Class Fundamentals

Objects and classes are fundamental parts of object-orientated programming, and therefore Java. Any concept you wish to implement in a Java program must be encapsulated within a class.

Classes provide a convenient way for packing together a group of logically related data items and functions that work on them. The most important thing to understand about a class is that it defines a new data type. Once defined, this new type can be used to create objects of that type. Class is a template for multiple objects with similar features. Classes embody all the features of a particular set of objects. When you write a program in an object-oriented language, you don't define individual objects. You define classes from which you can create objects.

The General Form of a Class

When you define a class, you declare its exact form and nature. You do this by specifying the data that it contains and the code that operates on that data. While very simple classes may contain only code or only data, most real-world classes contain both. As you will see, a class' code defines the interface to its data.

A class is declared by use of the **class** keyword. The general form of a class definition is shown:

```
class classname
{
      type instance-variable1;
      type instance-variable2;
       ...
      type instance-variableN;
   type methodname1(parameter-list)
    {
   `   body of method
    }
type methodname2(parameter-list)
    {
      body of method
   }
  ……
type methodnameN(parameter-list)
   {
       body of method
   }
}
```

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

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

A Simple Class

Let's begin our study of the class with a simple example. Here is a class called Box that defines three instance variables: width, height, and depth. Currently, Box does not contain any methods

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

As stated, a **class** defines a new type of data. In this case, the new data type is called Box. You will use this name to declare objects of type Box. A class declaration creates a template; it does not create an actual object. Thus the above example does not cause any objects of type Box to come into existence.

The following line of code creates an object of Box called **mybox**

```
Box mybox = new Box();
```

Each time you create an instance of a class, you are creating an object that contains its own copy of each instance variable defined by the class. Thus, every Box object will contain its own copies of the instance variables width, height, and depth.

Declaring Objects

On creating a class we are creating a new data type. You can use this type to declare **objects** of that type. Obtaining objects of a class is a two-step process.

First, you must declare a variable of the class type. This variable does not define an object. Instead, it is simply a variable that can refer to an object.

Second, you must acquire an actual, physical copy of the object and assign it to that variable. You can do this using the **new** operator. The new operator dynamically allocates (that is, allocates at run time) memory for an object and returns a reference to it. This reference is, more or less, the address in memory of the object allocated by **new**.

Eg: ***Box mybox; // declare reference to object***
mybox = new Box(); // allocate a Box object

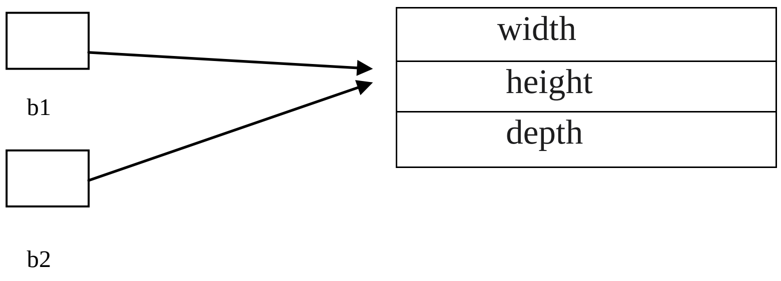
The first line declares mybox as a reference to an object of type Box. After this line executes, mybox contains the value null, which indicates that it does not yet point to an actual object. Any attempt to use mybox at this point will result in a compile-time error. The next line allocates an actual object and assigns a reference to it to mybox. After the second line
executes, you can use mybox as if it were a Box object.

Assigning Object Reference Variables

Object reference variables act differently than you might expect when an assignment takes place

```
Box b1 = new Box();
Box b2 = b1;
```

After executing the above code, **b1** and **b2** will both refer to the same object. The assignment of **b1** to **b2** did not allocate any memory or copy any part of the original object. It simply makes **b2** refer to the same object as does **b1**. Thus, any changes made to the object through **b2** will affect the object to which **b1** is referring, since they are the same object.



Although **b1** and **b2** both refer to the same object, they are not linked in any other way. For example, a subsequent assignment to **b1** will simply unhook b**1** from the original object without affecting the object or affecting **b2**.

Eg:
```
Box b1 = new Box();
Box b2 = b1;
// ...
b1 = null;
```

Here, **b1** has been set to null, but **b2** still points to the original object.

Methods in Classes

Classes usually consist of two things: instance variables and methods.

This is the general form of a method:

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

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

return value; where, **value** is the value returned.

The type of data returned by the method must be compatible with the return type specified by the method. Also the variable receiving the value returned by a method must also be compatible with the return type specified for the method

Accessing Class members

Each object created has its own set of variables.Values should be assigned to these variables to use them in the program. In order to access the instance variables and methods **dot operator(.)** is used. The concerned object and dot operator is used as shown below:

```
Objectname.variablename=value;
Objectname.methodname(parameter-list);
```

objectname is the name of the object, whose members we want to access.
variablename is the name of instance variable inside the object we want to access
methodname is the method we wish to call and parameter list is the list of values that should be passed to the function definition. It should match in type and number with the parameters in the function definition.

Eg:
```
class Box
   {
      double width;
      double height;
      double depth;
```

```
            double volume()
            {
              return width * height * depth;
            }
           void setDim(double w, double h, double d)
           {
             width = w;
            height = h;
           depth = d;
            }
    }
     class BoxDemo3
     {
           public static void main(String args[])
            {
                Box mybox1 = new Box();
               Box mybox2 = new Box();
               double vol;
              mybox1.setDim(10, 20, 15);
           mybox2.setDim(3, 6, 9);
            vol = mybox1.volume();
           System.out.println("Volume is " + vol);
           vol = mybox2.volume();
           System.out.println("Volume is " + vol);
         }
 }
```

This program generates the following output

```
Volume is 3000.0
Volume is 162.0
```

Here setDim() is a parameterized function used to set the dimensions of each box For example, when mybox1.setDim(10, 20, 15); is executed, 10 is copied into parameter w, 20 is copied into h, and 15 is copied into d. Inside setDim( ) the values of w, h, and d are then
assigned to width, height, and depth, respectively.

Constructors

Java allows objects to initialize themselves when they are created. This automatic initialization is performed through the use of a constructor.

A constructor initializes an object immediately upon creation. 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 have no return type, not even void. This is because It is the constructor's job to initialize the internal state of an object so that the code creating an
instance will have a fully initialized, usable object immediately.

You don't have to provide any constructors for your class. In that case the compiler automatically provides a no-argument, default constructor for any class without constructors. The default constructor automatically initializes all instance variables to zero. If a class defines an explicit constructor, it no longer has a default constructor to set the state of the objects.
If such a class requires a default constructor, its implementation must be provided Now let's reexamine the new operator. As you know, when you allocate an object, you use the following general form:

class-var = new classname( );

Now you can understand why the parentheses are needed after the class name. What is actually happening is that the constructor for the class is being called. Thus, in the line

Box mybox1 = new Box(); new Box( ) is calling the Box( ) constructor.

