStreamReader that uses the default charset.
e.g.: `InputStreamReader r= new InputStreamReader(System.in)`
System.in → a predefined input stream that is created automatically.

BufferedReader class:
BufferedReader class can be used to read text from a character input stream, buffering characters so as to provide for the efficient reading of characters, arrays, and lines.

Constructor: `BufferedReader(Reader r)`
e.g.: `BufferedReader br=new BufferedReader(r);`



We can also write: `BufferedReader br = new BufferedReader(new InputStreamReader(System.in));`

Reading Characters: Example : BRRead.java

<pre>import java.io.*; class BRRead { public static void main(String args[]) throws IOException{ char c; InputStreamReader r = new InputStreamReader(System.in); BufferedReader br=new BufferedReader(r); System.out.println("Enter Characters , 'q' to Quit"); do{ c=(char)br.read(); System.out.println(c); }while(c!='q'); } }</pre>	<p>Output:</p> <p>Enter Characters , 'q' to Quit 123abcq 1 2 3 a b c q</p>
---	---

Reading Strings: BRReadLine.java

<pre>import java.io.*; class BRReadLine{ public static void main(String args[]) throws IOException{ InputStreamReader r=new InputStreamReader(System.in); BufferedReader br=new BufferedReader(r); String name=""; while(!name.equals("stop")){ System.out.println("Enter data:"); name=br.readLine(); System.out.println("data is:"+name); } br.close(); r.close(); } }</pre>	<p>Output: Enter data: Anil data is: Anil Enter data: 10 data is: 10 Enter data: stop data is: stop</p>
--	--

Console class (I/O)

The Console class can be used to get input from the keyboard.

How to get the object of Console class?

System class provides a static method named console() that returns the unique instance of Console class.

Syntax: public static Console console(){ }

Commonly used methods of Console class:

- 1) **public String readLine():** is used to read a single line of text from the console.
- 2) **public String readLine(String fmt, Object... args):** it provides a formatted prompt then reads the single line of text from the console.
- 3) **public char[] readPassword():** is used to read password that is not being displayed on the console.
- 4) **public char[] readPassword(String fmt, Object... args):** it provides a formatted prompt then reads the password that is not being displayed on the console.

Example Console class that reads name of user:	Example Console class that reads password:
<pre>import java.io.*; class A{ public static void main(String args[]){ Console c=System.console(); System.out.println("Enter your name:"); String n=c.readLine(); System.out.println("Welcome "+n); } }</pre>	<pre>import java.io.*; class B{ public static void main(String args[]){ Console c=System.console(); System.out.println("Enter password:"); char[] ch =c.readPassword(); System.out.println("Password is "); for(char ch2:ch) System.out.print(ch2); } }</pre>

java.util.Scanner class:

There are various ways to read input from the keyboard, the java.util.Scanner class is one of them. The Scanner class breaks the input into tokens using a delimiter which is whitespace by default. It provides many methods to read and parse various primitive values.

Commonly used methods of Scanner class:

There is a list of commonly used Scanner class methods:

- **public String next():** it returns the next token from the scanner.
- **public String nextLine():** it moves the scanner position to the next line and returns the value as a string.
- **public byte nextByte():** it scans the next token as a byte.
- **public short nextShort():** it scans the next token as a short value.
- **public int nextInt():** it scans the next token as an int value.
- **public long nextLong():** it scans the next token as a long value.
- **public float nextFloat():** it scans the next token as a float value.
- **public double nextDouble():** it scans the next token as a double value.

Example of java.util.Scanner class:

Let's see the simple example of the Scanner class which reads the int, string and double value as an input:

<pre>import java.util.Scanner; class ScannerTest{ public static void main(String args[]){ Scanner sc=new Scanner(System.in); System.out.println("Enter your roll no:"); int rollNo=sc.nextInt(); System.out.println("Enter your name"); String name=sc.next(); System.out.println("Enter your fee"); double fee=sc.nextDouble(); System.out.println("Rollno:"+ rollNo+" name:"+ name+" fee:"+ fee); } }</pre>

FileInputStream and FileOutputStream (File Handling):

FileInputStream and FileOutputStream classes are used to read and write data in file. In another words, they are used for file handling in java.

FileOutputStream class:

A FileOutputStream is an output stream for writing data to a file. If you have to write primitive values then use FileOutputStream. Instead, for character-oriented data, prefer FileWriter. But you can write byte-oriented as well as character-oriented data.

Constructor:

- FileOutputStream(File file)
- FileOutputStream(File file, boolean append)
- FileOutputStream(String name)
- FileOutputStream(String name, boolean append)

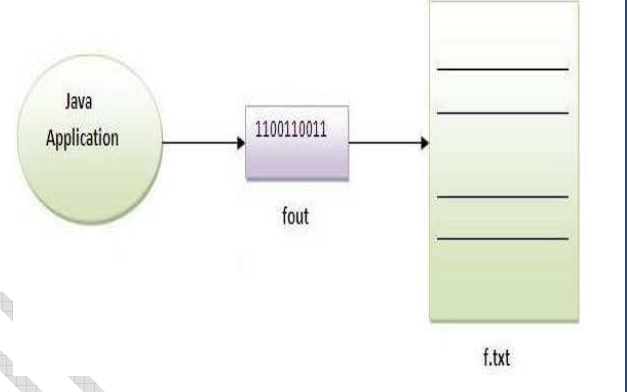
Method:

- void write(byte[] b): Writes b.length bytes from the specified byte array to this file output stream.
- void write(byte[] b, int off, int len): Writes len bytes from the specified byte array starting at offset off to this file output stream.
- void write(int b): Writes the specified byte to this file output stream.

Example of FileOutputStream class:

Simple program of writing data into the file

```
import java.io.*;
class Test{
    public static void main(String args[]){
        try{
            FileOutputStream fout=new FileOutputStream("f.txt");
            String s="Sachin Tendulkar is my favourite player";
            byte b[]=s.getBytes();
            fout.write(b);
            fout.close();
            System.out.println("success...");
        }catch(Exception e){ System.out.println(e); }
    }
}
```



Output: success...

FileInputStream class:

A FileInputStream obtains input bytes from a file. It is used for reading streams of raw bytes such as image data. For reading streams of characters, consider using FileReader. It should be used to read byte-oriented data. For example to read image etc.

Constructor:

- **FileInputStream(String filename)** :Creates a FileInputStream by opening a connection to an actual file, the file named by the path name filename in the file system.
e.g. FileInputStream fin=new FileInputStream("filename");
- **FileInputStream(File file)**: Creates a FileInputStream by opening a connection to an actual file, the file named by the File object file in the file system.
e.g. File f=new File("filename");
FileInputStream fin=new FileInputStream(f);

Methods:

- int available(): Returns the number of bytes that can be read from this file input stream without blocking.
- int read()
- int read(byte[] b): Reads up to b.length bytes of data from this input stream into an array of bytes.
- int read(byte[] b, int off, int len) : Reads up to len bytes of data from this input stream into an array of bytes.

Example of FileInputStream class: Simple program of reading data from the file

```
import java.io.*;
class SimpleRead{
    public static void main(String args[]){
        try{
            FileInputStream fin=new FileInputStream("f.txt");
            int i;
            while((i=fin.read())!=-1) System.out.println((char)i);
            fin.close();
        }catch(Exception e){ System.out.println(e); }
    }
}
```

Output: Sachin is my favourite player.

Example of Reading the data of current java file and writing it into another file

We can read the data of any file using the `FileInputStream` class whether it is java file, image file, video file etc. In this example, we are reading the data of `FileCopy.java` file and writing it into another file `NewFile.txt`.

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

ByteArrayOutputStream class:

In this stream, the data is written into a byte array. The buffer automatically grows as data is written to it. Closing a `ByteArrayOutputStream` has no effect.

Commonly used Constructors of ByteArrayOutputStream class:

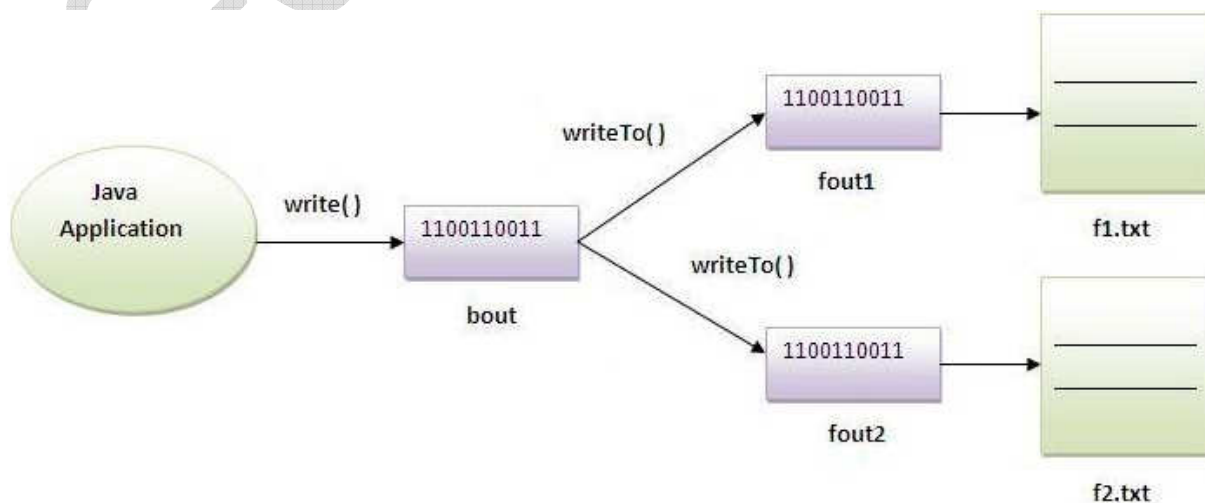
- 1) **`ByteArrayOutputStream()`**: creates a new byte array output stream with the initial capacity of 32 bytes, though its size increases if necessary.
- 2) **`ByteArrayOutputStream(int size)`**: creates a new byte array output stream, with a buffer capacity of the specified size, in bytes.

Commonly used Methods of ByteArrayOutputStream class:

- 1) **`public synchronized void writeTo(OutputStream out) throws IOException`**: writes the complete contents of this byte array output stream to the specified output stream.

Example of ByteArrayOutputStream class:

```
import java.io.*;
class S{
    public static void main(String args[]) throws Exception{
        FileOutputStream fout1=new FileOutputStream("f1.txt");
        FileOutputStream fout2=new FileOutputStream("f2.txt");
        ByteArrayOutputStream bout= new ByteArrayOutputStream();
        bout.write(239);
        bout.writeTo(fout1);
        bout.writeTo(fout2);
        bout.flush();
        bout.close();
        System.out.println("success...");
    }
}
```

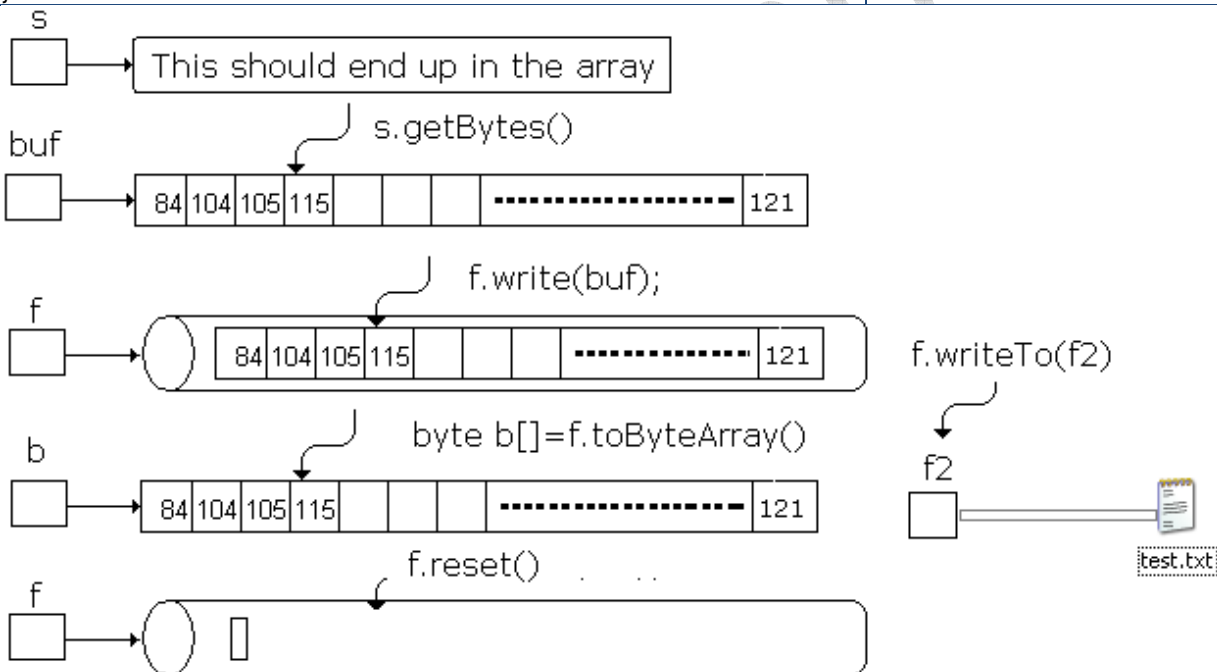


Example: `ByteArrayOutputStreamDemo.java`

```
class ByteArrayOutputStreamDemo {
    public static void main(String args[]) throws IOException {
        ByteArrayOutputStream f = new ByteArrayOutputStream();
        String s = "This should end up in the array";
        byte buf[] = s.getBytes();
        f.write(buf);
        System.out.println("Buffer as a string");
        System.out.println(f.toString());
        System.out.println("Into byte array");
        System.out.println("Size :"+f.size());
        byte b[] = f.toByteArray();
        for(int i=0; i<b.length; i++)
            System.out.print((char)b[i]);
        System.out.println("\nTo an OutputStream() ");
        OutputStream f2 = new FileOutputStream("test.txt");
        f.writeTo(f2);
        f2.close();
        System.out.println("Doing a reset");
        f.reset();
        System.out.println("Size :"+f.size());
        System.out.println("f :"+f.toString());
        for(int i=0; i<3; i++)
            f.write('X');
        System.out.println("f :"+f.toString());
    }
}
```

