Unit 4
Chapter 7
Multiple Inheritance - Interfaces

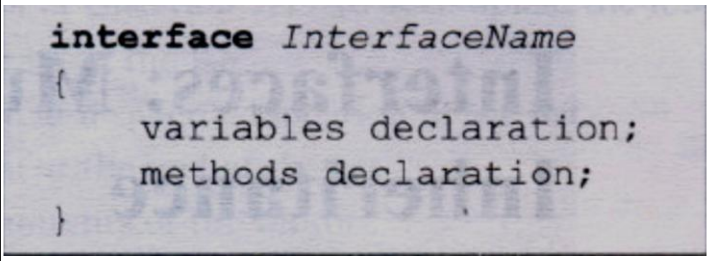
Java does not support multiple inheritance. That is, classes in Java cannot have more than one superclass. For instance, a definition like

```
class A extends B extends C
{
--------
----------
}
```

A large number of real-life applications require the use of multiple inheritance whereby we inherit methods and properties from several distinct classes. Java provides an alternate approach known as *interfaces* to support the concept of multiple inheritance) Although a Java class cannot be a subclass of more than one superclass, it can *implement* more than one interface, thereby enabling us to create classes that build upon other classes without the problems created by multiple inheritance.

7.1 Defining interfaces

An interface is basically a kind of class. Like classes, interfaces contains methods and variables but with a major difference. The difference is that interfaces define only abstract methods and final fields. This means that interfaces do not specify any code to implement these methods and data fields contain only constant
Therefore, it is the responsibility of the class that implements an interface to define the code for implementation of these methods. The syntax for defining an interface is very similar to that for defining a class. The general form of an interface definition is



```
interface InterfaceName
{
    variables declaration;
    methods declaration;
}
```

Here, interface is the key word and *InterfaceName*is any valid Java variable just like class names). Variables are declared as follows

static final *type VariableName= Value;*
Note that all variables are declared as constants. Methods declaration will contain only a list of methods without any body statements. Example
return-type*methodNamel*(parameter_list);

Here is an example of an interface definition that contains two variables and one method

```
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. Another example of an interface is:

```
interface Area
{
    final static float pi = 3.142F;
    float compute (float x, float y);
    void show ( );
}
```

7.1.1 Extending interfaces

Like classes, interfaces can also be extended. That is, an interface can be sub interfaced from other interfaces. The new sub interface will inherit all the members of the super interface in the manner similar to subclasses. This is achieved using the keyword extends as shown below:

```
interface name2 extends name1
{
    body of name2
}
```

For example, we can put all the constants in one interface and the methods in the other. This will enable us to use the constants in classes where the methods are not required.

```
interface  ItemConstants
{
    int code = 1001;
    string name = "Fan";
}
interface Item extends ItemConstants
{
    void display ( );
}
```