We can add parameters to the constructor, which makes them much more useful. Constructors may include parameters of various types. When the constructor is invoked using the **new** operator, the types must match those that are specified in the constructor definition.

For example, the following version of Box defines a parameterized constructor which sets the dimensions of a box as specified by those parameters.

```
Box(double w, double h, double d)
{
        width = w;
         height = h;
         depth = d;
}
Box mybox1 = new Box(10, 20, 15);
```

Here **new Box(10, 20, 15)** creates space in memory for the object and initializes its fields. Thus, mybox1's copy of width, height, and depth will contain the values 10, 20, and 15, respectively.

It is possible to use **this()** construct, to implement local chaining of constructors in a class. The **this()** call in a constructor invokes another constructor within the same class with a different parameter list**.**

```
 public class Platypus
  {
       String name;
      Platypus(String input)
       {
            name = input;
      }
     Platypus()
      {
            this("John/Mary Doe");
      }
     public static void main(String args[])
       {
            Platypus p1 = new Platypus("digger");
            Platypus p2 = new Platypus();
       }
}
```

In the code, there are two constructors. The first takes a String input to name the instance. The second, taking no parameters, calls the first constructor by the default name "John/Mary Doe".

Constructors use **super** to invoke the super class's constructor

```
public class SuperClassDemo
{
     SuperClassDemo()
     {
      }
}

class Child extends SuperClassDemo
 {
    Child()
    {
        super();
    }
}
```

In the above example, the constructor Child() includes a call to **super**, which causes the class SuperClassDemo to be instantiated, in addition to the Child class

Garbage Collection

In the Java programming language, dynamic allocation of objects is achieved using the **new** operator. An object once created uses some memory and the memory remains allocated till there are references for the use of the object. When there are no references for an object, it is assumed to be no longer needed and the memory occupied by the object can be reclaimed.

There is no explicit need to destroy an object as java handles the de-allocation automatically. The technique that accomplishes this is known as **Garbage Collection**. When no references to an object exist, that object is assumed to be no longer needed, and the memory occupied by the object can be reclaimed so that the space is made available for subsequent new objects. It relieves java programmer from memory management. Garbage collector (gc) is a thread, which is running behind the scenes, which de-allocates memory of unused objects in the heap. JVM watches your heap memory when you are running out of memory it invokes gc which deallocates memory of unused objects in the heap.
Garbage collector will be invoked by the JVM. As a programmer if you want to call the gc, it
can be done by following code :

System.gc();

Where system is a class in java.lang.package and gc() is a static method in System class.
We cannot force the garbage collector i.e. when we call System.gc( ), JVM may or may not invoke garbage collector.

Sometimes an object will need to perform some action when it is destroyed. For example, if an object is holding some non-Java resource such as a file handle or window character font, then you might want to make sure these resources are freed before an object is destroyed To handle such situations we have **finalize ()** method. Before removing an object from memory garbage collection thread invokes **finalize ()** method of that object.

The finalize( ) method has this general form:

```
protected void finalize( )
  {
    finalization code here
 }
```

Here, the keyword **protected** is a specifier that prevents access to **finalize( )** by code defined outside its class. The Java runtime calls that method whenever it is about to recycle an object of that class. Inside the **finalize( )** method you will specify those actions that must be performed before an object is destroyed. The garbage collector runs periodically, checking for objects that are no longer referenced by any running state or indirectly through other referenced objects. Right before an asset is freed, the Java run time calls the **finalize( )** method on the object.

Garbage Collection Advantages :

» It helps ensure program integrity.
» The programmer is freed from deallocating memory in the programs.
» Garbage collection is an important part of java's security strategy. Java programmers are unable to accidentally or purposely crash the JVM by incorrectly freeing memory.

Garbage Collection Disadvantages :

» It adds an overhead that can affect program performance.
» The java Virtual machine has to keep track of which objects are being referenced by the executing program, and finalize and free unreferenced objects on the fly. This activity will likely required more CPU time than would have been required if the program explicitly freed unnecessary memory.

Overloaded 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.

Eg: public String printString(String string)
public String printString(String string, int offset)

When an overloaded method is invoked, Java uses the type and/or number of arguments as its guide to determine which version of the overloaded method to actually call. Thus, overloaded methods must differ in the type and/or number of their parameters. While overloaded methods may have different return types, the return type alone is insufficient to distinguish two versions of a method. When Java encounters a call to an overloaded method, it simply executes the version of the method whose parameters match the arguments used in the call.

The value of overloading is that it allows related methods to be accessed by use of a common name. You, the programmer, need only remember the general operation being performed. Through the application of polymorphism, several names have been reduced to one.

Overloading Constructors

Because the constructor's name is predetermined by the name of the class, there can be only one constructor name. But on some occasions we need to create different objects of the same class with different initial values. Java provides a mechanism called constructor overloading in which we can define multiple constructors which differ in number and/or types of parameters. Also, like overloaded methods, Java determines which constructor to use on the basis of the arguments passed to **new**. The advantage of constructor overloading is that appropriate constructors are created on the basis of the needs of the application.

```
class Box
{
double width;
double height;
double depth;
Box(double w, double h, double d)
 {
  width = w;
  height = h;
  depth = d;
}
Box()
 {
width = -1; /
height = -1;
depth = -1;
}
Box(double len)
{
       width = height = depth = len;
}
double volume()
{
           return width * height * depth;
}
           }
class OverloadCons
{
       public static void main(String args[])
         {
           Box mybox1 = new Box(10, 20, 15);
            Box mybox2 = new Box();
            Box mycube = new Box(7);
             double vol;
             vol = mybox1.volume();
             System.out.println("Volume of mybox1 is " + vol);
             vol = mybox2.volume();
```

```
          System.out.println("Volume of mybox2 is " + vol);
          vol = mycube.volume();
          System.out.println("Volume of mycube is " + vol);
          }
      }
```

In the above example we have defined 3 constructors, which differ in their parameters. The first one doesn't take any parameter .It initializes the dimensions of box to -1. Second one takes a single parameter called length and it initializes all dimensions with that value. Third constructor takes 3 parameters which correspond to the dimensions of the box.

Using Objects as Parameters

In java it is both correct and common to pass objects as parameters to methods. One of the most common uses of object parameters involves constructors. Frequently you will want to construct a new object so that it is initially the same as some existing object. To do this, you must define a constructor that takes an object of its class as a parameter.

```
class Box
{
   double width;
   double height;
   double depth;
   Box(Box ob)
    {
       width = ob.width;
       height = ob.height;
       depth = ob.depth;
    }
   Box(double w, double h, double d)
   {
       width = w;
       height = h;
       depth = d;
    }
   }
class OverloadCons2
{
  public static void main(String args[]) \
   {
     Box mybox1 = new Box(10, 20, 15);
     Box myclone = new Box(mybox1);
     double vol;
     vol = mybox1.volume();
     System.out.println("Volume of mybox1 is " + vol);
```

```
      vol = myclone.volume();
      System.out.println("Volume of clone is " + vol);
      }
 }
```

Here we have created a clone of 'mybox1' called 'myclone'.by passing 'mybox1' as an argument to the constructor. So all the instant variables of 'myclone' will be assigned the values of the instant variables of 'mybox1'.

