Chapter 3 – Packages and Interfaces

Marks Allotted: 16
Lectures: 06

Semester Pattern:

Exam:	Winter 2008	Summer 2009
Marks:	16	12

Yearly Pattern (JPR-IF 1526):

Exam:	S'04	W'04	S'05	W'05	S'06	W'06	S'07	W'08
Marks:	--	20	--	16	08	16	--	--

Syllabus Contents:

3.1 Interface: Multiple Inheritance

Defining interfaces, Extending interfaces, Implementing interfaces, Accessing Interface variable

3.2 Packages: Putting Classes Together. System Package, Using system Package, Naming Convention, Creating Package, Accessing a package, Using a package, adding a class to a package

Package

In the preceding chapters we have introduced the term ‘package’ many times. Now we will learn the package in deep in this chapter.

With Interface, Package is Java’s most innovative feature. Package is Java’s way of grouping the classes and interfaces together. That is, a package is called as the collection of classes and interfaces. It can be called as the container for classes that is used to keep the class name space compartmentalized. Packages are stored in a hierarchical manner and are explicitly imported into new class definitions. By organizing the classes and interfaces into packages we can achieve the following goals:

1. Classes contained in packages of other programs can be easily reused.
2. Two classes in two different packages can have same name but they should be referred by their fully qualified name.
3. Packages provide a way of hiding classes. So we can prevent them from accessing outside.
4. Packages provide a way of separating ‘design’ from ‘coding’. We can design classes and decide their relationships and then we can implement the Java code needed for the methods. It is also possible to change the implementation of any method without affecting the rest of the design.

In general, a Java source file can contain any (or all) of the following four internal parts:

- A single package statement (optional)
- Any number of import statements (optional)
- A single public class declaration (required)
- Any number of classes private to the package (optional)

Only one of these, ‘the single public class declaration’ — has been used in the examples so far. This chapter will explore the remaining parts.

The Java API packages

Java API (Application Program Interface) library provides a large number of classes grouped into different packages according to functionality. Many times we use the packages available with the Java API. For example, while using the println( ) method we have to use the java.lang package which is by default imported in a Java program. Fig. below shows the fundamental packages that are frequently used in a Java program.

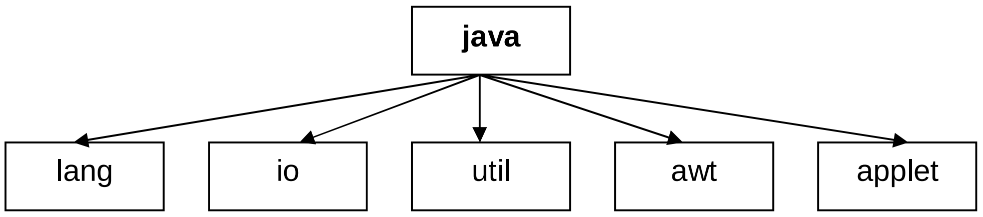

Fig. Frequently used Java packages

The classes that are included in these packages are:
java.lang – Language support classes such as System, Thread, Exception etc.
java.util – Utility classes such as Vector, Arrays, LinkedList, Stack etc.
java.io – Input output support classes such as BufferedReader, InputStream
java.awt – Classes using GUI such as Window, Frame, Panel etc.
java.applet – Classes for creating and implementing applets.

Using System packages

The packages are organized into the hierarchical structure as shown in fig. below:

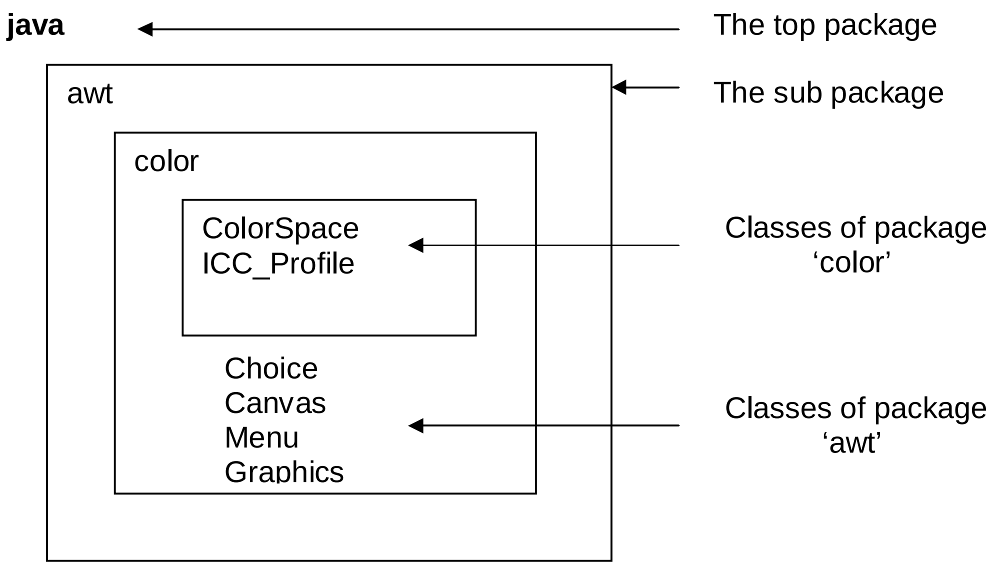

Fig. Java class hierarchy representation

By viewing the fig. above we can come to know about the class hierarchy used in Java. There can be any sub-packages created. All packages contain their own classes and classes contain their own methods. There are two way of accessing the classes from packages. The first is to use the fully qualified name of the class that we want to use in the program. This is done by using a package name containing the class and then appending the class name to it using dot operator. For example, if we want to refer the Vector class defined in java.util package we have to use this name:

```
java.util.Vector
```