```
interface ItemConstants
{
   int code = 1001;
   String name = "Fan";
}
interface ItemMethods
{
   void display( );
}
interface Item extends ItemConstants, ItemMethods
{
   ..........
   ..........
}
```

While interfaces are allowed to extend to other interfaces, subinterfaces cannot define the methods declared in the superinterfaces. After all, subinterfaces are still interfaces, not classes. Instead, it is the responsibility of any class that implements the derived interface to define all the methods. Note that when an interface extends two or more interfaces, they are separated by commas. It is important to remember that an interface cannot extend classes. This would violate the rule that an interface can have only abstract methods andconstants.

7.1.2 Implementing interfaces

Interfaces are used as "superclasses" whose properties are inherited by classes. It is therefore necessary to create a class that inherits the given interface. This is done as follows

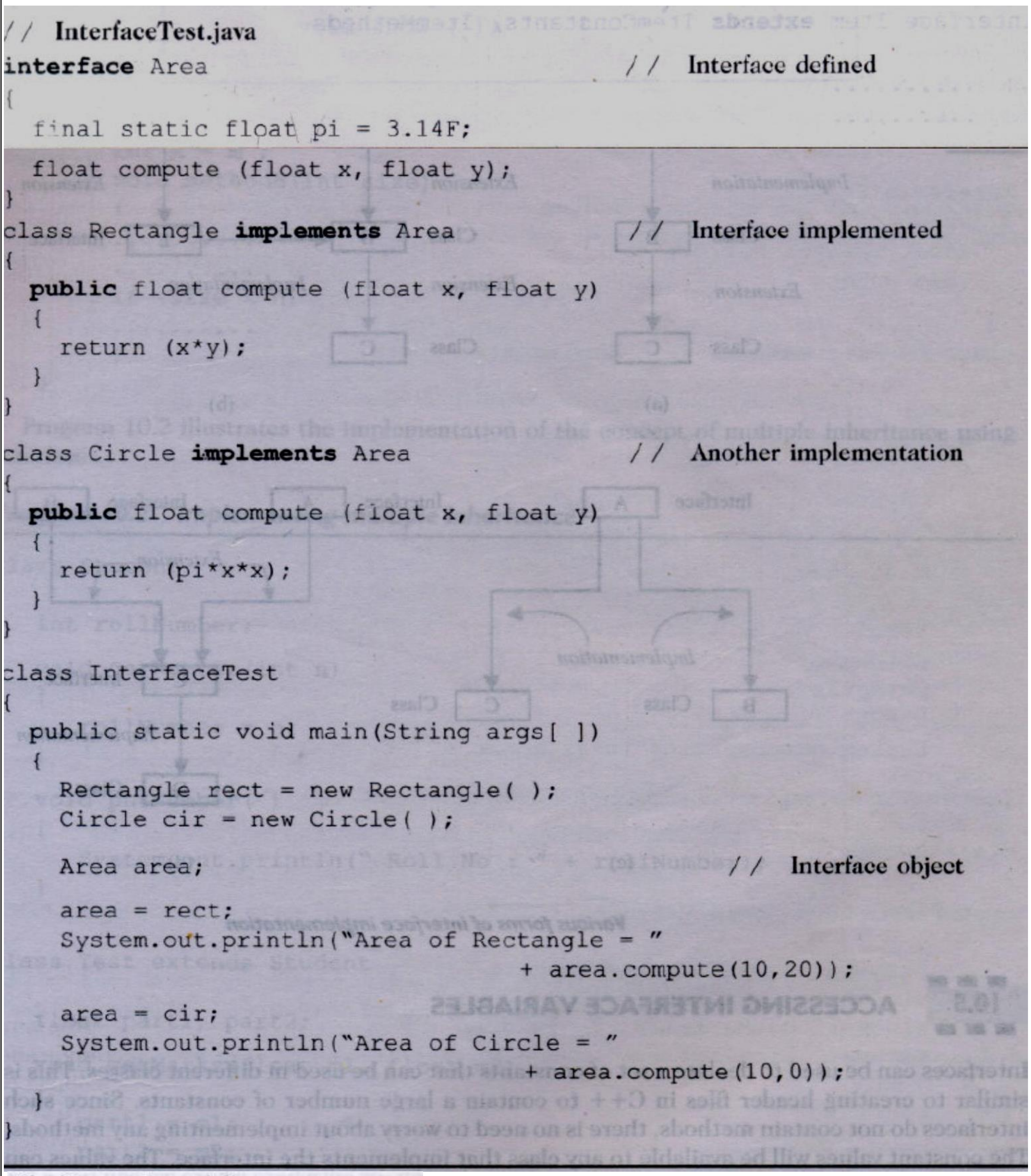
```
   class classname implements Interfacename
   {
        body of classname
   }
```

Here the class *classname*"implements" the interface *interfacename. A* more general form of implementation may look like this:

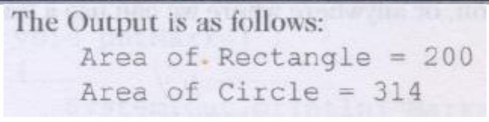
```
  class classname  extends superclass
        implements interface1,interface2, ......
  {
        body of classname
  }
```

This shows that a class can extend another class while implementing interfaces. When a class implements more than one interface, they are separated by a comma.
In this program, first we create an interface Area and implement the same in tw6 different classes, Rectangle and
Circle. We create an instance of each class using the new operator. Then we declare an object of type Area, the interface class. Now, we assign the reference to the Rectangle object rect to

area. When we call the compute method of area, the compute method of Rectangle class is invoked. We repeat the same thing with the Circle object.