In a method that takes object as a parameter, objects are passed by reference.When you create a variable of a class type, you are only creating a reference to an object. Thus, when you pass this reference to a method, the parameter that receives it will refer to the same object as that referred to by the argument. This effectively means that objects are passed to methods by use of call-by-reference. Changes to the object inside the method do affect the object used as an argument.

```
class Test
{
  int a, b;
  Test(int i, int j)
   {
     a = i;
     b = j;
   }
void meth(Test o)
 {
  o.a *= 2;
  o.b /= 2;
 }
}
class CallByRef
{
public static void main(String args[])
  {
      Test ob = new Test(15, 20);
      System.out.println("ob.a and ob.b before call: " +
      ob.a + " " + ob.b);
      ob.meth(ob);
      System.out.println("ob.a and ob.b after call: " +
      ob.a + " " + ob.b);
   }
}
```

```
ob.a and ob.b before call: 15 20
ob.a and ob.b after call: 30 10
```

As you can see, in this case, the actions inside meth( ) have affected the object used as an argument.

Recursion

Java supports recursion. Recursion is the process of defining something in terms of itself. As it relates to Java programming, recursion is the attribute that allows a method to call itself. A method that calls itself is said to be recursive. The classic example of recursion is the computation of the factorial of a number. The factorial of a number N is the product of all the whole numbers between 1 and N.For example, 3 factorial is 1 × 2 × 3, or 6.The main advantage to recursive methods is that they can be used to create clearer and simpler versions of several algorithms than can their iterative relatives

```
class Factorial
{
   int fact(int n)
    {
        int result;
        if ( n ==1) return 1;
        result = fact (n-1) * n;
       return result;
   }
}

class Recursion
  {
   public static void main (String args[])
   {
      Factorial f =new Factorial();
     System.out.println(“Factorial of 3 is “ + f.fact(3));
    System.out.println(“Factorial of 3 is “ + f.fact(4));
    System.out.println(“Factorial of 3 is “ + f.fact(5));

   }
  }
```

The output from this program is shown here:

```
Factorial of 3 is 6
Factorial of 4 is 24
Factorial of 5 is 120
```

When fact( ) is called with an argument of 1, the function returns 1; otherwise it returns the product of fact(n–1)*n. To evaluate this expression, fact( ) is called with n–1. This process repeats until n equals 1 and the calls to the method begin returning.

For example, when you compute the factorial of 3, the first call to fact( ) will cause a second call to be made with an argument of 2. This invocation will cause fact( ) to be called a third time with an argument of 1. This call will return 1, which is then multiplied by 2 (the value of n in the second invocation). This result (which is 2) is then returned to the original invocation of fact( ) and multiplied by 3 (the original value of n). This yields the answer, 6.

When a method calls itself, new local variables and parameters are allocated storage on the stack, and the method code is executed with these new variables from the start. A recursive call does not make a new copy of the method. Only the arguments are new. As each recursive call returns, the old local variables and parameters are removed from the stack, and execution resumes at the point of the call inside the method.

Recursive versions of many routines may execute a bit more slowly than the iterative equivalent because of the added overhead of the additional function calls. The main advantage to recursive methods is that they can be used to create clearer and simpler versions of several algorithms than iterative ones. When writing recursive methods, you must have an if statement somewhere to force the method to return without the recursive call being executed. If you
don't do this, once you call the method, it will never return.

Understanding static

Normally a class member must be accessed only in association with an object of its class. But sometimes you will want to define a class member that can be used independently of any object of that class. In Java it is possible to create a member that can be used by itself, without reference to a specific instance by preceding 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.

Methods declared as static have several restrictions:

■ They can only call other static methods.

■ They must only access static data.
■ They cannot refer to this or super in any way.

If you need to do computation in order to initialize your static variables, you can declare a static block which gets executed exactly once, when the class is first loaded. The following example shows a class that has a static method, some static variables, and a static initialization block:

```
class UseStatic
{
  static int a = 3;
  static int  b;
  static void meth(int x)
  {
    System.out.println("x = " + x);
    System.out.println("a = " + a);
    System.out.println("b = " + b);
  }
  static
  {
    System.out.println("Static block initialized.");
    b = a * 4;
  }
  public static void main(String args[])
  {
      meth(42);
  }
}
```

Output

```
Static block initialized.
  x = 42
  a = 3
  b = 12
```

UseStatic class is loaded, all of the static statements are run. First, a is set to 3, then the static block executes (printing a message), and finally, b is initialized to a * 4 or 12. Then main( ) is called, which calls meth( ), passing 42 to x. The three println( ) statements refer to the two static variables a and b, as well as to the local variable x.

Outside of the class in which they are defined, static methods and variables can be
used independently of any object. To do so, you need only specify the name of their class
followed by the dot operator. General form is:

classname.method( )

Here, classname is the name of the class in which the static method is declared.

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

Here the **static** function **callme()** defined in the class **StaticDemo** is accessed in another class **StaticByName** by the statement **StaticDemo.callme();**

final keyword

The **final** keyword is used in several different contexts as a modifier meaning that what it modifies cannot be changed in some sense.

final classes

A class declared as **final** will not be subclassed, restricting inheritance. Declaring a class as final implicitly declares all of its methods as final, too. It also provides some benefit in regard to security and thread safety. A common use is for a class that only allows static methods as entry points or static final constants –to be accessed without class instance.

e.g. public final class String

final methods

You can also declare that methods are **final**. A method that is declared final cannot be overridden in a subclass. If you attempt to do so, a compile-time error will result. It is useful when a method assigns a state that should not be changed in the classes that inherit it, and should use it as it is.

eg:**public final String convertCurrency()**

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. Inlining 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.

final variables

Declaring a variable as final prevents its contents from being modified. It can be set once (for instance when the object is constructed, but it cannot be changed after that.) Attempts to change it will generate either a compile-time error or an exception.

eg: final double PI = 3.14;

Subsequent parts of your program can now use PI as if it is a constant, without fear that a value has been changed.

Nested and Inner Classes

It is possible to define a class within another class. Such classes are known as nested classes. The scope of a nested class is bounded by the scope of its enclosing class. For Example class B is defined within class A, then B is known to A, but not outside of A. A nested class has access to the members, including private members, of the class in which it is nested. However, the enclosing class does not have access to the members of the nested class.

There are two types of nested classes: **static and non-static**. A static nested class is one which has the static modifier applied. Because it is static, it must access the members of its enclosing class through an object. That is, it cannot refer to members of its enclosing class directly.

The most important type of nested class is the inner class. An inner class is a **non-static** nested class. It has access to all of the variables and methods of its outer class and may refer to them directly in the same way that other non-static members of the outer class do. Thus, an inner class is fully within the scope of its enclosing class.

Why Use Nested Classes?

There are several compelling reasons for using nested classes, among them:

- It is a way of logically grouping classes that are only used in one place.
- It increases encapsulation.
- Nested classes can lead to more readable and maintainable code.