Output:

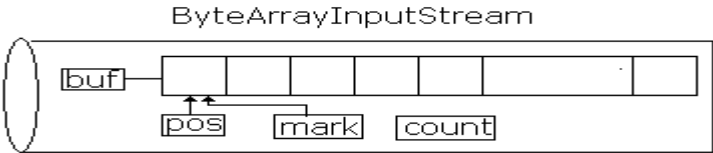
Buffer as a string
This should end up in the array
Into byte array
Size :31
This should end up in the array
To an OutputStream()
Doing a reset
Size :0
f :
f:XXX



Class `ByteArrayInputStream`

public class `ByteArrayInputStream` extends [InputStream](#) :

A `ByteArrayInputStream` contains an internal buffer that contains bytes that may be read from the stream. An internal counter keeps track of the next byte to be supplied by the `read` method. Closing a `ByteArrayInputStream` has no effect. The methods in this class can be called after the stream has been closed without generating an `IOException`.



Fields:

protected byte[]buf	An array of bytes that was provided by the creator of the stream
protected int count	The index one greater than the last valid character in the input stream buffer.
protected int mark	The currently marked position in the stream.
protected int pos	The index of the next character to read from the input stream buffer.

Constructor:

ByteArrayInputStream(byte[] buf)	Creates a ByteArrayInputStream so that it uses buf as its buffer array.
ByteArrayInputStream(byte[] buf, int offset, int length)	Creates ByteArrayInputStream that uses buf as its buffer array.

Methods:

int available()	Returns the number of bytes that can be read from this input stream without blocking.
void close()	Closing a ByteArrayInputStream has no effect.
void mark(int readAheadLimit)	Set the current marked position in the stream.
boolean markSupported()	Tests if this InputStream supports mark/reset.
int read()	Reads the next byte of data from this input stream.
int read(byte[] b, int off, int len)	Reads up to len bytes of data into an array of bytes from this input stream.
void reset()	Resets the buffer to the marked position.
long skip(long n)	Skips n bytes of input from this input stream.

Example: MyByteArrayInputStreamDemo.java

<pre>import java.io.*; class MyByteArrayInputStreamDemo{ public static void main(String[] args) { String s="abcdefghijk"; byte[] b= s.getBytes(); MyByteArrayInputStream in; in = new MyByteArrayInputStream(b); System.out.println("From show :"); in.show(); int c; b=null; while((c=in.read())!=-1){ System.out.print((char)c); } System.out.println("\nEnd : "+c); System.out.println("From show :"); in.show(); } }</pre>	<pre>class MyByteArrayInputStream extends ByteArrayInputStream { MyByteArrayInputStream(byte b[]){ super(b); } void show(){ System.out.println("Count :"+count); System.out.println("Pos :"+pos); System.out.println("mark :"+mark); for(int i=0; i<buf.length; i++) System.out.print((char)buf[i]+" "); System.out.println("\n"); } }</pre>
--	---

Output: From show : Count :11 Pos :0 mark :0 a b c d e f g h i j k abcdefghijk End : -1	From show : Count :11 Pos :11 mark :0 a b c d e f g h i j k
---	---

PushbackInputStream:

- 1) void unread(int ch)
- 2) void unread(byte[] buffer)
- 3) void unread(byte[] buffer, int offset, int numChars)

SequenceInputStream class: SequenceInputStream class is used to read data from multiple streams.

Constructors of SequenceInputStream class:

- **SequenceInputStream(InputStream s1, InputStream s2):** creates a new input stream by reading the data of two input stream in order, first s1 and then s2.
- **SequenceInputStream(Enumeration e):** creates a new input stream by reading the data of an enumeration whose type is InputStream.

Simple example of SequenceInputStream class

In this example, we are printing the data of two files f1.txt and f2.txt.

```
import java.io.*;
class Simple{
    public static void main(String args[]) throws Exception{
        FileInputStream fin1=new FileInputStream ("f1.txt");
        FileInputStream fin2=new FileInputStream ("f1.txt");
        SequenceInputStream sis=new SequenceInputStream(fin1,fin2);
        int i;
        while(i=sis.read()!=-1) {
            System.out.println((char)i);
        }
    }
}
```

Example of SequenceInputStream class that reads the data from multiple files using enumeration

If we need to read the data from more than two files, we need to have these informations in the Enumeration object. Enumeration object can be get by calling elements method of the Vector class. Let's see the simple example where we are reading the data from the 4 files.

```
import java.io.*;
import java.util.*;
class B{
    public static void main(String args[]) throws Exception{
        FileInputStream fin1=new FileInputStream ("f1.txt");
        FileInputStream fin2=new FileInputStream ("f2.txt");
        FileInputStream fin3=new FileInputStream ("f3.txt");
        FileInputStream fin4=new FileInputStream ("f4.txt");
        // creating Vector object to all the stream
        Vector v = new Vector();
        v.add(fin1); v.add(fin2);
        v.add(fin3); v.add(fin4);
        // creating enumeration object
        Enumeration e= v.elements();
        // passing the enumeration object
        SequenceInputStream sis= new SequenceInputStream(e);
        int i;
        while(i=sis.read()!=-1) {
            System.out.print((char)i);
        }
        sis.close();
        fin1.close(); fin2.close();
        fin3.close(); fin4.close();
    }
}
```

Serializable Interface and Serialization :-

Serializability of a class is enabled by the class implementing the **java.io.Serializable interface**. Classes that do not implement this interface will not have any of their state serialized or deserialized. All subtypes of a serializable class are themselves serializable. The serialization interface has no methods or fields and serves only to identify the semantics of being serializable.

ObjectOutputStream class:

An ObjectOutputStream is used to write primitive data types and Java objects to an OutputStream. Only objects that support the java.io.Serializable interface can be written to streams.

Commonly used Constructors:

- **public ObjectOutputStream(OutputStream out) throws IOException {}** : creates an ObjectOutputStream that writes to the specified OutputStream.

Commonly used Methods:

1. **public final void writeObject(Object obj) throws IOException {}**: write the specified object to the ObjectOutputStream.
2. **public void flush() throws IOException {}**: flushes the current output stream.

ObjectInputStream class:

An ObjectInputStream deserializes objects and primitive data written using an ObjectOutputStream.

Commonly used Constructors:

1. **public ObjectInputStream(InputStream in) throws IOException {}**: creates an ObjectInputStream that reads from the specified InputStream.

Commonly used Methods:

1. **public final Object readObject() throws IOException, ClassNotFoundException {}**: reads an object from the input stream.

Example of Serialization

```
import java.io.Serializable;
public class Student implements Serializable{
    int id;
    String name; // transient String name;
    public Student(int id, String name){
        this.id=id;
        this.name=name;
    }
}
```

```
import java.io.*;
class Persist{
    public static void main(String args[]) throws Exception{
        Student s1 =new Student(211,"ravi");
        FileOutputStream fout=new FileOutputStream("f.txt");
        ObjectOutputStream out=new ObjectOutputStream(fout);
        out.writeObject(s1);
        out.flush();
        System.out.println("success");
    }
}
```

Example of Deserialization:

```
import java.io.*;
class DePersist{
    public static void main(String args[]) throws Exception{
        ObjectInputStream in=new ObjectInputStream(new FileInputStream("f.txt"));
        Student s=(Student)in.readObject();
        System.out.println(s.id+" "+s.name);
        in.close();
    }
}
```

The transient keyword

The transient keyword is used in serialization. If you define any data member as transient, it will not be serialized. Let's take an example, I have declared a class as Student, it has three data members id, name and age. If you serialize the object, all the values will be serialized but I don't want to serialize one value, e.g. age then we can declare the age data member as transient.

Applet

Applets: Applets are small applications that are embedded in the webpage to generate the dynamic content, accessed on an Internet server, transported over the Internet to client side, automatically installed, and run inside the browser.

Advantage of Applet

There are many advantages of applet. They are as follows:

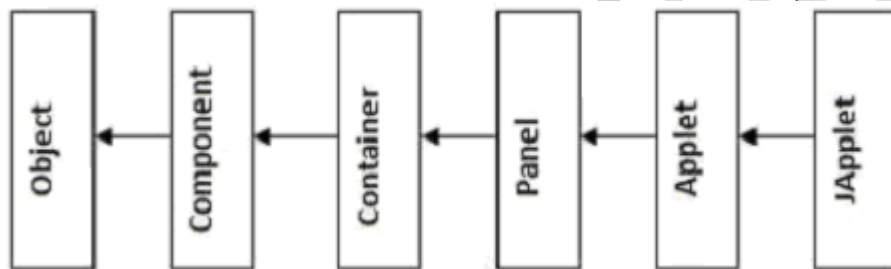
- It works at client side so less response time.
- Secured
- It can be executed by browsers running under many platforms, including Linux, Windows, Mac Os etc.

Drawback of Applet

- Plugin is required at client browser to execute applet.

Hierarchy of Applet

Applet class extends Panel. Panel class extends Container which is the subclass of Component.



```
import java.awt.*;
import java.applet.*;
public class SimpleApplet extends Applet {
    public void paint(Graphics g) {
        g.drawString("A Simple Applet", 20, 20);
    }
}
```

`void drawString(String message, int x, int y)`

Applets interact with the user through the AWT, not through the console-based I/O classes. The AWT contains support for a window-based, graphical interface.

All applets are subclasses of Applet:

This class must be declared as public, because it will be accessed by code that is outside the program. This applet begins with two import statements. The first import is the Abstract Window Toolkit (AWT) classes. Inside SimpleApplet, **paint()** is declared. **This method is defined by the java.awt.Component class and must be overridden by the applet.** paint() is called each time that the applet must redisplay its output. The paint() method has one parameter of type Graphics. This parameter contains 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.

Note: In a Java window, the upper-left corner is location 0, 0.

How to run an Applet?

There are two ways to run an applet

- By Java-compatible web browser using html file.
- By appletViewer tool (for testing purpose).

1. Executing the applet within a Java-compatible Web browser.

To execute an applet in a Web browser, you need to write a short HTML text file that contains the appropriate APPLET tag. Here is the HTML file that executes SimpleApplet:

Hello.html

```
<applet code="SimpleApplet" width=200 height=60></applet>
```

2. Using an Applet Viewer, such as the standard SDK tool, `appletviewer.exe`.

An `appletviewer` executes your applet in a window. This is generally the fastest and easiest way to test your applet. If you use this method, the SimpleApplet source file looks as under:

```
import java.awt.*;
import java.applet.*;
/* <applet code="SimpleApplet" width=200 height=60> </applet> */
public class SimpleApplet extends Applet {
    public void paint(Graphics g) {
        g.drawString("A Simple Applet", 20, 20);
    }
}
```

Applet Architecture:

Four Life cycle methods of Applet are — `init()`, `start()`, `stop()`, and `destroy()`. Another, `paint()`, is defined by the AWT's `Component` class.