```
// InterfaceTest.java
interface Area                                  // Interface defined
{
  final static float pi = 3.14F;
  float compute (float x, float y);
}
class Rectangle implements Area                 // Interface implemented
{
  public float compute (float x, float y)
  {
    return (x*y);
  }
}
class Circle implements Area                    // Another implementation
{
  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;                                     // Interface object
    area = rect;
    System.out.println("Area of Rectangle = "
                                    + area.compute(10,20));
    area = cir;
    System.out.println("Area of Circle = "
                                    + area.compute(10,0));
  }
}
```



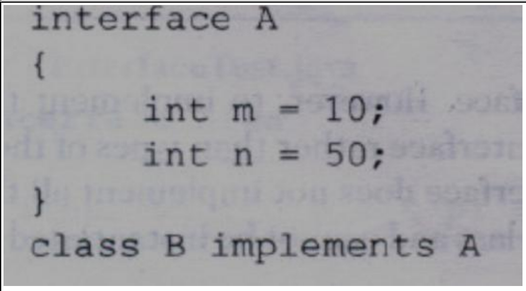
The Output is as follows:

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

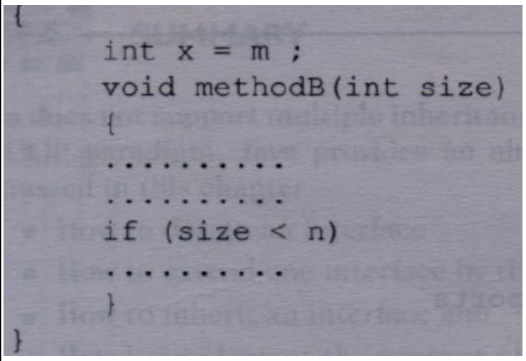
7.1.3 Accessing interface variables

Interfaces can be used to declare a set of constants that can be used in different classes. Interfaces do not contain methods, there is no 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 we can use afinal value.

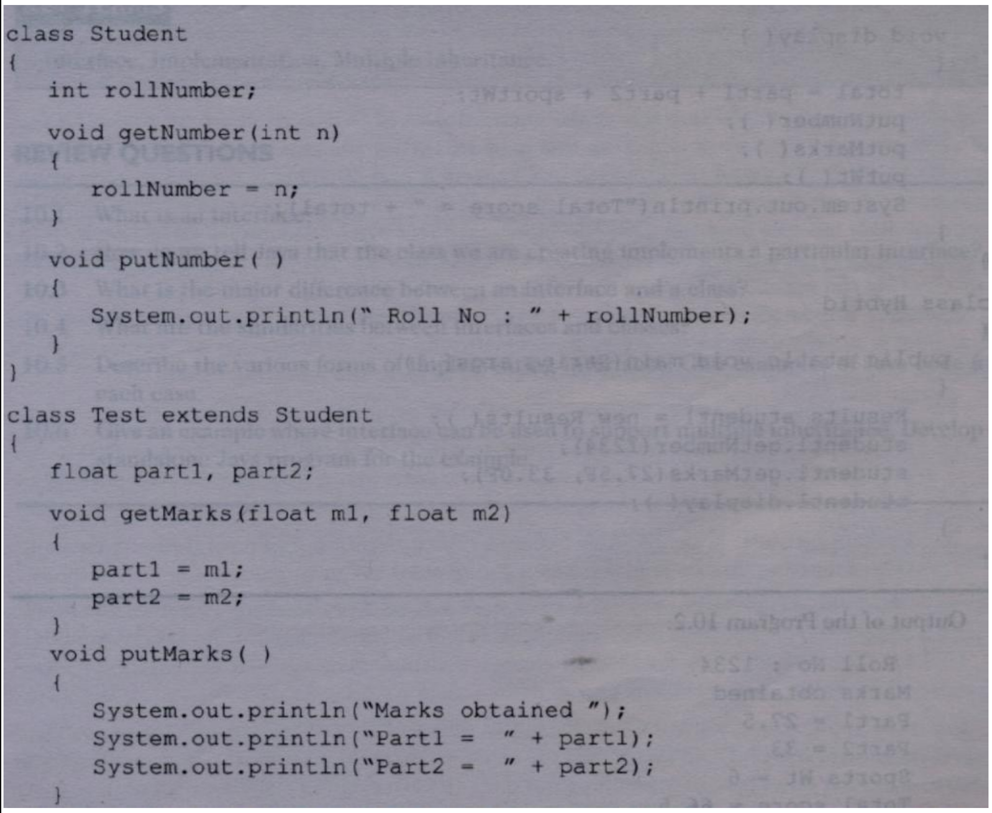
Example:



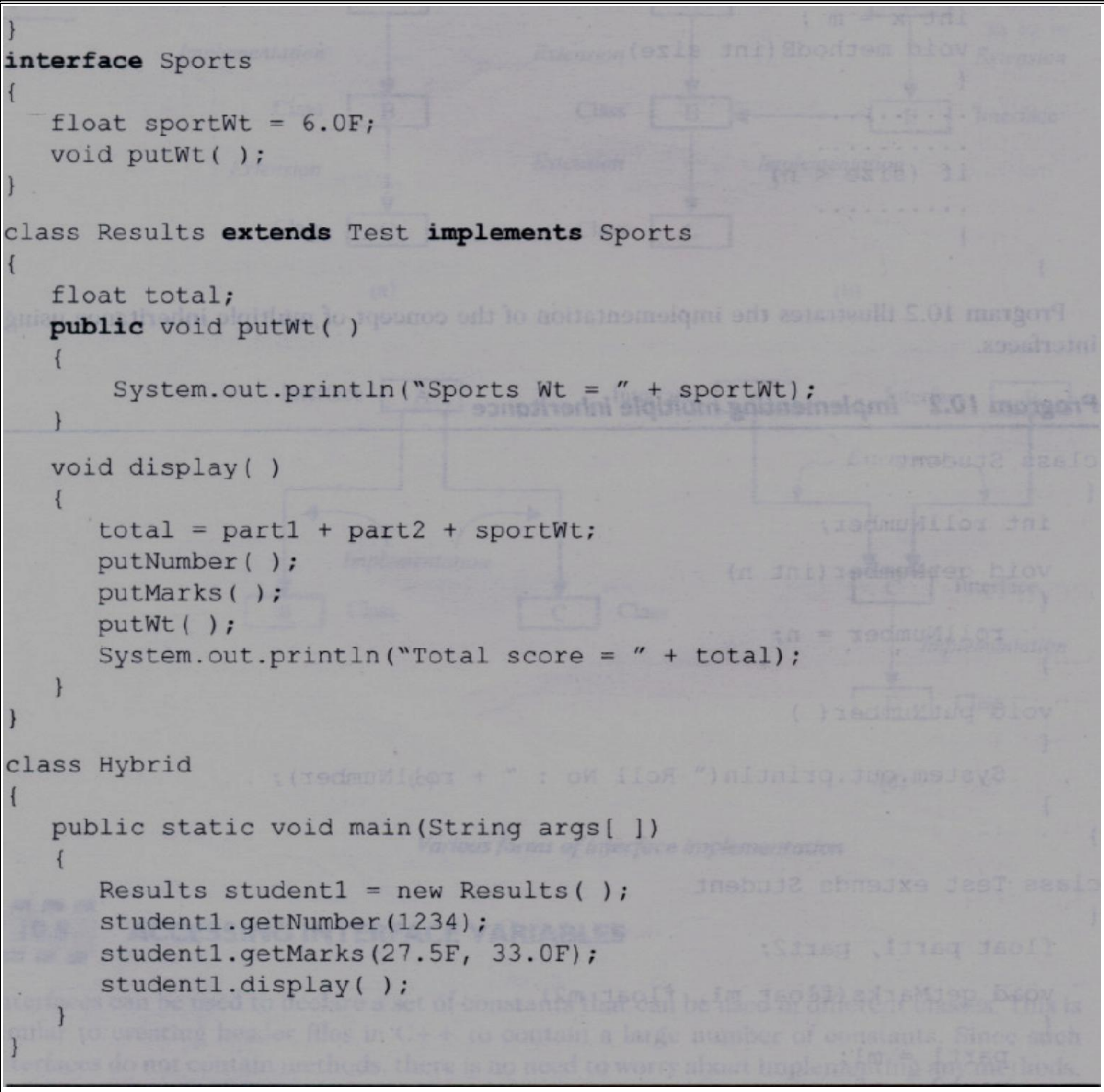
```
interface A
{
    int m = 10;
    int n = 50;
}
class B implements A
```



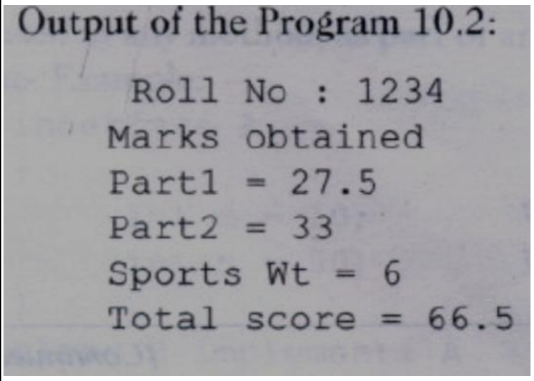
```
{
    int x = m ;
    void methodB(int size)
    {
    ..........
    ..........
    if (size < n)
    ...........
    }
}
```