Logical grouping of classes—If a class is useful to only one other class, then it is logical to embed it in that class and keep the two together. Nesting such "helper classes" makes their package more streamlined.

Increased encapsulation—Consider two top-level classes, A and B, where B needs access to members of A that would otherwise be declared private. By hiding class B within class A, A's members can be declared private and B can access them. In addition, B itself can be hidden from the outside world.

More readable, maintainable code—Nesting small classes within top-level classes places the code closer to where it is used.

The following program illustrates how to define and use an inner class. The class named Outer has one instance variable named outer_x, one instance method named test( ), and defines one inner class called Inner.

```
 class Outer
{
int outer_x = 100;
     void test()
      {
        Inner inner = new Inner();
        inner.display();
     }
      class Inner
      {
          void display()
          {
            System.out.println("display: outer_x = " + outer_x);
              }
       }
 }
 class InnerClassDemo
  {
      public static void main(String args[])
       {
           Outer outer = new Outer();
           outer.test();
         }
  }
```

Output

```
display: outer_x = 100
```

Here the inner class named **Inner** is defined within the scope of class **Outer**. Therefore, any code in class **Inner** can directly access the variable **outer_x**. An instance method named **display( )** is defined inside **Inner**. This method displays

outer_x on the standard output stream. The **main( )** method of **InnerClassDemo** creates an instance of class **Outer** and invokes its **test( )** method. That method creates an instance of class **Inner** and the **display( )** method is called.

Class **Inner** is known only within the scope of class **Outer**. The Java compiler generates an error message if any code outside of class **Outer** attempts to instantiate class **Inner**

Using Command-Line Arguments

Sometimes you will want to pass information into a program when you run it. This is accomplished by passing command-line arguments to **main( ).** A command-line argument is the information that directly follows the program's name on the command line when it is executed. To access the command-line arguments inside a Java program is quite easy—they are stored as strings in the **String** array passed to main( )

```
class CommandLine
{
    public static void main(String args[])
    {
        for(int i=0; i<args.length; i++)
        System.out.println("args[" + i + "]: " +args[i]);
    }
 }
```

Try executing this program, as shown here:

```
java CommandLine This a test program
```

Output will be:

```
args[0]: this
args[1]: is
args[2]: a
args[3]: test
args[4]:program
```

Inheritence

It is one of the most important feature of Object Oriented Programming which is used for reusability purpose. ***Inheritance*** is the capability of a class to use the properties and methods of another class while adding its own functionality. Or in other words inheritance is the mechanism through which we can derive classes from other classes. The derived class is called as **child class or the subclass** or we can say the extended class and the class from which we are deriving the subclass is called the **base class** or the **parent class**. To derive a class in java the keyword **extends** is used. Inheritance allows subclasses to inherit all the variables and methods of their parent classes. The concept of inheritance is used to make the things from general to more specific. The following kinds of inheritance are there in java.

- **Simple Inheritance**
- **Multilevel Inheritance**

Simple Inheritance

When a subclass is derived simply from its parent class then this mechanism is known as simple inheritance. In case of simple inheritance there is only a sub class and its parent class. It is also called single inheritance or one level inheritance.

Multilevel Inheritance

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

The general form of a class declaration that inherits a superclass is shown here:

```
class subclass-name extends superclass-name
 {
// body of class
}
```

You can only specify one **superclass** for any subclass that you create. Java does not support the inheritance of multiple **superclasses** into a single subclass. You can, as stated, create a hierarchy of inheritance in which a subclass becomes a **superclass** of another subclass

```
class A {
int i, j;
void showij() {
System.out.println("i and j: " + i + " " + j);
}
}
class B extends A {
int k;
void showk() {
System.out.println("k: " + k);
}
void sum() {
```

```
System.out.println("i+j+k: " + (i+j+k));
}
}
class SimpleInheritance {
public static void main(String args[]) {
A superOb = new A();
B subOb = new B();
superOb.i = 10;
superOb.j = 20;
System.out.println("Contents of superOb: ");
superOb.showij();
subOb.i = 7;
subOb.j = 8;
subOb.k = 9;
System.out.println("Contents of subOb: ");
subOb.showij();
subOb.showk();
System.out.println();
System.out.println("Sum of i, j and k in subOb:");
subOb.sum();
}
}
```

Output

```
Contents of superOb:
i and j: 10 20
Contents of subOb:
i and j: 7 8
k: 9
Sum of i, j and k in subOb:
i+j+k: 24
```

the subclass **B** includes all of the members of its superclass, **A**. This is why **subOb** can access **i** and **j** and call **showij( )**. Also, inside **sum( )**, **i** and **j** can be referred to directly, as if they were part of **B**.
Even though A is a superclass for B, it is also a completely independent, stand-alone class. Being a superclass for a subclass does not mean that the superclass cannot be used by itself. Further, a subclass can be a superclass for another subclass.

Points to note:

- Although a subclass includes all of the members of its superclass, it cannot access those members of the superclass that have been declared as private. It can be only accessed by other members of the same class.
- A reference variable of a superclass can be assigned a reference to any subclass derived from that superclass. 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. That is, when a reference to a subclass object is assigned to a superclass reference variable, you will have access only to those parts of the object defined by the superclass.

Using super

super has two general forms.

The first calls the superclass' constructor. The second is used to access a member of the superclass that has been hidden by a member of a subclass.

A subclass can call a constructor method defined by its superclass by use of the following form of super:

super(parameter-list);

Here, parameter-list specifies any parameters needed by the constructor in the superclass. **super( )** must always be the first statement executed inside a subclass' constructor.

Eg:
```
class Box
{
      double width;
      double height;
      double depth;
      Box(Box ob)
      {
        width = ob.width;
        height = ob.height;
      }
      Box(double w, double h, double d)
      {
        width = w;
        height = h;
        depth = d;
}
class BoxWeight extends Box
 {
       double weight; // weight of box
   BoxWeight(double w, double h, double d, double m)
     {
       super(w, h, d);
```

```
        weight = m;
      }
  }
```

Here, BoxWeight( ) calls **super( )** with the parameters w, h, and d. This causes the Box( ) constructor to be called, which initializes width, height, and depth using these values. BoxWeight no longer initializes these values itself. It only needs to initialize the value unique to it: weight.

Since constructors can be overloaded, **super( )** can be called using any form defined by the superclass. The constructor executed will be the one that matches the arguments.

The second form of super acts somewhat like this, except that it always refers to the superclass of the subclass in which it is used. This usage has the following general form:

super.member

Here, member can be either a method or an instance variable. This second form of **super** is most applicable to situations in which member names of a subclass hide members by the same name in the superclass.

Eg:
```
class A
     {
          int i;
     }
   class B extends A
   {
      int i;
      B(int a, int b) {
      super.i = a; // i in A
      i = b; // i in B
   }
```

Although the instance variable **i** in **B** hides the **i** in **A**, super allows access to the **i** defined in the superclass. **super** can also be used to call methods that are hidden by a subclass.