- 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 on screen. 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.
- 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.
- 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()`.

Who is responsible to manage the life cycle of an applet? Java Plug-in software.

```
import java.awt.*;
import java.applet.*;
/* <applet code = "AppletSkel" width = 300 height = 100>
</applet> */
public class AppletSkel extends Applet {
    public void init() {
        System.out.println("Inside init()");
    }
    public void start() {
        System.out.println("Inside start()");
    }
    public void stop() {
        System.out.println("Inside stop()");
    }
    public void destroy() {
        System.out.println("Inside destroy()");
    }
    public void paint(Graphics g) {
        System.out.println("Inside paint()");
    }
}
```

Output:

```
Inside init()
Inside start()
Inside paint()
Inside stop()
Inside start()
Inside paint()
Inside stop()
Inside destroy()
```

Methods: setBackground & setForeground

void setBackground(Color newColor) : Changes background color of applet
void setForeground(Color newColor) : Changes foreground color of applet

The class Color defines the constants shown below:

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

Simple Banner:

```
import java.awt.*;
import java.applet.*;
/* <applet code = "SimpleBanner" width =500 height = 400>
   </applet> */
public class SimpleBanner extends Applet implements Runnable
{
    String msg="          A   Simple   Moving   Banner.";
    Thread t=null;
    boolean stopFlag;
    public void init( ) {
        setBackground(Color.yellow);
        setForeground(Color.red);
    }
    public void start( ) {
        t=new Thread(this);
        stopFlag=false;
        t.start( );
    }
    public void stop( ) {
        stopFlag=true;
        t=null;
    }
    public void paint(Graphics g) {
        g.drawString(msg,150,330);
    }

    public void run( ) {
        char ch;
        for(;; ) {
            try {
                repaint( );
                Thread.sleep(50);
                ch=msg.charAt(0);
                msg=msg.substring(1,msg.length());
                msg+=ch;
                if(stopFlag)
                    break;
            }
            catch(InterruptedException e) { }
        } // for
    } // run()
} // SimpleBanner
```

Method: getDocumentBase() & getCodeBase()

URL getCodeBase():- Gets the base URL. e.g. d:/Java/Applet
URL getDocumentBase():- Gets the URL of the document in which this applet is embedded.
e.g. d:/Java/Applet/Filename.java

Example : On getCodeBase() and getDocumentBase()

```
import java.awt.*;
import java.applet.*;
import java.net.*;
// <applet code = "Bases" width = 300 height = 50> </applet>
public class Bases extends Applet {
    public void paint(Graphics g) {
        String msg;
        URL url = getCodeBase();
        msg = "Code Base :"+url.toString();
        g.drawString(msg,10,20);
        System.out.println(""+msg);
        url = getDocumentBase();
        msg = "document base; " + url.toString();
        g.drawString(msg,10,40);
        System.out.println(""+msg);
    }
}
```

AppletContext Interface and showDocument() :

This interface corresponds to an applet's environment: the document containing the applet and the other applets in the same document. The methods in this interface can be used by an applet to obtain information about its environment.

- **AppletContext getAppletContext():** Returns the context of the currently executing applet
- **showDocument(URL url):** Transfer control to another URL,
- **showStatus(String msg) :** Displays msg string in the "status window".

<pre>import java.awt.*; import java.applet.*; import java.net.*; public class ACDemo extends Applet { URL url; public void start() { AppletContext ac= getAppletContext(); url= getCodeBase(); try{ url=new URL(url+"Hello.html"); Thread.sleep(3000); ac.showDocument(url); } catch(Exception e) { showStatus("URL not found"); } } }</pre>	<pre>url=new URL(url+"acdemo.bmp"); url=new URL(url+"notes.txt"); url=new URL(url+"Thinkinginjava.pdf");</pre>
--	--

Example on Status Window:

```
import java.awt.*;
import java.applet.*;
/* <applet code = "StatusWindow" width =500 height = 50> </applet> */

public class StatusWindow extends Applet {
    public void init( ) {
        setBackground(Color.cyan);
    }
    public void paint(Graphics g) {
        g.drawString(" This is an applet Windows .",10,20);
        showStatus(" This is  Status Window .");
    }
}
```

Parameters Example:

```
import java.awt.*;
import java.applet.*;
/*<applet code = "ParamDemo" width =500 height = 200>
    <Param name=fontName value=Courier>
    <Param name=fontSize value=14>
    <Param name=leading value=2>
    <Param name=accountEnabled value=true>
</applet>
*/

public class ParamDemo extends Applet {
    String fontName;
    int fontSize;
    float leading;
    boolean active;
    public void start( ) {
        String param;
        fontName=getParameter("fontName");
        if(fontName==null)
            fontName="Not Found";
        param=getParameter("fontSize");
        try{
            if(param!=null)
                fontSize=Integer.parseInt(param);
            else
                fontSize=0;
        }
        catch(NumberFormatException e) {
            fontSize=-1;
        }
        param=getParameter("leading");
        try{
            if(param!=null)
                leading=Float.valueOf(param).floatValue( );
            else
                leading=0;
        }
        catch(NumberFormatException e) {
            leading=-1;
        }
        param=getParameter("accountEnabled");
        if(param!=null)
            active=Boolean.valueOf(param).booleanValue();
    } // start()

    public void paint(Graphics g) {
        g.drawString("Font name :" +fontName ,0, 10);
        g.drawString("Font size :" + fontSize ,0, 26);
        g.drawString("Leading :" + leading ,0, 42);
        g.drawString("Account Active :" + active ,0, 58);
    }
} // ParamDemo
```

Event Classes

At the root of the Java event class hierarchy is `EventObject`, which is in `java.util` package. It is the super class for all events.

Constructor: `EventObject(Object src)`

Methods: `Object getSource()` and `String toString()`:

`Object getSource()`

The class **AWTEvent**, defined within the `java.awt` package, is a subclass of `EventObject`.

```
java.lang.Object
└─ java.util.EventObject
    └─ java.awt.AWTEvent
```

int getID()

`EventObject` is a superclass of all events.

`AWTEvent` is a superclass of all AWT events that are handled by the delegation event model.

Event Classes

1) ActionEvent :- Generated when a button is pressed, a list item is double-clicked, or a menu item is selected.

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

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

modifiers :- 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`.

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.

`String getActionCommand()` : returns label of object.

`int getModifiers()` : returns modifier

`long getWhen()` : returns the time at which the event occurs

2) AdjustmentEvent :- Generated when a scroll bar is manipulated.

3) ComponentEvent :- Generated when a component is hidden, moved, resized, or becomes visible.

4) ContainerEvent :- Generated when a component is added to or removed from a container.

5) FocusEvent :- Generated when a component gains or loses keyboard focus.

6) InputEvent :- Abstract super class for all component input event classes.

7) ItemEvent :- 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

ItemEvent constructor: `ItemEvent(ItemSelectable src, int type, Object entry, int state)`

Methods: `Object getItem()`

`int getStateChange()` : `SELECTED` or `DESELECTED`

`boolean getState()` : `true` or `false`

8) KeyEvent :- Generated when input is received from the keyboard.

`KEY_PRESSED`, `KEY_RELEASED`, and `KEY_TYPED`.

Methods: `char getKeyChar()`

`int getKeyCode()`

9) MouseEvent :- Generated when the mouse is dragged, moved, clicked, pressed, or released.
Also generated when the mouse enters or exits a component.

Constructor: `public MouseEvent(Component source, int id, long when, int modifiers,
int x, int y, int clickCount, boolean popupTrigger)`

10) MouseWheelEvent :- Generated when the mouse wheel is moved.

11) TextEvent :- Generated when the value of a text area or text field is changed.

12) WindowEvent :- Generated when a window is activated, closed, deactivated, deiconified, iconified, opened, or quit

Listener interfaces :-

1) ActionListener a) `void actionPerformed(ActionEvent e)`

2) AdjustmentListener a) `void adjustmentValueChanged(AdjustmentEvent e)`

3) ComponentListener

- a) `void componentHidden(ComponentEvent e)`
 Invoked when the component has been made invisible.
- b) `void componentMoved(ComponentEvent e)`
 Invoked when the component's position changes.
- c) `void componentResized(ComponentEvent e)`
 Invoked when the component's size changes.
- d) `void componentShown(ComponentEvent e)`
 Invoked when the component has been made visible.

4) ContainerListener

- a) `void componentAdded(ContainerEvent e)`
 Invoked when a component has been added to the container.
- b) `void componentRemoved(ContainerEvent e)`
 Invoked when a component has been removed from the container.

5) FocusListener

- a) `void focusGained(FocusEvent e)`
 Invoked when a component gains the keyboard focus.
- b) `void focusLost(FocusEvent e)`
 Invoked when a component loses the keyboard focus.

6) ItemListener :

- a) `void itemStateChanged(ItemEvent e)`
 Invoked when an item has been selected or deselected by the user.

7) KeyListener

- a) `void keyPressed(KeyEvent e)` : Invoked when a key has been pressed.
- b) `void keyReleased(KeyEvent e)` : Invoked when a key has been released.
- c) `void keyTyped(KeyEvent e)` : Invoked when a key has been typed.

8) MouseListener

- a) `void mouseClicked(MouseEvent e)`
 Invoked when the mouse button has been clicked (pressed and released) on a component.
- b) `void mouseEntered(MouseEvent e)`
 Invoked when the mouse enters a component.
- c) `void mouseExited(MouseEvent e)`
 Invoked when the mouse exits a component.
- d) `void mousePressed(MouseEvent e)`
 Invoked when a mouse button has been pressed on a component.
- e) `void mouseReleased(MouseEvent e)`
 Invoked when a mouse button has been released on a component.

9) **MouseMotionListener**

- a) void mouseDragged(MouseEvent e) :
 Invoked when a mouse button is pressed on a component and then dragged.
- b) void mouseMoved(MouseEvent e) :
 Invoked when the mouse cursor has been moved onto a component but no buttons have been pushed.

10) **MouseWheelListener**

- void mouseWheelMoved(MouseWheelEvent e):
 Invoked when the mouse wheel is rotated.

11) **TextListener**

- void textValueChanged(TextEvent e)
 Invoked when the value of the text has changed.

12) **WindowListener**

- a) void windowActivated(WindowEvent e)
 Invoked when the Window is set to be the active Window.
- b) void windowClosed(WindowEvent e)
 Invoked when a window has been closed as the result of calling dispose on the window.
- c) void windowClosing(WindowEvent e)
 Invoked when the user attempts to close the window from the window's system menu.
- d) void windowDeactivated(WindowEvent e)
 Invoked when a Window is no longer the active Window.
- e) void windowDeiconified(WindowEvent e)
 Invoked when a window is changed from a minimized to a normal state.
- f) void windowIconified(WindowEvent e)
 Invoked when a window is changed from a normal to a minimized state.
- g) void windowOpened(WindowEvent e)
 Invoked the first time a window is made visible.

Using the Delegation Event Model

Applet programming using the delegation event model is actually quite easy. Just follow these two steps:

1. Implement the appropriate interface in the listener so that it will receive the type of event desired.
2. Implement code to register and unregister (if necessary) the listener as a recipient for the event notifications.

Handling Mouse Events

To handle mouse events, you must implement the **MouseListener** and the **MouseMotionListener** interfaces. The following applet demonstrates the process.

int getX(): Returns the horizontal x position of the event relative to the source component.

int getY(): Returns the vertical y position of the event relative to the source component.

int getClickCount(): Returns the number of mouse clicks associated with this event.

```
import java.awt.*;
import java.awt.event.*;
import java.applet.*;
/*      <applet code = "MouseEvents" width = 300 height = 100>
      </applet>
*/
```

<pre>public class MouseEvents extends Applet implements MouseListener, MouseMotionListener { String msg = " "; int mouseX = 0, mouseY = 0, i=0; public void init() { addMouseListener(this); addMouseMotionListener(this); } public void mouseClicked(MouseEvent me) { i=me.getID(); mouseX = 10; mouseY = 30; msg = "Mouse Clicked.="+i+":-> "+me.getClickCount()+" c="+c; repaint(); } public void mouseEntered(MouseEvent me) { i=me.getID(); mouseX = 0; mouseY = 10; msg = "Mouse Entered.="+i; repaint(); } }</pre>	<pre>public void mouseExited(MouseEvent me) { i=me.getID(); mouseX = 0; mouseY = 10; msg = "Mouse Exited.="+i; repaint(); } public void mousePressed(MouseEvent me) { i=me.getID(); mouseX = me.getX(); mouseY = me.getY(); msg = "Down="+i; repaint(); } public void mouseReleased(MouseEvent me) { i=me.getID(); mouseX = me.getX(); mouseY = me.getY(); msg = "Up="+i; repaint(); }</pre>
--	--

```

public void mouseDragged(MouseEvent me)
{
    i=me.getID();
    mouseX = me.getX();
    mouseY = me.getY();
    msg ="*="+i;
    showStatus("Dragging mouse at " +
               mouseX + ", " + mouseY);
    repaint();
}

```

```

public void mouseMoved(MouseEvent me)
{
    i=me.getID();
    showStatus("Moving mouse at i="+i+" "
               +me.getX() + ", " + me.getY());
}
public void paint(Graphics g )
{
    g.drawString(msg, mouseX, mouseY);
}
} // class MouseEvents

```

```

void addMouseListener(MouseListener ml);
void addMouseMotionListener(MouseMotionListener mml);

```

Handling Keyboard Events : You will be implementing the **KeyListener** interface here.