```
class Student
{
   int rollNumber;
   void getNumber(int n)
   {
      rollNumber = n;
   }
   void putNumber( )
   {
      System.out.println(" Roll No : " + rollNumber);
   }
}
class Test extends Student
{
   float part1, part2;
   void getMarks(float m1, float m2)
   {
      part1 = m1;
      part2 = m2;
   }
   void putMarks( )
   {
      System.out.println("Marks obtained ");
      System.out.println("Part1 =  " + part1);
      System.out.println("Part2 =  " + part2);
   }
```



```
}
interface Sports
{
   float sportWt = 6.0F;
   void putWt( );
}
class Results extends Test implements Sports
{
   float total;
   public void putWt( )
   {
       System.out.println("Sports Wt = " + sportWt);
   }
   void display( )
   {
      total = part1 + part2 + sportWt;
      putNumber( );
      putMarks( );
      putWt( );
      System.out.println("Total score = " + total);
   }
}
class Hybrid
{
   public static void main(String args[ ])
   {
      Results student1 = new Results( );
      student1.getNumber(1234);
      student1.getMarks(27.5F, 33.0F);
      student1.display( );
   }
}
```



Output of the Program 10.2:

```
     Roll No : 1234
    Marks obtained
    Part1 = 27.5
    Part2 = 33
    Sports Wt = 6
    Total score = 66.5
```

Chapter-8
Packages: Putting Classes Together

8.1 Introduction

One of the main features of OOP is its ability to reuse the code already created. One way of achieving this is by extending the classes and implementing the interfaces. This is limited to reusing the classes within a program. If we need to use classes from other programs we have to use packages. Packages are Java's way of grouping a variety of classes and/or interfaces together. The grouping is usually done according to functionality. In fact, packages act as "containers" for classes. By organizing our classes into packages we achieve the following benefits:

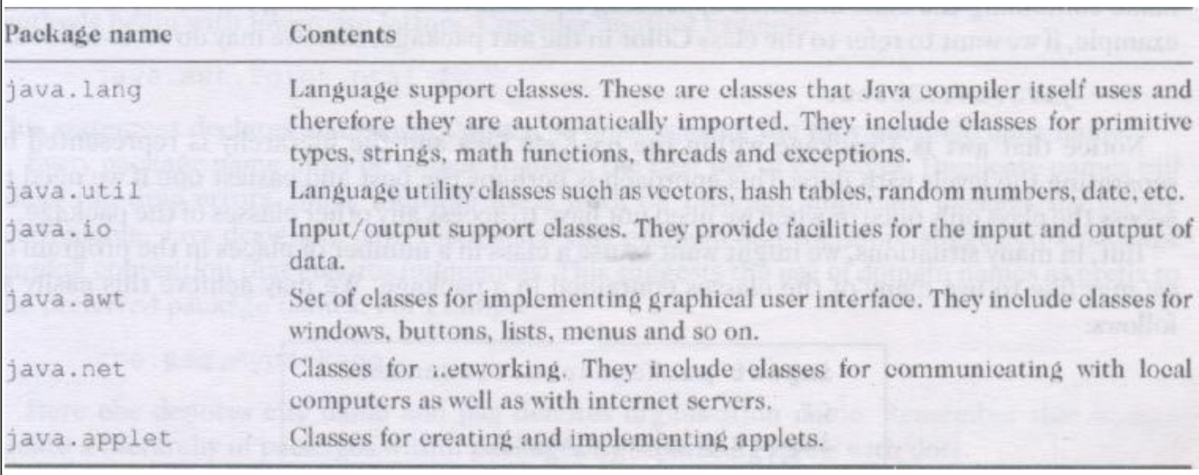
1. The classes contained in the packages of other programs can be easilyreused.
2. In packages, classes can be unique compared with classes in other packages. That is, two classes in two different packages can have the same name. They may be referred by their fully qualified name, comprising the package name and the classname.
3. Packages provide a way to "hide" classes thus preventing other programs or packages from accessing classes that are meant for internal useonly.
4. Packages also provide a way for separating "design" from "coding". First we can design classes and decide their relationships, and then we can implement the Java code needed for the methods. It is possible to change the implementation of any method without affecting the rest of thedesign.

For most applications, we will need to use two different sets of classes, one for the internal representation of our program's data, and the other for external presentation purposes. We may have to build our own classes for handling our data and use existing class libraries for designing user interfaces. Java packages are therefore classified into two types. The first category is known as *Java API packages* and the second is known as *user definedpackages.*

8.2 Java API packages

Java API provides a large number of classes grouped into different packages according to functionality.

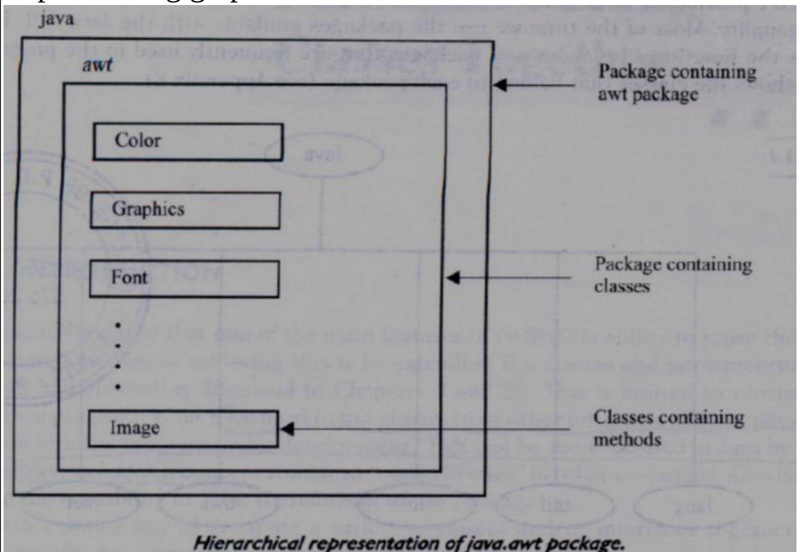
Java System Packages and Their Classes



Package name	Contents
`java.lang`	Language support classes. These are classes that Java compiler itself uses and therefore they are automatically imported. They include classes for primitive types, strings, math functions, threads and exceptions.
`java.util`	Language utility classes such as vectors, hash tables, random numbers, date, etc.
`java.io`	Input/output support classes. They provide facilities for the input and output of data.