Creating a Multilevel Hierarchy

It is possible to build hierarchies that contain as many layers of inheritance as you like. As mentioned, it is perfectly acceptable to use a subclass as a superclass of another. For example, given three classes called A, B, and C, C can be a subclass of B, which is a subclass of A. When this type of situation occurs, each subclass inherits all of the traits found in all of its superclasses. In this case, C inherits all aspects of B and A.

When a class hierarchy is created, constructors are called in order of derivation, from superclass to subclass. For example, given a subclass called B and a superclass called A. Here first A's constructor is called. After that only we are calling B's constructor. Further, since super() must be the first statement executed in a subclass' constructor, this order is the same whether or not super( ) is used. If super( ) is not used, then the default or parameter less constructor of each superclass will be executed.

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

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

Method Overriding

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. 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.

```
class A {
int i, j;
A(int a, int b) {
i = a;
j = b;
}
// display i and j
void show() {
System.out.println("i and j: " + i + " " + j);
}
}
class B extends A {
int k;
B(int a, int b, int c) {
super(a, b);
k = c;
}
// display k – this overrides show() in A
void show() {
System.out.println("k: " + k);
}
}
class Override {
public static void main(String args[]) {
B subOb = new B(1, 2, 3);
subOb.show(); // this calls show() in B
}
}
```

The output produced by this program is shown here:

```
k: 3
```

When show( ) is invoked on an object of type B, the version of show( ) defined within B is used. That is, the version of show( ) inside B overrides the version declared in A.

If you wish to access the superclass version of an overridden function, you can do so by using **super.**For example In B if we are using super.show() it calls the superclass version of show( ).

Method overriding occurs only when the names and the type signatures of the two methods are identical. If they are not, then the two methods are simply overloaded. It allows a
general class to specify methods that will be common to all of its derivatives, while allowing subclasses to define the specific implementation of some or all of those methods. Overridden methods are another way that Java implements the "one interface, multiple methods" aspect of polymorphism

Dynamic Method Dispatch

In Java, runtime polymorphism or dynamic method dispatch is the mechanism by which a call to an overridden method is resolved at run time, rather than compile time. Method overriding forms the basis for Dynamic method dispatch. In this process, an overridden method is called through the reference variable of a superclass. The determination of the method to be called is based on the object being referred to by the reference variable. Therefore, if a superclass contains a method that is overridden by a subclass, then when different types of objects are referred to through a superclass reference variable, different versions of the method are executed. For example:

```
Eg:  class A
      {
      void callme()
      {
         System.out.println("Inside A's callme method");
      }
   }
   class B extends A
   {
   void callme()
     {
         System.out.println("Inside B's callme method");
     }
    }
   class C extends A
    {
      void callme()
     {
        System.out.println("Inside C's callme method");
     }
   }
   class Dispatch
   {
```

```
public static void main(String args[])
{
   A a = new A(); // object of type A
     B b = new B(); // object of type B
     C c = new C(); // object of type C
   A r; // obtain a reference of type A
   r = a; // r refers to an A object
   r.callme(); // calls A's version of callme
     r = b; // r refers to a B object
     r.callme(); // calls B's version of callme
   r = c; // r refers to a C object
   r.callme(); // calls C's version of callme
}
    }
```

Output

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

This program creates one superclass called A and two subclasses of it, called B and C. Subclasses B and C override callme( ) declared in A. Inside the main( ) method, objects of type A, B, and C are declared. Also, a reference of type A, called r, is declared. The program then assigns a reference to each type of object to r and uses that reference to invoke callme( ). As the output shows, the version of callme( ) executed is determined by the type of object being referred to at the time of the call.

Dynamic, run-time polymorphism is one of the most powerful mechanisms that object-oriented design brings to bear on code reuse and robustness. The ability of existing code libraries to call methods on instances of new classes without recompiling while maintaining a clean abstract interface is a profoundly powerful tool.

Using Abstract Classes

There are situations in which you will want to define a superclass that declares the structure of a given abstraction without providing a complete implementation of every method. 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

Java provides a special type of class called an abstract class, which helps us to organize our classes based on common methods. An abstract class lets you put the common method names in one abstract class without having to write the actual implementation code.
An abstract class can be extended into sub-classes, these sub-classes usually provide

implementations for all of the abstract methods. To declare an abstract method, use this general form:

abstract type name(parameter-list);

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. Any subclass of an abstract class must either implement all of the abstract methods in the superclass, or be itself declared abstract.

Eg:
```
abstract class A
 {
    abstract void callme();
   void callmetoo()
    {
     System.out.println("This is a concrete method.");
    }
 }
 class B extends A
 {
    void callme()
     {
        System.out.println("B's implementation of callme.");
      }
 }
 class AbstractDemo
 {
     public static void main(String args[])
    {
        B b = new B();
        b.callme();
        b.callmetoo();
     }
          }
```

Here no objects of class **A** are declared in the program. As mentioned, it is not possible to instantiate an abstract class. One other point: class **A** implements a concrete method called **callmetoo( )**. This is perfectly acceptable. Abstract classes can include as much implementation as they see fit. Although abstract classes cannot be used to instantiate objects, they can be used to create object references, because Java's approach to run-time polymorphism is implemented through the use of superclass references. Thus, it must be possible to create a reference to an abstract class so that it can be used to point to a subclass object.

Packages and interfaces

Packages are Java's way of grouping a variety of classes and/or interfaces together. The grouping is done according to functionality. Packages act as containers for classes It is a concept similar to class libraries in other languages Packages are organized in hierarchical manner. By organizing our classes into packages we can achieve the following benefits.

- Classes contained in the packages of other programs can be reused
- Two classes in two different packages can have same name. They should be referred by their fully qualified name, comprising the package name and class name.
- Packages provide a way to hide classes thus preventing other programs or packages from accessing classes that are meant for internal use only.

In java packages can be classified into two.

- **Built-in Packages**
- **User Defined Packages**

Built-in Packages

These are packages which are already available in java language. These packages provide interfaces and classes, and then fields, constructors, and methods to be used in program. The following are the packages provided by Java(System packages).

java.lang

lang stands for language. This package contains primary classes and interfaces essential for developing a basic java program. It consists of wrapper classes which are useful to convert primary data types into objects. There are classes like String, StringBuffer to handle strings. There is a Thread class to create individual processes. Runtime and system classes are also present in this package which contain methods to execute an application and find the total memory and free memory available in JVM. This is the only package that is automatically
imported into every Java program.

java.io

Stands for input and output**.** A stream represents flow of data from one place to another place. Streams are useful to store data in the form of files and also to perform input-output related tasks

java.util

Package that contains miscellaneous utility classes like Stack, LinkedList, Hashtable, Vector, Arrays etc. These classes are called collections There are also classes for handling time, date operations

java.net

Stands for network. Client Server programing can be done using this package. It provides classes for network support, including URLs, TCP sockets, UDP sockets, IP addresses, and a binary-to-text converter.

java.awt

Stands for abstract window toolkit.Package that provides an integrated set of classes to manage user interface components such as windows, dialog boxes, buttons, checkboxes, lists, menus, scrollbars, and text fields. (AWT = Abstract Window Toolkit)

java.applet

Package that enables the creation of applets through the Applet class. It also provides several interfaces that connect an applet to its document and to resources for playing audio.

java.text

This package has two Important classes, DateFormat to format dates and times, and NumberFormat which is useful to format numeric values.

java.sql

This package helps to connect to databases like Oracle or Sybase, retrieve data from them and use it in a Java Program.