// Demonstrate the key event handlers.

```

import java.awt.*;
import java.awt.event.*;
import java.applet.*;
/*<applet code="SimpleKey" width=300
   height=100>
</applet> */

public class SimpleKey extends Applet
    implements KeyListener
{
    String msg = "";
    int X = 10, Y = 20; // output coordinates
    public void init() {
        addKeyListener(this);
        requestFocus(); // request input focus
    }
}

```

```

public void keyPressed(KeyEvent ke) {
    showStatus("Key Down");
}
public void keyReleased(KeyEvent ke) {
    showStatus("Key Up");
}
public void keyTyped(KeyEvent ke) {
    msg += ke.getKeyChar();
    repaint();
}

// Display keystrokes.
public void paint(Graphics g)
{
    g.drawString(msg, X, Y);
}
} // class SimpleKey

```

// Demonstrate some virtual key codes.

```

import java.awt.*;
import java.awt.event.*;
import java.applet.*;
/*
<applet code="KeyEvents" width=300
height=100>
</applet>
*/
public class KeyEvents extends Applet
    implements KeyListener
{
    String msg = "";
    int X = 10, Y = 20; // output coordinates
    public void init() {
        addKeyListener(this);
        requestFocus(); // request input focus
    }
}

```

```

public void keyReleased(KeyEvent ke)
{
    showStatus("Key Up");
}
public void keyTyped(KeyEvent ke)
{
    msg += ke.getKeyChar();
    repaint();
}

// Display keystrokes.
public void paint(Graphics g)
{
    g.drawString(msg, X, Y);
}

```

<pre> public void keyPressed(KeyEvent ke) { showStatus("Key Down"); int key = ke.getKeyCode(); switch(key) { case KeyEvent.VK_F1: msg += "<F1>"; break; case KeyEvent.VK_F2: msg += "<F2>"; break; case KeyEvent.VK_F3: msg += "<F3>"; break; } } </pre>	<pre> case KeyEvent.VK_PAGE_DOWN: msg += "<PgDn>"; break; case KeyEvent.VK_PAGE_UP: msg += "<PgUp>"; break; case KeyEvent.VK_LEFT: msg += "<Left Arrow>"; break; case KeyEvent.VK_RIGHT: msg += "<Right Arrow>"; break; } repaint(); } } // class KeyEvents </pre>
--	--

Adapter Classes :->

An adapter class provides an empty implementation of all methods in an event listener interface. Adapter classes are useful when you want to receive and process only some of the events that are handled by a particular event listener interface.

```

class MouseAdapter implements MouseListener
{
    void mouseClicked(MouseEvent e) { }
    void mouseEntered(MouseEvent e) { }
    void mouseExited(MouseEvent e) { }
    void mousePressed(MouseEvent e) { }
    void mouseReleased(MouseEvent e) { }
}

```

An adapter class provides an empty implementation of all methods in an event listener interface. Adapter classes are useful when you want to receive and process only some of the events that are handled by a particular event listener interface. You can define a new class to act as an event listener by extending one of the adapter classes and implementing only those events in which you are interested.

For example, the **MouseMotionAdapter** class has two methods, `mouseDragged()` and `mouseMoved()`. The signatures of these empty methods are exactly as defined in the `MouseMotionListener` interface. If you were interested in only mouse drag events, then you could simply extend `MouseMotionAdapter` and implement `mouseDragged()`. The empty implementation of `mouseMoved()` would handle the mouse motion events for you.

Adapter Class	Listener Interface
ComponentAdapter	ComponentListener
ContainerAdapter	ContainerListener
FocusAdapter	FocusListener
KeyAdapter	KeyListener
MouseAdapter	MouseListener
MouseMotionAdapter	MouseMotionListener
WindowAdapter	WindowListener

Inner Classes :->

Without Inner class	Using Inner class
<pre>import java.awt.*; import java.awt.event.*; import java.applet.*; /* <applet code = "AdapterDemo" width = 300 height = 100> </applet> */ public class AdapterDemo extends Applet { public void init(){ addMouseListener(new MyMouseAdapter(this)); } } class MyMouseAdapter extends MouseAdapter { AdapterDemo ad; public MyMouseAdapter(AdapterDemo ad) { this.ad = ad; } public void mouseClicked(MouseEvent me) { ad.showStatus("Mouse clicked"); } }</pre>	<pre>import java.awt.*; import java.awt.event.*; import java.applet.*; /* <applet code = "AdapterDemo1" width = 300 height = 100> </applet> */ public class AdapterDemo1 extends Applet { public void init() { addMouseListener(new MyMouseAdapter()); } } class MyMouseAdapter extends MouseAdapter { public void mouseClicked(MouseEvent me) { showStatus("Mouse clicked"); } } } // AdapterDemo1</pre>

Anonymous Inner Classes :->

An anonymous inner class is one that is not assigned a name.

The syntax `new MouseAdapter() { ... }` indicates to the compiler that the code between the braces defines an anonymous inner class.

//Above example using Anonymous Inner Class

<pre>import java.awt.*; import java.awt.event.*; import java.applet.*; /* <applet code = "AnonymousInnerDemo" width = 300 height = 100> </applet> */ public class AnonymousInnerDemo extends Applet {</pre>	<pre> public void init() { addMouseListener(new MouseAdapter(){ public void mouseClicked(MouseEvent me) { showStatus("Mouse clicked"); } }); } // init } // class AnonymousInnerDemo</pre>
---	---

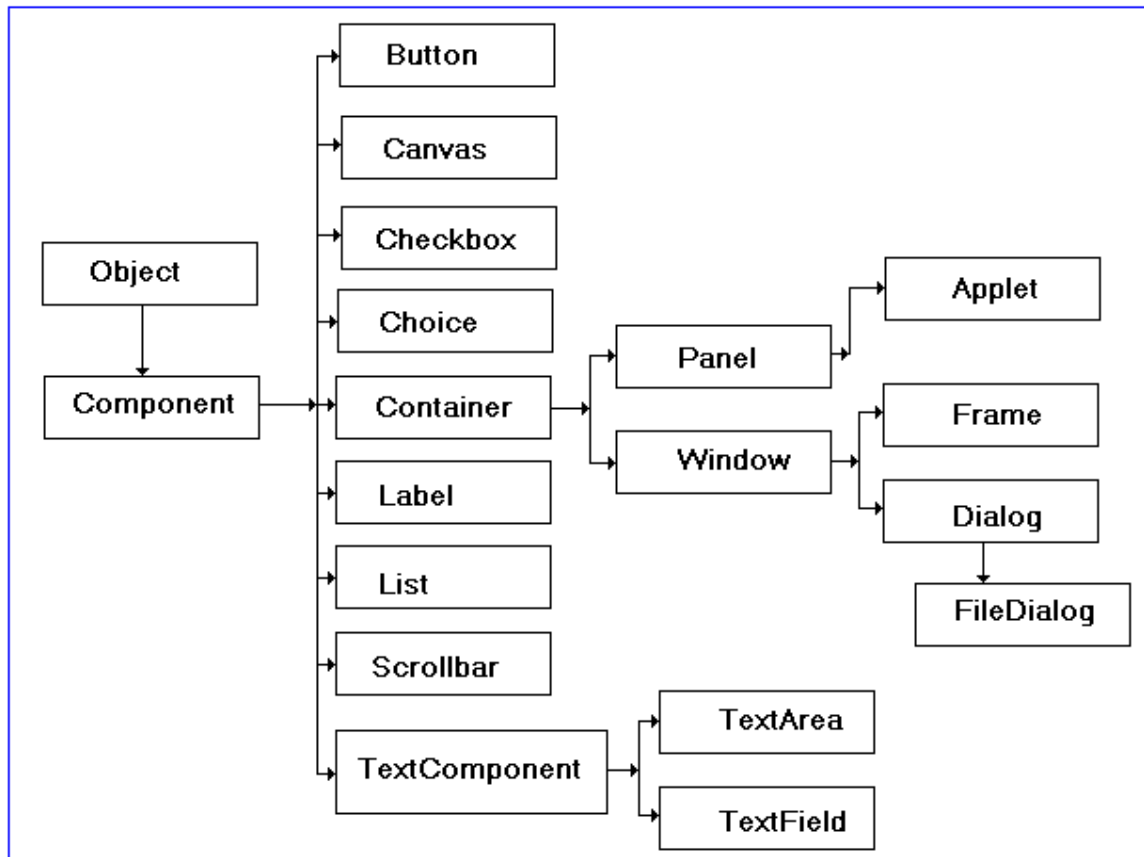
In the above example Anonymous Inner class extends MouseAdapter. This new class is not named, but it is automatically instantiated when this expression is executed.

Because this anonymous inner class is defined within the scope of AnonymousInner1, it has access to all of the variables and methods within the scope of that class. Therefore, it can call the `showStatus()` method directly.

AWT CLASSES:

GUI Control Components:

GUI control components are the primary elements of a graphical user interface that enable interaction with user. They are all subclasses of the component class.



The AWT defines windows according to a class hierarchy that adds functionality and specificity with each level. The two most common windows are those derived from Panel, which is used by applets, and those derived from Frame, which creates a standard window.

Component :->

At the top of the AWT hierarchy is the Component class. Component is an abstract class that encapsulates all of the attributes of a visual component. All user interface elements that are displayed on the screen and that interact with the user are subclasses of Component. It defines over a hundred public methods that are responsible for managing events, such as mouse and keyboard input, positioning and sizing the window, and repainting.

Container :->

The Container class is a subclass of Component. A container is responsible for laying out (that is, positioning) any components that it contains. It has additional methods that allow other Component objects to be nested within it. Other Container objects can be stored inside of a Container (since they are themselves instances of Component). This makes for a multileveled containment system.

Panel :->

The Panel class is a concrete subclass of Container. Panel is the superclass for Applet. When screen output is directed to an applet, it is drawn on the surface of a Panel object. A Panel is a window that does not contain a title bar, menu bar, or border and resizing corners.

Window :-> The Window class creates a top-level window.

Frame :-> Frame encapsulates what is commonly thought of as a "window." It is a subclass of Window and has a title bar, menu bar, borders, and resizing corners.

Frame()
Frame(String title)

void setSize() :->

The setSize() method is used to set the dimensions of the window.

void setSize(int newWidth, int newHeight)

void setSize(Dimension newSize)

Dimension getSize()

Hiding and Showing a Window : void setVisible(boolean visibleFlag)

Setting a Window's Title : void setTitle(String newTitle)

Closing a Frame Window:

by calling setVisible(false). To intercept a window-close

event, you must implement the windowClosing() method of the WindowListener interface. Inside windowClosing(), you must remove the window from the screen.