Note that, util is the package within the package java and the hierarchy is represented by separating the levels with dots. Take another example from fig. 4.2 above. If we want to access the class ColorSpace then we can use,

```
java.awt.color.ColorSpace
```

This is called as filly qualified name of the class. This tells that where the class is actually stored. This approach is perhaps the best and easiest one if we need to access the class only once or we need not have to access any other classes of the same package. That is, if you want to create an object of the class Vector we do so by,

```
java.util.Vector v = new java.util.Vector();
```

Here the object 'v' of type Vector is created. 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 classes contained in the package. We can do it by following statement:

```
import packagename.classname;
```

or

```
import packagename.*;
```

These are called as import statements and must appear at the top of the source file before any class declaration. The first statement allows the specified class in the specified package to be imported in the source file. We have already done this by importing Scanner and Vector class from java.util package as,

```
import java.util.Scanner;
```

Here only class Scanner will be available for use in the program. So there is no need to use fully qualified name of the class while using in statements.

The second statement imports every class of the package in the source program. So, all these are classes available in the program, without fully qualified name. For example,

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

This will import all the classes from ‘java.io’ package into the program. But it is recommended to use the first form in order to increase the execution speed of the program.

Like the top package java, we have other packages too but they are not the part of core Java such as com, javax, launcher, org, sunw etc.

Naming conventions for packages

Package names can be given using standard Java naming conventions rules. By this convention, the package name begins with lower case letter. This make easy to distinguish between the class or interface and package name by looking at them. Because by naming conventions, the first letter of a class name is always upper case letter.

For example look at the statement:

```
double x = java.lang.Math.sqrt(10);
```

This statement uses the fully qualified name of the class Math to use method sqrt( ). Note that the method name begins with a lower case letter.

Every package name must be unique to make the best use of packages. Duplicate names will cause the run-time errors. Since multiple users work on the Internet, duplicate package names are unavoidable. Java developers have recognized this problem and thus suggested a package naming convention to ensure its uniqueness. For example:

```
soc.col.mypackage
```

Here ‘soc’ denotes society name and ‘col’ denotes the college name. We can create the hierarchy of packages within packages by separating the levels by dot.

Creating the package

Creating a package is quite easy. We just have to include a package command as the first statement in a Java source file. Then any classes declared within that file will automatically belong to the specified package. The package statement defines a name space in which classes are stored. If we omit the package statement, the class names are put into the default package, which has no name. (This is why we haven’t had to worry about packages before now.) Most of the time, we will define a package for our code. This is the general form of the package statement:

```
package pkg;
```

Here, ‘pkg’ is the name of the package. For example, the following statement creates a package called ‘MyPackage’.

```
package myPackage;
public class MyClass
{
     //body of the class
}
```

Here myPackage is the name of package. The class MyClass is now considered as the part of this package. Java uses file system directories to store packages. For example, the .class files for any classes we declare to be part of myPackage must be stored in a directory called myPackage. Remember that case is significant, and the directory name must match the package name exactly. More than one file can include the same package statement. The package statement simply specifies to which package the classes defined in a file belong. It does not exclude other classes in other files from being part of that same package.

We can create a hierarchy of packages also. For this, we have to separate each package name from the one above it by use of a dot operator. The general form of a multileveled package statement is shown here:

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

A package hierarchy must be reflected in the file system of our Java development system. For example, a package declared as,

```
package java.awt.image;
```

needs to be stored in java\awt\image directory. We must be sure to choose our package names carefully. We cannot rename a package without renaming the directory in which the classes are stored.