User Defined Packages

Just like built-in packages users of java can create their own packages. They are called user defined packages.

Defining a Package

To create a package is quite easy: simply include a package command as the first statement in a Java source file. Any classes declared within that file will belong to the specified package. The package statement defines a name space in which classes are stored. If 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 pkg1;
```

Here **pkg1** is the name of the package. This must be the first statement in a java source file.

For example

```
package firstPackage;
```

```
public class FirstClass
{
 ……………….
 ………………..
 }
```

Here package name is firstPackage.The class FirstClass is now considered a part of this package. This would be saved as a file called FirstClass.java and located in the directory named firstPackage. On compiling java will create a class file and store it in the same directory.

Remember that the class file must be located in a directory that has the same name as package and this directory should be a sub directory of the directory where classes that will import the packages are located.

Here the program using the package firstPackage should be executed from a directory immediately above firstPackage. Or else **CLASSPATH** must be set to include the path to firstPackage. The first alternative is the easiest (and doesn't require a change to CLASSPATH), but the second alternative lets your program find firstPackage no matter what directory the program is in.

Also case is significant, and the directory name must match the package name exactly. More than one file can include the same package statement. The package statement simply specifies to which package the classes defined in a file belong.

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. The general form of a multileveled package statement is shown here:

package pkg1[.pkg2[.pkg3]];

This approach allows us to group related classes into packages and then group related packages into a larger package.A package hierarchy must be reflected in the file system of your Java development system. For example, a package declared as

package java.awt.image;

needs to be stored in java\awt\image in your file system.

Access Protection

Classes and packages are both means of encapsulating and containing the name space and scope of variables and methods. Packages act as containers for classes and other subordinate packages. Classes act as containers for data and code. The class is Java's smallest unit of abstraction. Because of the interplay between classes and packages, Java addresses four categories of visibility for class members in packages

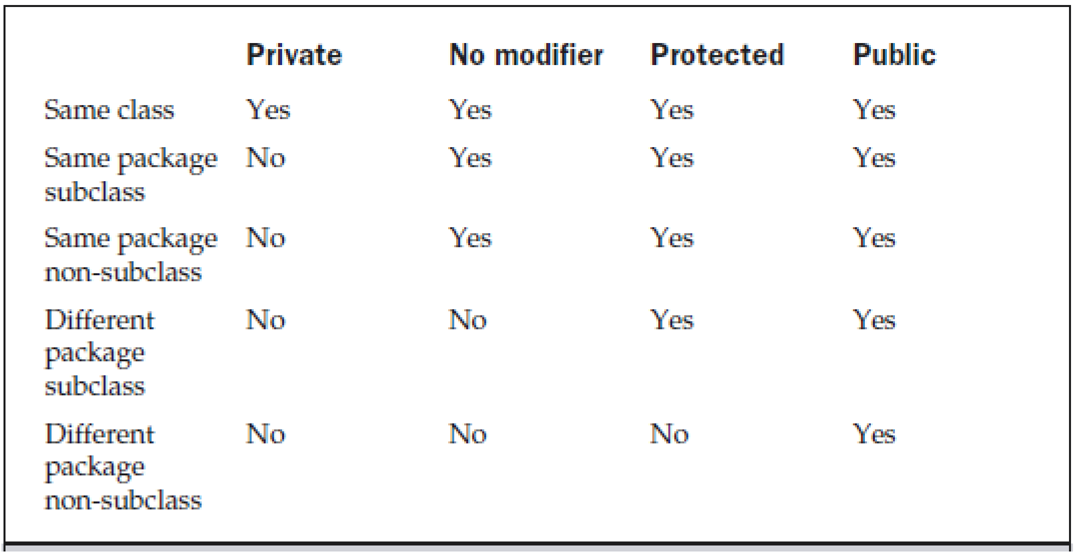
- Subclasses in the same package

- Non-subclasses in the same package
- Subclasses in different packages
- Classes that are neither in the same package nor subclasses

The three access specifiers, **private**, **public**, and **protected**, provide a variety of ways to produce the many levels of access required by these categories.

Anything declared public can be accessed from anywhere. Anything declared private cannot be seen outside of its class. When a member does not have an explicit access specification, it is visible to subclasses as well as to other classes in the same package. This is the default access. If you want to allow an element to be seen outside your current package, but only to classes that subclass your class directly, then declare that element protected.

The following table shows the access rules applicable to members of classes. A class has only two possible access levels: default and public. When a class is declared as public, it is accessible by any other code. If a class has default access, then it can only be accessed by other code within its same package.



	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

The following example shows all combinations of the access control modifiers. This example has two packages and five classes. Remember that the classes for the two different packages need to be stored in directories named after their respective packages—in this case, p1 and p2.

The source for the first package defines three classes: Protection, Derived, and SamePackage. The first class defines four int variables in each of the legal protection modes. The variable n is declared with the default protection, n_pri is private, n_pro is protected, and n_pub is public.

The second class, Derived, is a subclass of Protection in the same package, p1. This grants Derived access to every variable in Protection except for n_pri, the private one.The third class, SamePackage, is not a subclass of Protection, but is in the same package and also has access to all but n_pri.

```
package p1;
public class Protection {
int n = 1;
private int n_pri = 2;
protected int n_pro = 3;
public int n_pub = 4;
public Protection() {
System.out.println("base constructor");
System.out.println("n = " + n);
System.out.println("n_pri = " + n_pri);
System.out.println("n_pro = " + n_pro);
System.out.println("n_pub = " + n_pub);
}
}
This is file Derived.java:
package p1;
class Derived extends Protection {
Derived() {
System.out.println("derived constructor");
System.out.println("n = " + n);
// class only
// System.out.println("n_pri = " + n_pri);
System.out.println("n_pro = " + n_pro);
System.out.println("n_pub = " + n_pub);
}
}
This is file SamePackage.java:
package p1;
class SamePackage {
SamePackage() {
Protection p = new Protection();
System.out.println("same package constructor");
System.out.println("n = " + p.n);
// class only
// System.out.println("n_pri = " + p.n_pri);
System.out.println("n_pro = " + p.n_pro);
System.out.println("n_pub = " + p.n_pub);
}
}
```

Following is the source code for the other package, p2. The two classes defined in p2 cover the other two conditions which are affected by access control. The first class, Protection2, is a subclass of p1.Protection. This grants access to all of p1.Protection'svariables except for n_pri (because it is private) and n, the variable declared with the default protection. Remember, the default only allows access from within the class or the package, not extra-package subclasses. Finally, the class OtherPackage has access to only one variable, n_pub, which was declared public.
This is file Protection2.java:

```
package p2;
class Protection2 extends p1.Protection {
System.out.println("derived other package constructor");
// class or package only
// System.out.println("n = " + n);
// class only
// System.out.println("n_pri = " + n_pri);
 System.out.println("n_pro = " + n_pro);
 System.out.println("n_pub = " + n_pub);
}
}
```

This is file OtherPackage.java:

```
package p2;
class OtherPackage {
OtherPackage() {
p1.Protection p = new p1.Protection();
System.out.println("other package constructor");
// class or package only
// System.out.println("n = " + p.n);
// class only
// System.out.println("n_pri = " + p.n_pri);
// class, subclass or package only
// System.out.println("n_pro = " + p.n_pro);
System.out.println("n_pub = " + p.n_pub);
}
}
```

If you wish to try these two packages, here are two test files you can use. The one for package p1 is shown here:

```
// Demo package p1.
package p1;
// Instantiate the various classes in p1.
public class Demo {
public static void main(String args[]) {
Protection ob1 = new Protection();
Derived ob2 = new Derived();
SamePackage ob3 = new SamePackage();
}
}
```

The test file for p2 is shown next:

```
// Demo package p2.
package p2;
// Instantiate the various classes in p2.
public class Demo {
public static void main(String args[]) {
Protection2 ob1 = new Protection2();
OtherPackage ob2 = new OtherPackage();
}
}
```

AccessingPackages

To use a class contained in a package, you can use one of two mechanisms:

- If the class you want to use is in some other package, you can refer to that class by its full name, including any package names (for example, java.awt.Font).
- For classes that you use frequently from other packages, you can import individual classes or a whole package of classes. After a class or a package has been imported, you can refer to that class by its class name.

Full Package and Class Names

To refer to a class in some other package, you can use its full name: the class name preceded by any package names. You do not have to import the class or the package to use it this way:

```
java.awt.Font f = new java.awt.Font()
```

For classes that you use only once or twice in your program, using the full name makes the most sense.

Importing Packages

To import classes from a package, use the import command.. import statements occur immediately following the package statement (if it exists) and before any class definitions. This is the general form of the **import statement:**

```
import pkg1[.pkg2].(classname|*);
```

Here, pkg1 is the name of a top-level package, and pkg2 is the name of a subordinate package inside the outer package separated by a dot (.). There is no practical limit on the depth of a package hierarchy, except that imposed by the file system. Finally, you specify either an explicit classname or a star (*), which indicates that the Java compiler should import the entire
package.This code fragment shows both forms in use:

```
import java.util.Date;
import java.io.*;
```

All of the standard Java classes included with Java are stored in a package called java. The basic language functions are stored in a package inside of the java package called **java.lang**. Normally, you have to import every package or class that you want to use, but since Java is useless without much of the functionality in **java.lang**, it is implicitly imported by the compiler for all programs. This is equivalent to the following line being at the top of all of your programs:

```
import java.lang.*;
```

If a class with the same name exists in two different packages that you import using the star form, the compiler will remain silent, unless you try to use one of the classes. In that case, you will get a compile-time error and have to explicitly name the class specifying its package.

When a package is imported, only those items within the package declared as public will be available to non-subclasses in the importing code.

Using a Package

Consider some simple programs that will use classes from other packages. The following is a program which shows a package named package1 containing a single *class ClassA*

```
package package1;
public class ClassA
{
  public void displayA()
  {
     System.out.println("Class A");
   }
}
```

This source file should be named ClassA.java and stored in subdirectory package1.The resultant ClassA.class will be stored in the same sub directory.

```
import package1.ClassA;
Class PackageTest1
{
  public static void main(String args[])
  {
     ClassA objectA=new ClassA();
     objectA.display()
   }
}
```

The above program imports class ClassA from package package1.The source file should be saved as PackageTest1.java and then compiled. The source file should be saved as PackageTest1.java and compiled. The source file and compiled file should be saved in a directory of which package1 is a subdirectory.

During compilation of PackageTest1.java the compiler checks for file ClassA.class in package1 directory for information it needs, but it doesnot include the code from ClassA.class in the file PackageTest1.class.When PackageTest1 program is run, java looks for the file PackageTest1.class and loads it. Now the code from ClassA.class is also loaded.

When you are importing multiple packages in your program ,it is likely that two or more packages contains classes with identical names.

```
package p1;
public class Teacher
{……..}
public class Student
{……….}
package p2;
public class Courses
{………..}
public class Student
{………}
```

When we are importing and using this packages like this

```
import p1;
import p2;
Student student1;
```

Since both the packages contain the class **Student** compiler cannot understand which one to use and generates an error. In such cases we need to explicitly specify which one we intend to use.

For example

import p1;
import p2;
p1.Student student1;
p2.Student student2;

Hiding Classes

When we import a package using * all public classes are imported. Some times we may prefer not to import certain classes. That is we may like to hide these classes from accessing from outside the package. Such classes should be declared not **public .** Example:

```
        package p1;
        public class X
        {
         / /body of X
        }
        class Y
         {
           / /body of Y
          }
```

Here the class Y which is declared not public is hidden from outside of package p1.This class can be seen and used only by other classes in same package. A java source file contains only one public class and any number of non-public classes.

Now consider the following code which imports package p1 that contains classes X and Y.

```
import p1.*;
X object X;
Y objectY;
```

The above code generates error message because the class Y which is not public is not imported and therefore not available for creating its objects.

Interfaces

Interface is a kind of class. In a class we have methods and variables defined. Interfaces also have methods and variables defined, but with a difference. The difference is that an interface has only abstract methods and final fields. None of the methods in an interface has its code defined. So an interface can be defined as a specification of prototypes. Since we are writing abstract methods in interface, it is possible to provide different implementations of abstract methods depending upon the requirement of objects. Although they are similar to abstract classes, interfaces have an additional capability. A class can implement more than one interface. By contrast, a class can only inherit a single superclass. It allows classes, regardless of their locations in the class hierarchy, to implement common behaviors

Since an interface has only abstract methods it is not possible to create objects of interface. So we can create separate classes where we can implement all methods of the interface. These classes are called implementation classes. The flexibility of interface lies in the fact that every implementation class can have its own implementation of abstract methods of interface. By providing the interface keyword, Java allows you to fully utilize the "one interface, multiple methods" aspect of polymorphism..

Why do we use Interfaces?

Reveals the functionality of the object without revealing its implementation– This is the concept of encapsulation

Since the implementation happens in The implementation can change without affecting the caller of the interface

– The caller does not need the implementation at the compile time

● It needs only the interface at the compile time

● During runtime, actual object instance is associated

To have unrelated classes implement similar methods (behaviors).Interface can be implemented by two completely unrelated classes. The implementation classes need not be subclasses,

To model multiple inheritance

A class can implement multiple interfaces while it can extend only one class

Defining an Interface

An interface is defined much like a class. This is the general form of an interface:

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

access is either **public** or not used. When no access specifier is included, then default access results, and the interface is only available to other members of the package in which it is declared. When it is declared as public, the interface can be used by any other code. name is the name of the interface, and can be any valid identifier. Notice that the methods which are declared have no bodies. They end with a semicolon after the parameter list. They are, essentially, abstract methods; there can be no default implementation of any method specified within an interface. Each class that includes an interface must implement all of the methods.Variables can be declared inside of interface declarations. They are implicitly final and static, meaning they cannot be changed by the implementing class. They must also be initialized with a constant value. All methods and variables are implicitly public if the interface, itself, is declared as public.

Here is an example of an interface definition.