Creating a Frame Window in an Applet :

<pre>import java.awt.*; import java.applet.*; import java.awt.event.*; /* <applet code = "AppletFrame" width = 300 height = 50> </applet> */ class SampleFrame extends Frame { SampleFrame(String title) { super(title); MyWindowAdapter adapter = new MyWindowAdapter(this); addWindowListener(adapter); } public void paint(Graphics g) { g.drawString("This is in frame window",10,20); } }</pre>	<pre>class MyWindowAdapter extends WindowAdapter { SampleFrame sampleFrame; public MyWindowAdapter(SampleFrame sampleFrame) { this.sampleFrame = sampleFrame; } public void windowClosing(WindowEvent we) { sampleFrame.setVisible(false); } }</pre>
---	--

<pre>public class AppletFrame extends Applet { Frame f; public void init() { f = new SampleFrame("A Frame Window") f.setSize(250,250); f.setVisible(true); } public void start() { f.setVisible(true); } }</pre>	<pre>public void stop() { f.setVisible(false); } public void paint(Graphics g) { g.drawString("This is in applet window", 10,20); } } // AppletFrame</pre>
--	---

<pre> import java.awt.*; import java.applet.*; import java.awt.event.*; /* <applet code = "AppletFrame1" width = 300 height = 50> </applet> */ class SampleFrame extends Frame { SampleFrame(String title) { super(title); addWindowListener(new WindowAdapter() { public void windowClosing(WindowEvent we) { setVisible(false); } }); } // constructor public void paint(Graphics g) { g.drawString("This is in frame window", 10,40); } } // class SampleFrame </pre>	<pre> public class AppletFrame1 extends Applet { Frame f; public void init() { f = new SampleFrame("A Frame Window"); f.setSize(250,250); f.setVisible(true); } public void start() { f.setVisible(true); } public void stop() { f.setVisible(false); } public void paint(Graphics g) { g.drawString("This is in applet window", 10,20); } } // class AppletFrame1 </pre>
---	--

Public Static Void Main with Frame :->

```

import java.awt.*;
import java.applet.*;
import java.awt.event.*;

```

```

class SampleFrame extends Frame {
    SampleFrame(String title) {
        super(title);
        addWindowListener(new WindowAdapter() {
            public void windowClosing(WindowEvent we) {
                setVisible(false);
            }
        });
    }
    public void paint(Graphics g) {
        g.drawString("This is in frame window",10,40);
    }
}

public class AppletFrame1 {
    public static void main(String arg[]) {
        Frame f;
        f = new SampleFrame("A Frame Window");
        f.setSize(250,250);
        f.setVisible(true);
    }
}

```

Working with Graphics

Drawing Lines: Lines are drawn by means of the **drawLine()** method, shown here:
void drawLine(int startX, int startY, int endX, int endY)

```

import java.awt.*;
import java.applet.*;
import java.awt.event.*;
/* <applet code = "Lines" width = 300 height = 300> </applet> */
public class Lines extends Applet {
    public void init(){
        setBackground(Color.black);
        setForeground(Color.red);
    }
    public void paint(Graphics g)
    {
        g.drawLine(33, 33, 350, 350);
    }
}

```

Drawing Rectangles: The **drawRect()** and **fillRect()** methods display an outlined and filled rectangle, respectively. They are shown here:

void drawRect(int top, int left, int width, int height)
void fillRect(int top, int left, int width, int height)

```

import java.awt.*;
import java.applet.*;
import java.awt.event.*;

```

```

/* <applet code = "Rectangle" width = 400 height = 400> </applet> */
public class Rectangle extends Applet {
    public void init() {
        //setBackground(Color.black);
        setForeground(Color.red);
    }
    public void paint(Graphics g) {
        g.drawRect(10,10,50,50);
        g.fill3DRect(70,10,50,50,true);
        g.fillRect(10,75,50,50);
        g.fill3DRect(70,75,50,50,false);
    }
}

```

Drawing Ellipses and Circles: To draw an ellipse, use **drawOval()**. To fill an ellipse, use **fillOval()**. These methods are shown here:

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

```

import java.awt.*;
import java.applet.*;
import java.awt.event.*;
/* <applet code = "Ellipses" width = 400 height = 400> </applet> */
public class Ellipses extends Applet {
    public void init() {
        setBackground(Color.blue);
        setForeground(Color.red);
    }
    public void paint(Graphics g) {
        g.drawOval(10,10,50,50);
        g.fillOval(100,10,50,50);
    }
}

```

Working with Color

Color(int red, int green, int blue)

Color(int rgbValue)

Color(float red, float green, float blue)

```
import java.awt.*;
import java.applet.*;
import java.awt.event.*;
/* <applet code = "ColorDemo" width = 400 height = 400> </applet> */

public class ColorDemo extends Applet {
    public void paint(Graphics g)
    {
        Color c1=new Color(255,100,100);
        Color c3=new Color(100,100,100);
        g.setColor(c1);
        g.fillOval(10,10,50,50);

        g.setColor(c3);
        g.fillOval(100,10,50,50);
    }
}
```

Working with Fonts :->

Font(String name, int style, int size):

Creates a new Font from the specified name, style and point size.

```
import java.awt.*;
import java.applet.*;
import java.awt.event.*;
/* <applet code = "SampleFonts" width = 400 height = 400> </applet> */
public class SampleFonts extends Applet {
    Font f;
    public void init()
    {
        setForeground(Color.red);
        f=new Font("Arial", Font.ITALIC ,72);
        setFont(f);
    }
    public void paint(Graphics g)
    {
        g.drawString("Hello",133,133);
    }
}
```

AWT Controls :->

Controls are components that allow a user to interact with your application in various ways—for example, a commonly used control is the push button. A *layout manager* automatically positions components within a container.

Label, push buttons, check boxes, choice List, List, scrollbar, Text editing
Adding controls

Adding and Removing Controls

Component add(Component compObj)

Label :- A label is an object of type Label

Constructors	Fields	Methods
Label() Label(String str) Label(String str, int alignment)	static int CENTER static int LEFT static int RIGHT	void setText(String text) String getText()

```
import java.awt.*;
import java.applet.*;
/*<applet code = "LableDemo" width = 300 height = 200></applet>*/
public class LableDemo extends Applet
{
    public void init() {
        Label user=new Label("User Name", Label.CENTER);
        Label pass=new Label("Password", Label.CENTER);
        add(user);
        add(pass);
    }
}
```

Button :- Button class creates a labeled button.

Constructor: **Button()**: Creates a button with an empty string.

Button(String label): Constructs a button with specified label.

Methods: **String getLabel()**:Gets the label of this button.

void setLabel(String label): Sets the button's label to be the specified string.

Handling Buttons : Event → ActionEvent

Listener → ActionListener → public void actionPerformed(ActionEvent e)

```
import java.awt.*;
import java.applet.*;
import java.awt.event.*;
/*<applet code = "ButtonDemo1" width = 300 height = 200></applet>*/
public class ButtonDemo1 extends Applet implements ActionListener {
    String msg = " ";
    Button add, sub;
    public void init() {
        add =new Button("ADD");
        sub =new Button("SUB");
        add(add); add(sub);
        add.addActionListener(this);
        sub.addActionListener(this);
    }
    public void actionPerformed(ActionEvent ae) {
        Object ob= ae.getSource();
        if(ob==add) msg="You pressed ADD";
        if(ob==sub) msg="You pressed SUB";
        repaint();
    }
    public void paint(Graphics g) {
        g.drawString(msg,6,100) ;
    }
}
```

Check Boxes :->

1) Checkbox() : Creates a check box with an empty string for its label.

2) Checkbox(String label) : Creates a check box with the specified label.

3) Checkbox(String label, boolean state) : Checkbox with the specified label and state.

4) Checkbox(String label, CheckboxGroup group, boolean state): Checkbox in specified checkbox group

Method:

```
boolean getState()
void setState(boolean state)
```

Handling Checkbox : Event → ItemEvent

Listener → ItemListener

Method → public void itemStateChanged(ItemEvent ie)

CheckboxGroup: The CheckboxGroup class is used to group together a set of Checkbox buttons

Method: Checkbox getSelectedCheckbox()

void setSelectedCheckbox(Checkbox box)

```
import java.awt.*;
import java.applet.*;
import java.awt.event.*;
/* <applet code = "CheckboxDemo" width = 300 height = 200></applet> */
public class CheckboxDemo extends Applet implements ItemListener {
    String msg = " ";
    Checkbox cric, hoc;
    String strCric, strHoc;
    public void init() {
        cric = new Checkbox("Cricket");
        hoc = new Checkbox("Hockey");
        add(cric);
        add(hoc);
        cric.addItemListener(this);
        hoc.addItemListener(this);
    }
    public void itemStateChanged(ItemEvent ie) {
        if(cric.getState()) strCric="Cricket"; else strCric=null;
        if(hoc.getState()) strHoc="Hockey"; else strHoc=null;
        repaint();
    }
    public void paint(Graphics g) {
        msg="Current State";
        g.drawString(msg,6,80) ;
        msg="Cricket " + cric.getState( );
        g.drawString(msg,6,100) ;
        msg="Hockey " + hoc.getState( );
        g.drawString(msg,6,120) ;
        System.out.println("Cricket="+strCric);
        System.out.println("Hockey="+strHoc);
    }
}
```

Choice Control

Method:

```
void addItem(String name):
void add(String name):
String getSelectedItem():
int getSelectedIndex():
int getItemCount():
void select(int index):
void select(String name):
String getItem(int index) :
int getItemCount() :
```

Event: ItemEvent

```

import java.awt.*;
import java.applet.*;
import java.awt.event.*;
/*<applet code = "ChoiceDemo1" width = 300 height = 200> </applet>*/

public class ChoiceDemo1 extends Applet implements ItemListener {
    Choice branch;
    String msg = " ";
    public void init() {
        branch=new Choice( );
        branch.add("Electronics");
        branch.add("Computer");
        branch.add("Mech.");
        branch.add("Civil");
        branch.select("Civil");
        add(branch);
        branch.addItemListener(this);
    }
    public void itemStateChanged(ItemEvent ie) {
        repaint();
    }
    public void paint(Graphics g) {
        msg="Current Branch  : ";
        msg+=branch.getSelectedItem();
        g.drawString(msg,6,120) ;
    }
}

```

List :->

List() : Creates a new scrolling list.

List(int rows) : Creates a new scrolling list initialized with the specified number of visible lines.

List(int rows, boolean multipleMode)

Creates a new scrolling list initialized to display the specified number of rows. if multipleMode is true then user can select more than one item.

Methods:

- void add(String name)
- void add(String name, int index)
- String getSelectedItem()
- int getSelectedIndex()
- String[] getSelectedItems()
- int[] getSelectedIndexes()
- int getItemCount()
- String getItem(int index)

```

import java.awt.*;
import java.applet.*;
import java.awt.event.*;

/* <applet code = "ListDemo" width = 300 height = 200></applet> */

public class ListDemo extends Applet implements ActionListener
{
    List os, browser;
    String msg = " ";
}

```

```

public void init() {
    os=new List(4,true);
    browser=new List(4,false);
    os.add("Win 98/XP");
    os.add("Win NT/2000");
    os.add("Solaris");
    os.add("MacOS");
    browser.add("Netscape 3.x");
    browser.add("Netscape 4.x");
    browser.add("Netscape 5.x");
    browser.add("Netscape 6.x");
    browser.add("Internet Explorer 4.0");
    browser.add("Internet Explorer 5.0");
    browser.add("Internet Explorer 6.0");
    browser.add("Lynx 2.4");
    browser.select(1);
    add(os);
    add(browser);
    os.addActionListener(this);
    browser.addActionListener(this);
}

public void paint(Graphics g) {
    int idx[ ];
    msg="Current OS";
    idx=os.getSelectedIndexes( );
    for(int i=0;i<idx.length;i++)
        msg+=os.getItem(idx[i] )+ " ";
    g.drawString(msg,6,120) ;
    msg="Current Browser";

    msg+=browser.getSelectedItem( );
    g.drawString(msg,6,140) ;
}

public void actionPerformed(ActionEvent ae) {
    repaint();
}
}

```

TextField :-> The **TextField** class implements a single-line text-entry area, usually called an *edit control*.

- 1) TextField() : Constructs a new text field.
- 2) TextField(int columns) : Constructs a new empty text field with the specified number of columns.
- 3) TextField(String text) : Constructs a new text field initialized with the specified text.
- 4) TextField(String text, int columns)
Constructs a new text field initialized with the specified text to be displayed, and wide enough to hold the specified number of columns.