`java.awt`	Set of classes for implementing graphical user interface. They include classes for windows, buttons, lists, menus and so on.
`java.net`	Classes for ıetworking. They include classes for communicating with local computers as well as with internet servers.
`java.applet`	Classes for creating and implementing applets.

Using system packages

The packages are organised in a hierarchical structure. This shows that the package named java contains the package awt, which in turn contains various classes required for implementing graphical user interface.


java
awt
Color
Graphics
Font
Image
Package containing awt package
Package containing classes
Classes containing methods

Hierarchical representation of java.awt package.

There are two ways of accessing the classes stored in a package. The first approach is to use *the fully qualified class name* of the class that we want to use. This is done by using the package name containing the class and then appending the class name to it using the dot operator. For example, if we want to refer to the class Color in the awt package, then we may do so as follows:
java.awt.Colour

Notice that awt is a package within the package java and the hierarchy is represented by separating the levels with dots. This approach is perhaps the best and easiest one if we need to access the class only once or when we need not have to access any other classes of the package. But, in many situations, we might want to use a class in a number of places in the program or we may like to use many of the classes contained in a package. We may achieve this easily asfollows:
import package name. class name;
or
importpackagename.*;

These are known as *import statements* and must appear at the top of the file, before any class declarations. The first statement allows the specified class in the specified package to be imported. For example, the statement

importjava.awt.Color;

imports the class Colour and therefore the class name can now be directly used in the program. There is no need to use the package name to qualify the class. The second statement imports every class contained in the specified package. For example, the statement
importjava.awt.*; will bring all classes of java.awt package.

8.3 Namingconventions

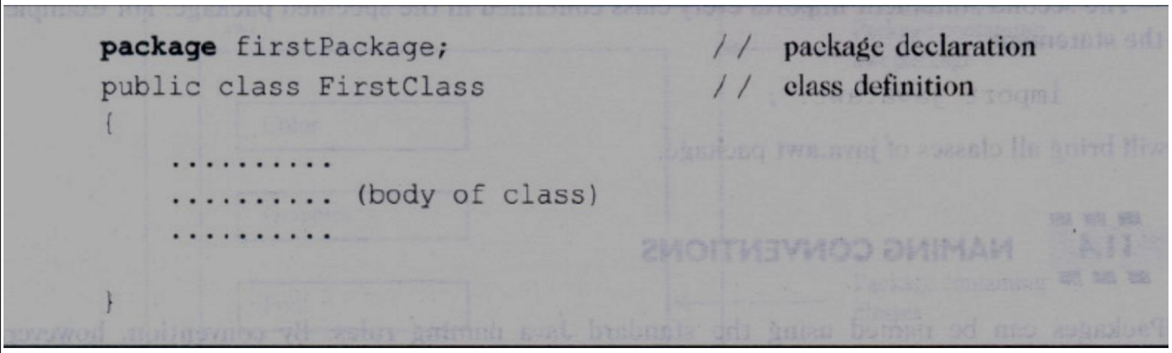
Packages can be named using the standard Java naming rules. By convention, however, packages begin with lowercase letters. This makes it easy for users to distinguish package names from class names when looking at an explicit reference to a class. We know that all class names, again by convention, begin with an uppercase letter. For example, look at the following statement:

double y= iava.lang.Math.sqrr(x);
This statement uses a fully qualified class name Math to invoke the method sqrt ().
Every package name must be unique to make the best use of packages. Duplicate names will cause run-time errors. Since multiple users work on Internet, duplicate package names are unavoidable. Java designers have recognised this problem and therefore suggested a package naming convention that ensures uniqueness. This suggests the use of domain names as prefix to the predefined package names.
For example:
cbe.psg.mypackage

Here cbe denotes city name and psg denotes organisation name. We can create a hierarchy of packages within packages by separating levels with dot.

8.4 Creatingpackages

In order to create a package we must first declare the name of the package using the package keyword followed by a package name. This must be the first statement in a Java source file (except for comments and white spaces). Then we define a class, just as we normally define a class. Here is an example:

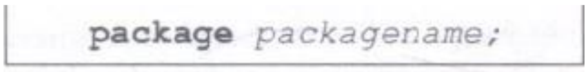