```
Interface Item
 {
      static final int code=1001;
      static final String name=”Fan”;
      void display();
 }
```

Note that the code for the method is not included in the interface and the method declaration simply ends with a semicolon. The class that implements this interface must define the code for the method.

Implementing Interfaces

Once an interface has been defined, one or more classes can implement that interface. To implement an interface, include the **implements** clause in a class definition, and then create the methods defined by the interface. The general form of a class that includes the **implements** clause looks like this:

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

This shows that a class can extend another class while implementing interfaces. Here, access is either **public** or not used. If a class implements more than one interface, the interfaces are separated with a comma. If a class implements two interfaces that declare the same method, then the same method will be used by clients of either interface. The methods that implement an interface must be declared public. Also, the type signature of the implementing method must match exactly the type signature specified in the interface definition.

After your class implements an interface, subclasses of your class will inherit those new methods (and can override or overload them) just as if your superclass had actually defined them. If your class inherits from a superclass that implements a given interface, you don't have to include the implements keyword in your own class definition.

Let's examine one simple Example

```
Interface Area
{
   final static float pi=3.14f;
   float compute(float x,float y);
}
class Rectangle implements Area
{
   public float compute(float x, float y)
    {
      return (x*y);
    }

   void Display()
  {
    System.out.println("Inside Rectangle");
  }
  }
class Circle implements Area
{
   public float compute(float x,float y)
   {
     return(pi*x*x);
    }
}
class InterfaceTest
{
   public static void main(String args[])
    {
       Rectangle rect=new Rectangle();
        Circle cir=new Circle();
        Area area;
        area=rect;
        System.out,println("Area of Rectangle="+area.compute(10,20));
        area=cir;
        System.out,println("Area of Circle="+area.compute(10,0));

     }
}
```

Output is

```
Area of Rectangle=200
Area of Circle=314
```

Notice that compute ( ) is declared using the **public** access specifier.

Accessing Implementations Through Interface References

It is possible to declare reference variables of an interface. Any instance of any class that implements the declared interface can be referred to by such a variable. When you call a method through one of these references, 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, allowing classes to be created later than the code which calls methods on them.

In the above example variable area is declared to be of the interface type Area, yet it was assigned instance of Rectangle and then Circle. Although area can be used to access the compute( )method, it cannot access any other members of the Rectangle and Circle Classes An interface reference variable only has knowledge of the methods declared by its interface.

Partial Implementations

If a class includes an interface but does not fully implement the methods defined by that interface, then that class must be declared as abstract. For example:

```
interface Callback
 {
   void callback(int param);
 }
abstract class Incomplete implements Callback
{
   int a, b;
   void show()
  {
     System.out.println(a + " " + b);
  }
    …………………
}
```

Here, the class **Incomplete** does not implement **callback( )** and must be declared as abstract. Any class that inherits Incomplete must implement **callback( )** or be declared abstract itself.

Accessing Interface Variables

You can use interfaces to import shared constants into multiple classes by simply declaring an interface that contains variables which are initialized to the desired values This is similar to header files in C++ to contain a large number of constants. Since such interfaces do not contain methods, there is not need to worry about implementing any methods. The constant values will be available to any class that implements the interface. The values can be used in any method, as part of any variable declaration, or anywhere where we can use a final value.

```
interface A
{
   int m=10;
   int n=50;
}
class B implements A
{
   int x=m;
   void methodB(int size)
    {
      ……………..
      ………………
      if(size<n)
     }
}
```

Interfaces Can Be Extended

One interface can inherit another by use of the keyword extends. The syntax is the same as for inheriting classes. When a class implements an interface that inherits another interface, it must provide implementations for all methods defined within the interface inheritance chain.

```
interface A {
void meth1();
void meth2();
}
// B now includes meth1() and meth2() -- it adds meth3().
interface B extends A {
void meth3();
}
```

```
// This class must implement all of A and B
class MyClass implements B {
public void meth1() {
System.out.println("Implement meth1().");
}
public void meth2() {
System.out.println("Implement meth2().");
}
public void meth3() {
System.out.println("Implement meth3().");
}
}
class IFExtend {
public static void main(String arg[]) {
MyClass ob = new MyClass();
ob.meth1();
ob.meth2();
ob.meth3();
}
}
```

On removing the implementation for **meth1(**) in **MyClass**, we will get a compile-time error. As stated earlier, any class that implements an interface must implement all methods defined by that interface, including any that are inherited from other interfaces.

Interfaces in a Package

It is also possible to write interfaces in a package. But whenever we create an interface the implementation classes also should be created. We cannot create an object to the interface, but we can create objects for implementation classes and use them. Consider the following Example.

```
package mypack;
public interface MyDate
{
  void showDate();
}
package mypack;
import mypack.MyDate;
import java.util.*;
public class DateImpl implements MyDate
{
    public void ShowDate()
    {
      Date d=new Date();
      System.out.println(d);
   }
}
import mypack.DateImpl;
class DateDisplay
{
   public static void main(String args[])
    {
      MyDate obj=new DateImpl();
      obj.showDate();
     }
}
```

Create a directory with name **mypack** and place the file **MyDate.java** in that directory. Compile the preceding code. MyDate.class will be created in mypack, which denotes the byte code of interface. The class DateImp which Implements the function **showDate()** is also defined in the same package. On compiling the corresponding class file is created in **mypack**.
We have created the class **DateDisplay** which uses the function **showDate()** defined in the implementation class **DateImp** and displays the system date and time.

Multiple Inheritance using Interfaces

In multiple inheritance, sub classes are derived from multiple super classes. If two super classes have same names for members then which member is inherited in the sub class is the main confusion in multiple inheritance. This is the reason java does not support the concept of multiple inheritance. This confusion can be reduced by using multiple interfaces to achieve multiple inheritance.

```
interface A
{
  int x=20;
  void method();
}
interface B
{
  int x=30;
  void method();
}
```

Create a class MyClass as the implementation class for these two interfeces.

```
class Myclass implements A,B
```

Here to refer to interface A's X we can write:
A.X
And to refer to interface B's X we can write:
B.X
Similarly there will not be any confusion regarding which method is available to the implementation class, since both methods in the interface do not have body and body is provided in implementation class, Myclass.
Example:

```
 interface Father
{
     float HT=6.2f;
     void height();
}
interface Mother
{
     float HT=6.2f;
     void height();
 }
class Child implements Father,Mother
{
    public void height()
    {
         float ht=(Father.HT+Mother.HT)/2;
          System.out.println("Child's height="+ht);
     }
  }
  Class Multi
   {
      public static void main(String args[])
         {
             Child ch=new Child();
             ch.height();
          }
   }
```