Methods:

- String getText()
- void setText(String str)
- void select(int startIndex, int endIndex)
- String getSelectedText()
- void setEditable(boolean canEdit)
- void setEchoChar(char ch)

```

import java.awt.*;
import java.awt.event.*;
import java.applet.*;

/* <applet code="TextFieldDemo" width=380 height=150> </applet> */

public class TextFieldDemo extends Applet implements ActionListener {
    TextField name, pass;
    public void init( ) {
        Label namep=new Label("Name : ",Label.RIGHT);
        Label passp=new Label("Password : ",Label.RIGHT);
        name=new TextField(3);
        pass=new TextField(8);
        pass.setEchoChar('*');
        add(namep);
        add(name);
        add(passp);
        add(pass);
        name.setText("InetSoft");
        name.select(4,7);
        name.setEditable(false);
        name.addActionListener(this);
        pass.addActionListener(this);
    }

    public void actionPerformed(ActionEvent ae) {
        repaint( );
    }

    public void paint(Graphics g) {
        g.drawString("Name : " + name.getText( ),6,60);
        g.drawString("Selected text in Name : " + name.getSelectedText( ),6,80);
        g.drawString("Password : " + pass.getText( ),6,100);
    }
}

```

TextArea:
 TextArea()
 TextArea(int rows, int columns)
 TextArea(String text)

```

import java.awt.*;
import java.awt.event.*;
import java.applet.*;
/* <applet code="TextAreaDemo" width=380 height=150></applet>*/
public class TextAreaDemo extends Applet {
    public void init( )
    {
        String val="Hello " ;
        TextArea text=new TextArea(val,5,30);
        add(text);
    }
}

```

Layout Managers :->

All of the components that we have shown so far have been positioned by the default layout manager. As we mentioned at the beginning of this chapter, a layout manager automatically arranges your controls within a window by using some type of algorithm. Each **Container** object has a layout manager associated with it. A layout manager is an instance of any class that implements the **LayoutManager** interface. The layout manager is set by the **setLayout()** method. If no call to **setLayout()** is made, then the default layout manager is used.

void setLayout(LayoutManger layoutObj)
layoutObj :->

Position components manually using setBounds() method

setLayout(null);
void setBounds(int x, int y, int width, int height)

FlowLayout Manager :->

FlowLayout is the default layout manager. This is the layout manager that the preceding examples have used. **FlowLayout** implements a simple layout style, which is similar to how words flow in a text editor. Components are laid out from the upper-left corner, left to right and top to bottom.

FlowLayout :->

FlowLayout()

FlowLayout(int align)

FlowLayout(int align, int hgap, int vgap)

align :-> **FlowLayout.LEFT**
 FlowLayout.CENTER
 FlowLayout.RIGHT

```
public class FlowLayoutDemo extends Applet
implements ItemListener {
    String msg = "";
    Checkbox Win98, winNT, solaris, mac;
    public void init() {
        // set left-aligned flow layout
        setLayout(new FlowLayout(FlowLayout.LEFT));
        :
    }
    :
}
```

BorderLayout :->

A border layout lays out a container, arranging and resizing its components to fit in five regions: north, south, east, west, and center. Each region may contain no more than one component, and is identified by a corresponding constant: NORTH, SOUTH, EAST, WEST, and CENTER. When adding a component to a container with a border layout, use one of these five constants, for example:

BorderLayout()

BorderLayout(int hgap, int vgap)

void addComponent(Component compObj, Object region)

region:->

BorderLayout.CENTER

BorderLayout.SOUTH

BorderLayout.EAST

BorderLayout.WEST

BorderLayout.NORTH

```

import java.awt.*;
import java.util.*;
import java.applet.*;
/* <applet code="BorderLayoutDemo" width=380 height=150> </applet> */

public class BorderLayoutDemo extends Applet {
    public void init( ) {
        setLayout(new BorderLayout() );
        add(new Button("This is across the top ."),BorderLayout.NORTH);
        add(new Label("The footer message might go here ."),BorderLayout.SOUTH);
        add(new Button("Right ."),BorderLayout.EAST);
        add(new Button("Left ."),BorderLayout.WEST);
        String val="HELLO " ;
        add(new TextArea(val),BorderLayout.CENTER);
    }
}

```

GridLayout :->

GridLayout lays out components in a two-dimensional grid. When you instantiate a GridLayout, you define the number of rows and columns.

```

GridLayout()
GridLayout(int rows, int cols)
GridLayout(int rows, int cols, int hgap, int vgap)

```

4 X 4

```

import java.awt.*;
import java.applet.*;
/* <applet code="GridLayoutDemo" width=380 height=150> </applet> */

public class GridLayoutDemo extends Applet
{
    static final int n=4;
    public void init( ){
        setLayout(new GridLayout(n,n));
        setFont(new Font("SansSerif",Font.BOLD,24));
        for(int i=0;i<n;i++) {
            for(int j=0;j<n;j++) {
                int k=i*n+j;
                if(k>0)
                    add(new Button( " " + k));
            }
        }
    }
}

```

CardLayout :->

```

CardLayout()
CardLayout(int horz, int vert)

```

```
void add(Component panelobj, Object name)
```

Methods to activate a card :->

- 1) void first(Container deck)
- 2) void last(Container deck)
- 3) void next(Container deck)
- 4) void previous(Container deck)
- 5) void show(Container deck, String cardName)


```

import java.awt.*;
import java.awt.event.*;
import java.applet.*;

/* <applet code="CardLayoutDemo" width=380 height=150> </applet> */
public class CardLayoutDemo extends Applet implements ActionListener, MouseListener {
    Checkbox Win98,WinNT,solaris,mac;
    Panel osCards;
    CardLayout cardLO;
    Button Win,Other;
    public void init( ) {
        Win=new Button("Windows");
        Other=new Button("Other");
        add(Win);
        add(Other);
        cardLO=new CardLayout( );
        osCards=new Panel( );
        osCards.setLayout(cardLO);

        Win98=new Checkbox("Windows 98/XP ", null, true);
        WinNT=new Checkbox("Windows NT/2000 ");
        solaris=new Checkbox("Solaris");
        mac=new Checkbox("MacOS");

        Panel winPan=new Panel();
        winPan.add(Win98);
        winPan.add(WinNT);

        Panel otherPan=new Panel();
        otherPan.add(solaris);
        otherPan.add(mac);

        osCards.add(winPan, "Windows");
        osCards.add(otherPan, "Other");
        add(osCards);

        Win.addActionListener(this);
        Other.addActionListener(this);
        addMouseListener(this);
    }

    public void mousePressed(MouseEvent me) {
        cardLO.next(osCards);
    }

    public void mouseClicked(MouseEvent me) { }
    public void mouseEntered(MouseEvent me) { }
    public void mouseExited(MouseEvent me) { }
    public void mouseReleased(MouseEvent me) { }
    public void actionPerformed(ActionEvent ae)
    {
        if(ae.getSource( )==Win)
            cardLO.show(osCards, "Windows");
        else
            cardLO.show(osCards, "Other");
    }
}

```

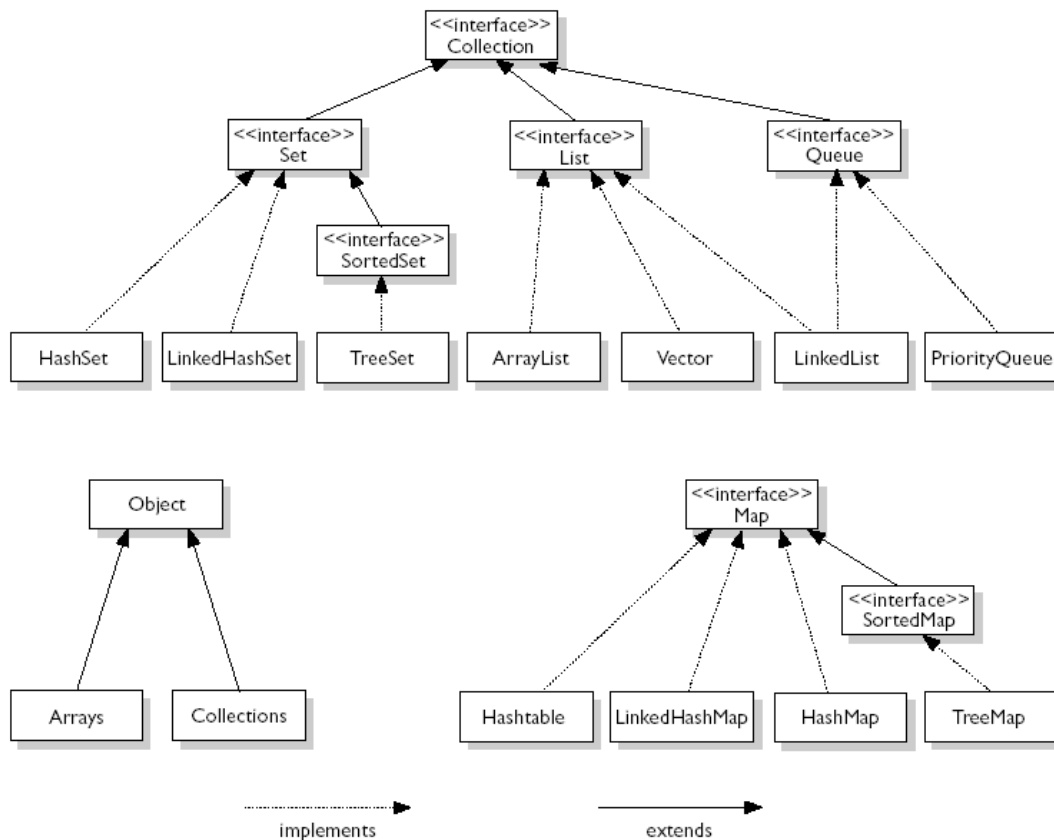
Notes
Vijay Sir: 9371640066

UTIL

Advantages of a collections :

- Reduces programming effort by providing useful data structures and algorithms so you don't have to write them yourself.
- Increases performance
- Reduces the effort required to learn APIs .

*The interface and class hierarchy for collections



Collections's method :->

- 1) boolean add(Object obj) :->
- 2) boolean addAll(Collection c);->
- 3) void clear()
- 4) boolean contains(Object obj)
Returns true if this collection contains the specified obj.
- 5) boolean containsAll(Collection c)
Returns true if this collection contains all of the elements in the specified collection.
- 6) boolean equals(Object obj)
- 7) int hashCode()
- 8) boolean isEmpty()

- Returns true if this collection contains no elements.
- 9) Iterator iterator()
 - Returns an iterator over the elements in this collection.
- 10) boolean remove(Object obj)
- 11) boolean removeAll(Collection c)
- 12) boolean retainAll(Collection c)
- 13) int size()
 - Returns the number of elements in this collection.
- 14) Object[] toArray()
 - Returns an array containing all of the elements in this collection.

=====

Ordered :->

When a collection is ordered, it means you can iterate through the collection in a specific (not-random) order.

e.g. ArrayList

ArrayList, keeps the order established by the elements' index position (just like an array).

Hashtable itself has internal logic to determine the order based on hashcodes.

sorted :->

A sorted collection means that the order in the collection is determined according to some rule or rules, known as the sort order.

=====

List :->

The List interface extends Collection and declares the behavior of a collection that stores a sequence of elements. Elements can be inserted or accessed by their position in the list, using a zero-based index. A list may contain duplicate elements.

Method's defined by List interface.

- =====
- 1) void add(int index, Object obj)
 - 2) boolean addAll(int index, Collection c)
 - 3) Object get(int index)
 - 4) int indexOf(Object obj)
 - 5) int lastIndex(Object obj)
 - 6) ListIterator listIterator(int index)
 - 7) Object remove(int index)
 - 8) Object set(int index, Object obj)
 - 9) List subList(int start, int end) :->
- =====

Implementing classes of List interface. :->

1) ArrayList :->

The ArrayList class extends AbstractList and implements the List interface. ArrayList supports dynamic arrays that can grow as needed.

Constructors:

- 1) ArrayList()
- 2) ArrayList(Collection c)
- 3) ArrayList(int initialCapacity)

Example on ArrayList:

```
import java.util.*;
class ArrayListDemo
{
    public static void main(String args[])
    {
        ArrayList al = new ArrayList();
        System.out.println("Initial size of al: " + al.size());
        al.add("C");
        al.add("A");
        al.add("E");
        al.add("B");
        al.add("D");
        al.add("F");
        al.add(1,"A2");

        System.out.println("Size of al after additions : "+ al.size());
        System.out.println("contents of al : "+al);

        al.remove("F");
        al.remove(2);

        System.out.println("Size of al after deletions : " + al.size());
        System.out.println("Contents of al : " +al);
    }
}
/*
```

```
Initial size of al: 0
Size of al after additions : 7
contets of al : [C, A2, A, E, B, D, F]
Size of al after deletions : 5
Contents of al : [C, A2, E, B, D] */
A Hashtable collection is not ordered.
```

```
=====
Vector :->
```

It is same as ArrayList
but Vector is synchronized.

```
=====
Obtaining Array from an ArrayList
Object[] toArray()
```

```
=====
The Set Interface
```

A Set cares about uniqueness—it doesn't allow duplicates.

collection that does not allow duplicate elements.

HashSet :->

*A HashSet is an unsorted, unordered Set.

*HashSet implements the Set interface.

*It creates a collection that uses a hash table for storage.

*A hash table stores information by using a mechanism called hashing.

*In hashing, the informational content of a key is used to determine a unique value, called its hash code.