```
package firstPackage;              // package declaration
public class FirstClass            // class definition
{
    ..........
    .......... (body of class)
    ..........

}
```

Here the package name is firstPackage. The class FirstClass is now considered a part of this package. This listing would be saved as a file called FirstClass.java, and located in a directory named firstPackage. When the source file is compiled, Java will create a .class file and store it in the samedirectory.

Remember that the .class files must be located in a directory that has the same name as the package, and this directory should be a subdirectory of the directory where classes that will import the package are located.
To recap, creating our own package involves the following steps:
1. Declare the package at the beginning of a file using theform



```
package packagename;
```

2. Define the class that is to be put in the package and declare itpublic.
3. Create a subdirectory under the directory where the main source files arestored.
4. Store the listing as the classname.java file in the subdirectorycreated.
5. Compile the file. This creates .class file in thesubdirectory.

Remember that case is significant and therefore the subdirectory name must match the package name exactly Java also supports the concept of package hierarchy. This is done by specifying multiple names in a package statement, separated by dots.

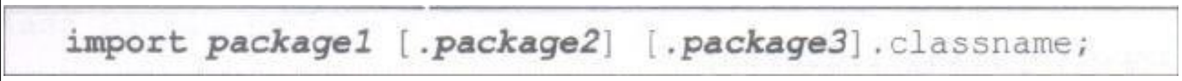
packagefirstPackage.secondPackage;

This approach allows us to group related classes into a package and then group related packages into a larger package. Remember to store this package in a subdirectory named firstPackage\secondPackage.

A Java package file can have more than one class definitions. In such cases, only one of the classes may be declared public and that class name with .java extension is the source file name. When a source file with more than one class definition is compiled, Java creates independent .class files for those classes.

8.5 Accessing aPackage

The import statement can be used to search a list of packages for a particular class. The general form of
import statement for searching a class is as follows:



```
import package1 [.package2] [.package3].classname;
```

Here *packagel*is the name of the top level package,*package2* is the name of the package that is inside the *packagel*, and so on. We can have any number of packages in a package hierarchy. Finally, the explicit *classname*is specified.
Note that the statement must end with a semicolon (;).The import statement should appear before any class definitions in a source file. Multiple import statements are allowed. The following is an example of importing a particular class:

importfirstPackage.secondPackage.MyClass;

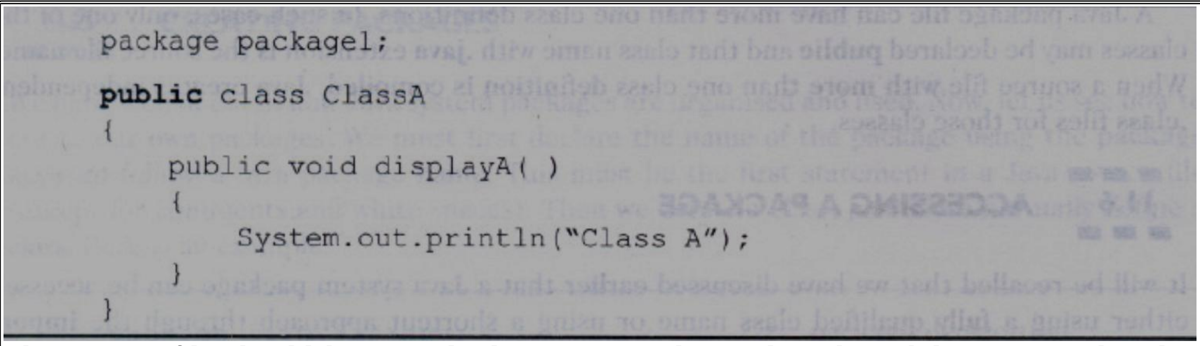
After defining this statement, all the members of the class MyClass can be directly accessed using the class name or its objects (as the case may be) directly without using the package name. We can also use another approach as follows:

import*packagename.*;*

Here, *packagename*may denote a single package or a hierarchy of packages . The star (*) indicates that the compiler should search this entire package hierarchy where it encounters a class name. This implies that we can access all classes contained in the above package directly. The major drawback of the shortcut approach is that it is difficult to determine from which package a particular member came. This is particularly true when a large number of packages are imported. But the advantage is that we need not have to use long package names repeatedly in the program.

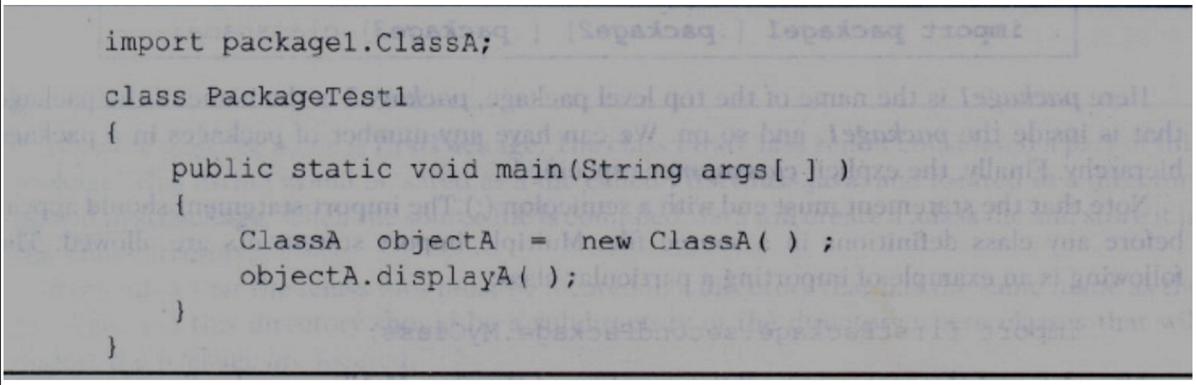
Using a package

The listing below 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 the subdirectory packagel as stated earlier. Now compile this java file. The resultant ClassA.class will be stored in the samesubdirectory.
Now consider the listing shown below:



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

This listing shows a simple program that imports the class ClassA from the package packagel. The source file should be saved as PackageTestl.java and then compiled. The source file and the compiled file would be saved in the directory of which package 1 was a subdirectory. Now we can run the program and obtain theresults.
During the compilation of PackageTestl.java the compiler checks for the file ClassA.class in the packagel directory for information it needs, but it does not actually include the code from ClassA.class in the file PackageTestl.class. When the PackageTestl program is run, Java looks for the file PackageTestl.class and loads it using something called *class loader*. Now the interpreter knows that it also needs the code in the file ClassA.class and loads it aswell.

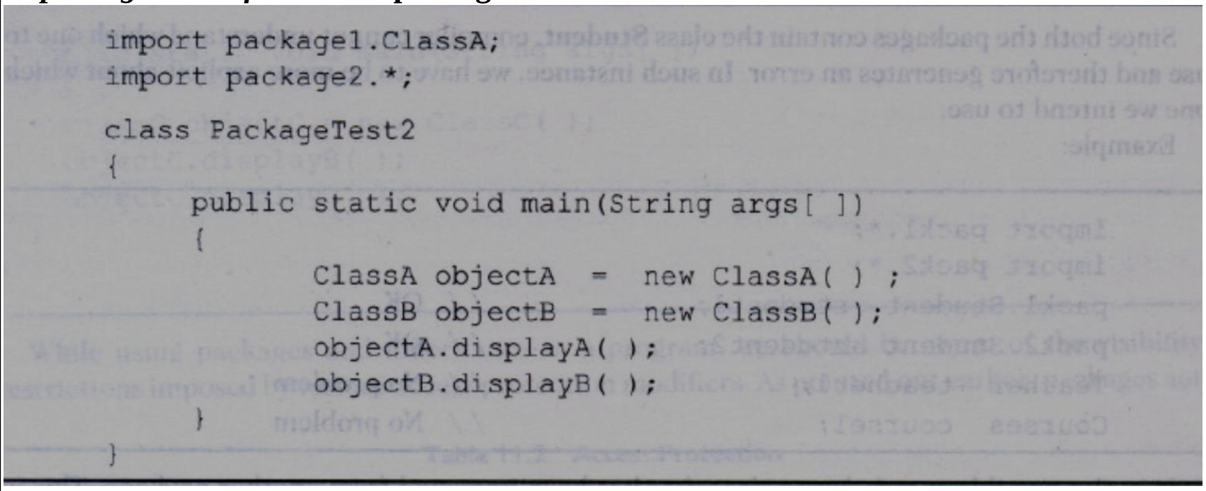
Now let us consider another package named package2 containing again a single class as shown below:

```
package package2;
public class ClassB
{
protectedint m = 10
public void displayB( )
{
  System.out.println("Class Bn);
System.out.println("m = " + m);
}

}
```

The source file and the compiled file of this package are located in the subdirectory package2.

***Importing* classes *from other* packages**



```
import package1.ClassA;
import package2.*;

class PackageTest2
{
        public static void main(String args[ ])
        {
                ClassA objectA  =  new ClassA( ) ;
                ClassB objectB  =  new ClassB( );
                objectA.displayA( );
                objectB.displayB( );
        }
}
```

This program may be saved as PackageTest2.java, compiled and run to obtain the results. The output will be as under

```
Class A
Class B
m = 10
```

When we import multiple packages it is likely that two or more packages contain classes with identical names. While using packages and inheritance in a program, we should be aware of the visibility restrictions imposed by various access protection modifiers.

8.6 Adding a class to apackage

The package p1 contains one public class by name A. Suppose we want to add another class B to this package. This can be done asfollows:

1. Define the class and make itpublic.
2. Place the package statement

```
package p1;
```

before the class definition as follows:

```
package p1;
public class B
 {
   / / body of B
 }
```

3. Store this as B.java file under the directoryp1.
4. Compile B.java file. This will create a B.class file and place it in the directorypl.

Note that we can also add a non-public class to a package using the same procedure. Now, the package pI will contain both the classes A and B. A statementlike

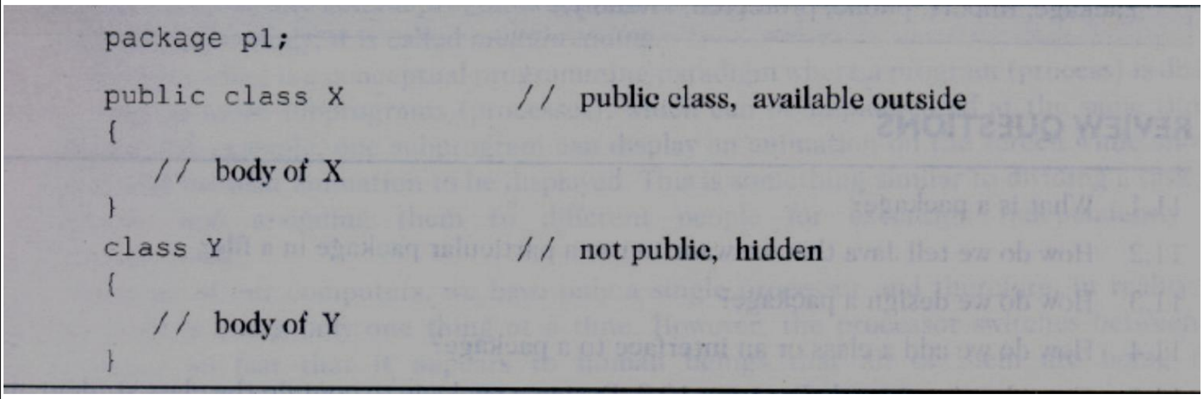
import pl.*;

will import both of them. Remember that, since a Java source file can have only one class declared as public, we cannot put two or more public classes together in a .java file. This is because of the restriction that the file name should be same as the name of the public class with .java extension. If we want to create a package with multiple public classes in it, we may follow the following steps:

1. Decide the name of thepackage.
2. Create a subdirectory with this name under the directory where main source files are stored.
3. Create classes that are to be placed in the package in separate source files and declare the packagestatement
    packagepackagename;
at the top of each source file.
4. Switch to the subdirectory created earlier and compile each source file. When completed, the package would contain .class files of all the sourcefiles.

8.7 Hidingclasses

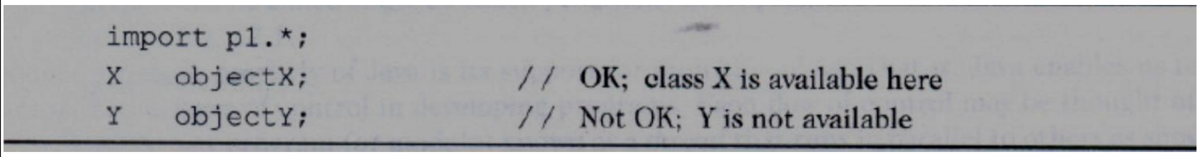
When we import a package using asterisk (*), all public classes are imported. However, we may prefer to "not import" certain classes. That is, we may like to hide these classes from accessing from outside of the package. Such classes should be declared "not public". Example:



```
package p1;

public class X              // public class, available outside
{
    // body of X
}
class Y                     // not public, hidden
{
    // body of Y
}
```

Here, the class Y which is not declared public is hidden from outside of the package p1. This class can be seen and used only by other classes in the same package. Note that a Java source file should contain only one public class and may include any number of non-public classes. We may also add a single non-public class using the procedure suggested in the previous section. Now, consider the following code, which imports the package p1 that contains classes X andY:



```
import p1.*;
X   objectX;                // OK; class X is available here
Y   objectY;                // Not OK; Y is not available
```

Java compiler would generate an error message for this code because the class Y, which has not been, declared public, is not imported and therefore not available for creating its objects.