*The hash code is then used as the index at which the data associated with the key is stored.

*The transformation of the key into its hash code is performed automatically—you never see the hash code itself.

*Use this class when you want a collection with no duplicates and you don't care about order when you iterate through it.

Constructors:

HashSet()

HashSet(Collections c)

Example HashSetDemo.java

```
import java.util.*;
class HashSetDemo
{
    public static void main(String args[])
    {
        HashSet hs = new HashSet();
        System.out.println("Initial size of ts: " + hs.size());

        hs.add("C");
        hs.add("A");
        hs.add("E");
        hs.add("B");
        hs.add("D");
        hs.add("F");
        System.out.println(hs);
    }
}
/*
Initial size of ts: 0
[D, A, F, C, B, E]
*/
```

TreeSet :

- The TreeSet is sorted collections.
- TreeSet provides an implementation of the Set interface that uses a tree for storage.
- Objects are stored in sorted, ascending order. Access and retrieval times are quite fast, which makes TreeSet an excellent choice when storing large amounts of sorted information that must be found quickly.

TreeSet() :- Constructs a new, empty set, sorted according to the elements' natural order.

TreeSet(Collection c) :

TreeSet(Comparator c) : Constructs a new, empty set, sorted according to the specified comparator.

TreeSet(SortedSet s) :

```
import java.util.*;
class TreeSetDemo {
    public static void main(String args[]) {
        TreeSet ts = new TreeSet();
        System.out.println("Initial size of ts: " + ts.size());
        ts.add("C");
        ts.add("A");
        ts.add("E");
        ts.add("B");
        ts.add("D");
        ts.add("F");
        System.out.println(ts);
    }
}
/*
[A, B, C, D, E, F]
*/
```

Iterator :

Iterator iterator() : Returns an iterator over the elements in this collection.

1)boolean hasNext() : Returns true if the iteration has more elements.

2)Object next() : Returns the next element in the iteration.

3) void remove() : Removes from the underlying collection the last element returned by the iterator (optional operation).

ListIterator :

ListIterator listIterator(int index)

boolean hasPrevious() : Returns true if this list iterator has more elements when traversing the list in the reverse direction.

Object previous() : Returns the previous element in the list.

void set(Object o): Replaces the last element returned by next or previous with the specified element (optional operation).

steps to use an Iterator:

1. Obtain an iterator to the start of the collection by calling the collection's iterator() method.
2. Set up a loop that makes a call to hasNext(). Have the loop iterate as long as hasNext() returns true.
3. Within the loop, obtain each element by calling next().


```

1) Iterator itr=al.iterator() ;

2) while(itr.hasNext()) {

3)   Object ob=itr.next();

   }

```

```

//IteratorDemo.java
import java.util.*;
class IteratorDemo {
    public static void main(String args[]) {
        ArrayList al = new ArrayList();
        al.add("C");
        al.add("A");
        al.add("E");
        al.add("B");
        al.add("D");
        al.add("F");
        System.out.println("Original contents of al : "+al);

        Iterator itr = al.iterator();
        while(itr.hasNext()) {
            Object element = itr.next();
            System.out.println(element + " ");
        }
        System.out.println();

        ListIterator litr = al.listIterator();
        while (litr.hasNext()) {
            Object element = litr.next();
            litr.set(element + "+");
        }
        System.out.print("Modified contents of al : ");
        itr = al.iterator();
        while(itr.hasNext()) {
            Object element = itr.next();
            System.out.print(element + " ");
        }
        System.out.println();
        System.out.print("Modified list backwards : ");
        while(litr.hasPrevious()) {
            Object element = litr.previous();
            System.out.print(element + " ");
        }
        System.out.println();
    }
}
/*
Original contents of al : [C, A, E, B, D, F]
C
A

```

E
B
D
F
Modified contents of al : C+ A+ E+ B+ D+ F+
Modified list backwards : F+ D+ B+ E+ A+ C+
*/

Map Interface

Methods Defined by Maps

*A map is an object that stores associations between keys and values, or key/value pairs.

*Given a key, you can find its value. Both keys and values are objects.

*The keys must be unique, but the values may be duplicated.

values, others cannot.

- 1) void clear()
Removes all key/value pairs from the invoking Map.
- 2) boolean containsKey(Object key)
Returns true if this map contains a mapping for the specified key.
- 3) boolean containsValue(Object value)
Returns true if this map maps one or more keys to the specified value.
- 4) Set entrySet()
Returns a set view of the mappings contained in this map.
- 5) boolean equals(Object o)
- 6) Object get(Object key)
Returns the value to which this map maps the specified key.
- 7) int hashCode()
Returns the hash code value for invoking map.
- 8) boolean isEmpty()
Returns true if this map contains no key-value mappings.
- 9) Set keySet()
Returns a set view of the keys contained in this map.
- 10) Object put(K key, V value)
Associates the specified value with the specified key in this map (optional operation).
- 11) void putAll(Map m)
Copies all of the mappings from the specified map to this map (optional operation).
- 12) remove(Object key)
Removes the mapping for this key from this map if it is present (optional operation).
- 13) int size()
Returns the number of key-value mappings in this map.
- 14) Collection values()
Returns a collection view of the values contained in this map.

=====

2) The SortedMap Interface

- 1) Comparator comparator()
- 2) Object firstKey()
- 3) SortedMap headMap(Object end)
- 4) Object lastKey()
- 5) SortedMap subMap(Object start, Object end)

6) SortedMap tailMap(Object start)

Returns a view of the portion of this sorted map whose keys are greater than or equal to fromKey.

=====

3) The Map.Entry Interface

It is the inner class of Map

1) Object getKey()

Returns the key corresponding to this entry.

2) Object getValue()

Returns the value corresponding to this entry.

3) Object setValue(V value)

Replaces the value corresponding to this entry with the specified value .

=====

The Map classes

classes	desription
---------	------------

1)AbstractMap :->	
-------------------	--

2) HashMap :->	
----------------	--

3)TreeMap :->	
---------------	--

4)WeakHashMap :->	weak
-------------------	------

1)HashMap

The HashMap gives you an unsorted, unordered Map. When you need a Map and you don't care about the order (when you iterate through it),

HashMap ()

HashMap (Map m)

HashMap(int initialcapacity)

//example HashMapDemo.java

=====

TreeMap

*TreeMap is a sorted Map.

*The TreeMap class implements the Map interface by using a tree.

* A TreeMap provides an efficient means of storing key/value pairs in sorted order, and allows rapid retrieval.

*unlike a hash map, a tree map guarantees that its elements will be sorted in ascending key order.

1)TreeMap()

2)TreeMap(Comparator c)

sorted according to the given comparator.

3)TreeMap(Map m)

4)TreeMap(SortedMap m)

Constructs a new map containing the same mappings as the given SortedMap, sorted according to the same ordering.

=====

Comparator

int compare() and equals()

compare(Object obj1, Object obj2)

int compareTo()----->

obj1==obj2 0
obj1>obj2 +ve
obj1<obj2 -ve

boolean equals(Object obj)

=====

Legacy classes

- 1) Vector
- 2) Stack
- 3) Dictionary
- 4) HashTable
- 5) Properties

The Properties class represents a persistent set of properties. The Properties can be saved to a stream or loaded from a stream. Each key and its corresponding value in the property list is a string.

void load(InputStream inStream)

Reads a property list (key and element pairs) from the input stream.

void store(OutputStream out, String comments)

Writes this property list (key and element pairs) in this Properties table to the output stream in a format suitable for loading into a Properties table using the load method.

String getProperty(String key)

Searches for the property with the specified key in this property list.

Interface Enumeration

boolean hasMoreElements()

Object nextElement()

=====

StringTokenizer

The string tokenizer class allows an application to break a string into tokens. The tokenization method is much simpler than the one used by the StreamTokenizer class. The StringTokenizer methods do not distinguish among identifiers, numbers, and quoted strings, nor do they recognize and skip comments.

The set of delimiters (the characters that separate tokens) may be specified either at creation time or on a per-token basis.

An instance of StringTokenizer behaves in one of two ways, depending on whether it was created with the returnDelims flag having the value true or false:

If the flag is false, delimiter characters serve to separate tokens. A token is a maximal sequence of consecutive characters that are not delimiters.

If the flag is true, delimiter characters are themselves considered to be tokens. A token is thus either one delimiter character, or a maximal sequence of consecutive characters that are not delimiters.

A StringTokenizer object internally maintains a current position within the string to be tokenized. Some operations advance this current position past the characters processed.

A token is returned by taking a substring of the string that was used to create the StringTokenizer object.

The following is one example of the use of the tokenizer. The code:

```
StringTokenizer st = new StringTokenizer("this is a test");
while (st.hasMoreTokens()) {
    System.out.println(st.nextToken());
}
```

prints the following output:

```
this
is
a
test
```

StringTokenizer is a legacy class that is retained for compatibility reasons although its use is discouraged in new code. It is recommended that anyone seeking this functionality use the split method of String or the java.util.regex package instead.

The following example illustrates how the String.split method can be used to break up a string into its basic tokens:

```
String[] result = "this is a test".split("\\s");
for (int x=0; x<result.length; x++)
    System.out.println(result[x]);
```

prints the following output:

```
this
is
a
test
```

1)StringTokenizer(String str)

Constructs a string tokenizer for the specified string.

2)StringTokenizer(String str, String delim)

Constructs a string tokenizer for the specified string. The characters in the delim argument are the delimiters for separating tokens. Delimiter characters themselves will not be treated as tokens.

3)StringTokenizer(String str, String delim, boolean returnDelims)

Constructs a string tokenizer for the specified string. If returnDelims is true

String nextToken()

Returns the next token from this string tokenizer.

boolean hasMoreTokens()

Tests if there are more tokens available from this tokenizer's string.

Example on User Object:

```
import java.util.*;
class Address
{
    String name;
    String city;
    int pcode;

    Address(String name, String city,int pcode)
    {
        this.name=name;
        this.city=city;
        this.pcode=pcode;
    }

    public String toString()
    {
        return "name="+name+"\n"+"City="+city+"\n"+"pcode="+pcode+"\n";
    }
}

class UserObject
{
    public static void main(String args[])
    {
        ArrayList al = new ArrayList();

        al.add(new Address("Anil","Nagpur",12345));
        al.add(new Address("Sunil","Kanpur",123456));
        al.add(new Address("Rajan","Pune",0000));

        //System.out.println(" "+al);

        Iterator itr=al.iterator();

        while(itr.hasNext())
        {
            Object e=itr.next();
            System.out.println("\n"+e);
        }
    }
}
```

Example on LinkedListDemo

```
import java.util.*;

class LinkedListDemo
```

```

{
    public static void main(String args[])
    {
        LinkedList ll = new LinkedList();
        System.out.println("Initial size of al: " + ll.size());

        ll.add("C");
        ll.add("A");
        ll.add("E");
        ll.add("B");
        ll.add("D");
        ll.add("F");
        ll.addLast("A2");
        ll.addFirst("Z");

        ll.add(1,"G");

        System.out.println("Size of al after additions : "+ ll.size());

        System.out.println("contents of al : "+ll);

        ll.remove("F");
        ll.remove(2);

        System.out.println("Size of al after deletions : " + ll.size());
        System.out.println("contents of al : "+ll);
        ll.removeFirst();
        ll.removeLast();

        System.out.println("Contents of ll after removing first and last element: " +ll);

        Object val=ll.get(3);
        ll.set(3,"hello");
        System.out.println("Contents of ll after changed : "+ll);

    }
}

/*
Initial size of al: 0
Size of al after additions : 9
contents of al : [Z, G, C, A, E, B, D, F, A2]
Size of al after deletions : 7
Contents of ll after removing first and last element: [G, A, E, B, D]
Contents of ll after changed : [G, A, E, Bhello, D]
*/

```

Example on HashMapDemo:

```
import java.util.*;
```



```

class HashMapDemo
{
    public static void main(String args[])
    {
        HashMap hm = new HashMap();

        hm.put("John Doe",new Double(3434.34));
        hm.put("Tom Smith",new Double(123.22));
        hm.put("Jane Baker",new Double(1378.00));
        hm.put("Todd Hall",new Double(99.22));
        hm.put("Ralph Smith",new Double(-19.08));

        Set set = hm.entrySet();//

        Iterator i = set.iterator();

        while(i.hasNext())
        {
            Map.Entry me = (Map.Entry)i.next();
            System.out.print(me.getKey() + ": ");
            System.out.println(me.getValue());
        }
        System.out.println();

        double balance = ((Double)hm.get("John Doe")).doubleValue();//
        hm.put("John Doe",new Double(balance + 1000));
        System.out.println("John Doe's new balance : " + hm.get("John Doe"));
    }
}
/*
Todd Hall: 99.22
Ralph Smith: -19.08
Tom Smith: 123.22
John Doe: 3434.34
Jane Baker: 1378.0

John Doe's new balance : 4434.34
*/

```

i)Internet Protocol(IP) :->

is a low-level routing protocol that breaks the data into small packets and sends them to an address across a network, which does not guarantee to deliver said packets to the destination.

ii)TCP (Transmission Control Protocol) :--> is a higher-level protocol that manages to robustly string together packets, sorting and retransmitting them as necessary to reliably transmit your data.

iii)UDP (User Datagram Protocol) :-> sits next to TCP and can be used directly to support fast, connectionless, unreliable transport of packets.

Client/Server

You often hear the term client/server mentioned in the context of networking. A server is anything that has some resource that can be shared. There are computer servers, which provide computing power; print servers, which manage a collection of printers; disk servers, which provide networked disk space; and web servers, which store web pages. A client is simply any other entity that wants to gain access to a particular server. The interaction between client and server is just like the interaction between a lamp and an electrical socket. The server is a permanently available resource, while the client is free to “unplug” after it has been served.

Proxy Servers

A proxy server speaks the client side of a protocol to another server. This is often required when clients have certain restrictions on which servers they can connect to. Thus, a client would connect to a proxy server, which did not have such restrictions, and the proxy server would in turn communicate for the client. A proxy server has the additional ability to filter certain requests or cache the results of those requests for future use. A caching proxy HTTP server can help reduce the bandwidth demands on a local network's connection to the Internet. When a popular web site is being hit by hundreds of users, a proxy server can get the contents of the web server's popular pages once, saving expensive internet work transfers while providing faster access to those pages to the clients.

Internet Addressing

Every computer on the Internet has an address. An Internet address is a number that uniquely identifies each computer on the Net. Originally, all Internet addresses consisted of 32-bit values.

Domain Naming Service (DNS)

DNS server manages the mapping between the domain name and the IP address. Domain Naming Service (DNS). Just as the four numbers of an IP address describe a network hierarchy from left to right, the name of an Internet address, called its domain name, describes a machine's location in a name space, from right to left.

For example, www.hotmail.com is in the COM domain (reserved for U.S. commercial sites), it is called hotmail (after the company name), and www is the name of the specific computer that is hotmail's web server. www corresponds to the rightmost number in the equivalent IP address.

All the domain names and the IP addresses are stored in DNS server. The network administrator is responsible to tell that if one cannot find out the IP address of a particular company in a DNS. Then contact the nearest DNS in another network. If it is not available in that network also go to the other nearest network at any point of time you can find out that address. In most of the cases m/c's are configured to reach multiple DNS's on a network.

java.net package & Class InetAddress :

The InetAddress class is used to encapsulate both the numerical IP address we discussed earlier and the domain name for that address. You interact with this class by using the name of an IP host, which is more convenient and understandable than its IP address. The InetAddress class hides the number inside.

Factory Methods

The InetAddress class has no visible constructors. To create an InetAddress object, you have to use one of the available factory methods.

1. static InetAddress getLocalHost() :- method simply returns the InetAddress object that represents the local host.

2. static InetAddress getByName(String hostName): method returns an InetAddress for a host name passed to it.
3. static InetAddress[] getAllByName(String hostName): returns an array of InetAddresses that represent all of the addresses that a particular name resolves to.

Instance Methods

The InetAddress class also has several other methods, which can be used on the objects returned by the methods just discussed.

String getAddress(): Returns a string that represents the host address associated with the InetAddress object.

String getHostName(): Returns a string that represents the host name associated with the InetAddress object.

Example :

```
import java.net.*;
class InetAddressDemo {
    public static void main(String arg[]) {
        try{
            if(arg.length!=1) throw new IllegalArgumentException();
            InetAddress host= InetAddress.getByName(arg[0]);
            System.out.println("IPAddress "+host.getAddress());
            System.out.println("Domain Name "+host.getHostName());
            System.out.println(host);
        }catch(Exception e) {
            System.out.println("InetAddressDemo :"+e);
        }
    }
}
```

```
D:\network>java InetAddressDemo localhost
IPAddress 127.0.0.1
Domain Name localhost
localhost/127.0.0.1
```

TCP/IP Sockets

TCP/IP sockets are used to implement reliable, bidirectional, persistent, point-to- point, stream-based connections between hosts on the Internet. A socket can be used to connect Java's I/O system to other programs that may reside either on the local machine or on any other machine on the Internet. There are two kinds of TCP/IP sockets in Java.

- One is for servers, and the other is for clients.
- The ServerSocket class is designed to be a “listener,” which waits for clients to connect before doing anything.

The Socket class is designed to connect to ServerSocket and initiate protocol exchanges.

TCP/IP Client Sockets

The creation of a Socket object implicitly establishes a connection between the client and server. There are no methods or constructors that explicitly expose the details of establishing that connection.

Here are two constructors used to create client sockets:

Socket(String hostName, int port)

Socket(InetAddress ipAddress, int port)

Methods :->

- 1)InetAddress getAddress(): Returns the InetAddress associated with the Socket object.
- 2)int getPort() :Returns the remote port to which this Socket object is connected.
- 3)int getLocalPort() Returns the local port to which this Socket object is connected.

- 4) `InputStream getInputStream()` : Returns the `InputStream` associated with the invoking socket.
5) `OutputStream getOutputStream()` : Returns the `OutputStream` associated with the invoking socket.

TCP/IP Server Sockets

- The `ServerSocket` class is used to create servers that listen for either local or remote client programs to connect to them on published ports.
- `ServerSocket`'s are quite different from normal Sockets. When you create a `ServerSocket`, it will register itself with the system as having an interest in client connections. The constructors for `ServerSocket` reflect the port number that you wish to accept connections on.

Constructors:

`ServerSocket(int port)` : Creates server socket on the specified port with a queue length of 50.

`ServerSocket(int port, int maxQueue)` :

- `ServerSocket` has a method called `accept()`, which is a blocking call that will wait for a client to initiate communications, and then return with a normal `Socket` that is then used for communication with the client.
`Socket accept()`

Server.java

```
import java.net.*;
import java.io.*;
public class Server {
    public static void main(String arg[]) throws Exception {
        ServerSocket ss=new ServerSocket(65533);
        while(true) {
            Socket s=ss.accept();
            ServerThread est=new ServerThread(s);
            Thread t=new Thread(est);
            t.start();
        }
    }
}
```

ServerThread.java

```
import java.net.*;
import java.io.*;
public class ServerThread implements Runnable {
    Socket s;
    public ServerThread(Socket s) {
        this.s=s;
    }
    public void run() {
        try{
            InputStream in=s.getInputStream();
            DataInputStream dis=new DataInputStream(in);
            String line=dis.readLine();
            System.out.println("Message came from : "+s.getInetAddress()+" : msg is:"+line);
            PrintStream out=new PrintStream(s.getOutputStream());
            out.println("Message returned back from server is :"+line);
            out.close();
            dis.close();
            in.close();
        }catch(Exception e){
            e.printStackTrace();
        }
    }
}
```

Client.java

```
import java.net.*;
import java.io.*;
public class Client{
    public static void main(String arg[]) throws Exception{
        Socket s=new Socket("localhost",65533);
        //Socket s=new Socket("127.0.0.1",65533);
        PrintStream out=new PrintStream(s.getOutputStream());
        out.println(arg[0]);
        DataInputStream in=new DataInputStream(s.getInputStream());
        String response=in.readLine();
        System.out.println(response);
        in.close();
        out.close();
        s.close();
    }
}
```

D:\Client_Server>javac *.java

D:\Client_Server>start java Server

D:\Client_Server>java Client Hello_God

URL

It is about WWW, the World Wide Web. The Web is a loose collection of higher-level protocols and file formats, all unified in a web browser. One of the most important aspects of the Web is that Tim Berners-Lee devised a scalable way to locate all of the resources of the Net. Once you can reliably name anything and everything, it becomes a very powerful paradigm. The Uniform Resource Locator (URL) does exactly that.

The URL provides a reasonably intelligible form to uniquely identify or address information on the Internet. URLs are ubiquitous; every browser uses them to identify information on the Web. In fact, the Web is really just that same old Internet with all of its resources addressed as URLs plus HTML. Within Java's network class library, the URL class provides a simple, concise API to access information across the Internet using URLs.

Examples of URLs are <http://www.yahoo.com/>
and <http://www.yahoo.com:80/index.htm>.

A URL specification is based on four components.

- The first is the protocol to use, separated from the rest of the locator by a colon (:). Common protocols are http, ftp, gopher, and file, Although these days almost everything is being done via HTTP (in fact, most browsers will proceed correctly if you leave off the "http://" from your URL specification).
- The second component is the host name or IP address of the host to use. This is delimited on the left by double slashes (//) and on the right by a slash (/) or optionally a colon (:).
- The third component, the port number, is an optional parameter, delimited on the left from the host name by a colon (:) and on the right by a slash (/). (It defaults to port 80, the predefined HTTP port; thus ":80" is redundant.)
- The fourth part is the actual file path. Most HTTP servers will append a file named index.html or index.htm to URLs that refer directly to a directory resource.
Thus, <http://www.yahoo.com/> is the same as <http://www.yahoo.com/index.htm>.

URL class constructor

1) URL(String urlSpecifier)

The next two forms of the constructor allow you to break up the URL into its component parts:

2) URL(String protocolName, String hostName, int port, String path)

3) URL(String protocolName, String hostName, String path)

4) URL(URL urlObj, String urlSpecifier)

Example :->

```
import java.net.*;
class URLEDemo {
    public static void main(String args[]) throws MalformedURLException {
        URL hp = new URL("http://www.yahoo.com/index.html");
        System.out.println("Protocol: " + hp.getProtocol());
        System.out.println("Port: " + hp.getPort());
        System.out.println("Host: " + hp.getHost());
        System.out.println("File: " + hp.getFile());
        System.out.println("Ext:" + hp.toExternalForm());
    }
}
/*
```

```
Protocol: http
Port: -1
Host: www.yahoo.com
File: /index.html
Ext:http://www.yahoo.com/index.htm */
```

the port is -1; this means that one was not explicitly set

URLConnection

URLConnection is a general-purpose class for accessing the attributes of a remote resource. Once you make a connection to a remote server, you can use URLConnection to inspect the properties of the remote object before actually transporting it locally. These attributes are exposed by the HTTP protocol specification and, as such, only make sense for URL objects that are using the HTTP protocol. To access the actual bits or content information of a URL, you create a URLConnection object from it, using URLConnection.openConnection().

In the following example, we create a URLConnection using the openConnection() method of a URL object and then use it to examine the document's properties and content:

```
// Demonstrate URLConnection.
import java.net.*;
import java.io.*;
import java.util.Date;
class UCDemo{
    public static void main(String args[]) throws Exception {
        int c;
        URL hp = new URL("http://www.yahoo.com");
        URLConnection hpCon = hp.openConnection();
        // get content type
        System.out.println("Content-Type: " + hpCon.getContentType());
        // get content length
        int len = hpCon.getContentLength();
        System.out.println("Content-Length: " + len);
        if(len != 0) {
            System.out.println("=== Content ===");
            InputStream input = hpCon.getInputStream();
            int i = len;
            while (((c = input.read()) != -1)) {
                System.out.print((char) c);
            }
            input.close();
        } else {
```

```
    System.out.println("No content available.");  
  }  
}  
}
```

The program establishes an HTTP connection to www.yahoo.com over port 80.

Datagrams

For most of your internetworking needs, you will be happy with TCP/IP-style networking. It provides a serialized, predictable, reliable stream of packet data. This is not without its cost, however. TCP includes many complicated algorithms for dealing with congestion control on crowded networks, as well as pessimistic expectations about packet loss. This leads to a somewhat inefficient way to transport data.

Datagrams provide an alternative. Datagrams are bundles of information passed between machines. They are somewhat like a hard throw from a well-trained but blindfolded catcher to the third baseman. Once the datagram has been released to its intended target, there is no assurance that it will arrive or even that someone will be there to catch it. Likewise, when the datagram is received, there is no assurance that it hasn't been damaged in transit or that whoever sent it is still there to receive a response. Java implements datagrams on top of the UDP protocol by using two classes: The `DatagramPacket` object is the data container, while the `DatagramSocket` is the mechanism used to send or receive the `DatagramPackets`.