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

Finding packages and CLASSPATH

By default, the Java run-time system uses the current working directory as its starting point. Thus, if our package is in the current directory, or a subdirectory of the current directory, it will be found by Java run-time system. Also, we can specify a directory path or paths by setting the CLASSPATH environmental variable. For example, consider the following package declaration.

```
package MyPack;
```

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

```
// A simple package
package myPack;

class Nation
{
      String name;
      long pop;

      Nation(String n, long p)
      {
         name = n;
         pop = p;
      }
      void display()
      {
         System.out.println("Nation:"+name);
         System.out.println("Population:"+pop);
      }
}
class SimplePack
{
    public static void main(String args[])
    {
       Nation asia[] = new Nation[3];

       asia[0] = new Nation("Korea", 56233314);
       asia[1] = new Nation("Iran",  35264124);
       asia[2] = new Nation("Singapore", 2356660);

       for(int i=0; i<3; i++)
           asia[i].display();
    }
}
```

Simple Package example

Now, save this file with name SimplePack.java, and put it in a directory called myPack. Next, compile the file. This can be compiled by following command:

```
javac myPack\SimplePack.java
```

After this the resulting .class file will get stored also in the myPack directory. Then try executing the SimplePack.class, using the following command line:

```
java myPack.SimplePack
```

We will get the output,

```
Nation:Korea
Population:56233314
Nation:Iran
Population:35264124
Nation:Singapore
Population:2356660
```

As explained, SimplePack is now part of the package myPack. This means that it cannot be executed by itself. That is, we cannot use this command line:

```
java SimplePack
```

SimplePack must be qualified with its package name. If we do so, an exception (NoClassDefFoundError) will occur.

Access protection

In the preceding chapters, we learned about various aspects of Java's access control mechanism and its access modifiers. We have already seen the access modifiers of Java. Now after introduction of packages it will be simpler to learn all four accesses.

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 while classes act as containers for data and code. The class is Java's smallest unit of abstraction. Because of the intercommunication between classes and packages, Java addresses four categories of visibility for class members:

- 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 specified with any access specification, it is visible to subclasses as well as to other classes in the same package. This is called as the default access or package access. If we want to allow an element to be seen outside our current package, but only to classes that subclass your class directly, then declare that element protected.

	private	**default**	**protected**	**public**
Same class	Yes	Yes	Yes	Yes
Same package subclass	No	Yes	Yes	Yes
Same package no subclass	No	Yes	Yes	Yes
Different package subclass	No	**No**	Yes	Yes
Different package no subclass	No	No	No	Yes

Table: Class member access

Table above shows access protection for member of the 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.

Program below illustrates the concept of default access with implementation of packages.

```
//Protection.java
package first;
  public class Protection
  {
        int n = 5;
        private int n_pri = 10;
        protected int n_pro = 43;
        public int n_pub = 92;

        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);
        }
  }
  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);
        }
  }
  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);
        }
  }
```

Program: a package access example

This file is **Protection.java**

```
//OtherPackage.java
package second;
  class Protection2 extends first.Protection
  {
        Protection2()
        {
           System.out.println("Keen constructor");

           //  class or package only
           //System.out.println("n = " + n);//statement1

           //  class only
           //  System.out.println("n_pri = " + n_pri);

           System.out.println("n_pro = " + n_pro);
           System.out.println("n_pub = " + n_pub);
        }
  }
  public class OtherPackage
  {
        OtherPackage()
        {
            first.Protection p = new first.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);
        }
  }
```

Program: a package access example

This file is **OtherPackage.java**

Program above illustrates the concept of package access i.e. default access. First file named, Protection.java contains three classes that is Protection, Derived and SamePackage. They are stored in the package ‘first’. Second file contains two classes Protection2 and OtherPackage. Both are the part of package ‘second’. Here class Protection2 is inherited from Protection class from first package. Now remember that the members declared as public and protected from ‘Protection’ are accessible in class ‘Protection2’. But the default access members can not be used in class ‘Protection2’. They are commented. Default access members are available in the same package only. They act as public in their own package. As variable n is available in all the classes from ‘first’ but it is not available in ‘second’.

In order to compile the program, just remove the comments from output statement1 from second program, we will get the error.

```
second\OtherPackage.java:10: n is not public in
first.Protection; cannot be accessed from outside package
```

Importing/accessing the packages

There is no core Java classes in the unnamed default package. All of the standard classes are stored in some named packages of Java’s library. Since classes within packages must be fully qualified with their package name or names. It could become tedious to type in the long dot-separated package path name for every class that we want to use in program. For this reason, Java includes the import statement to bring certain classes, or entire packages, into visibility. Once imported, a class can be referred to directly, using only its name. The import statement is a convenience to the programmer and is not technically needed to write a complete Java program. If we are going to refer to a few dozen classes in our application, however, the import statement will save a lot of typing.

In a Java source file, 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][.pkg3].classname;
```

or

```
import pkg1[.pkg2][.pkg3].*;
```

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 (.). The set of square brackets indicate the optional part of import statement. There is no practical limit on the depth of a package hierarchy, except that imposed by Java's library packages. At end, we specify either an explicit classname or a asterisk(*). In the first form of import, only the respective classname and its method will be imported. And the second form imports the entire package. So, all classes from that package are available in the program. For example:

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

The asterisk form may increase compilation time, especially when if we import several large packages such as 'awt'. For this reason it is a good habit to explicitly name the classes that we want to use rather than importing whole packages. However, the asterisk form has absolutely no effect on the run-time performance or size of our classes.

Remember user defined packages can also be imported by this such as,

```
import first.Protection;
```

Here, 'first' is the name of package and 'Protection' is the class name. 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, we have to import every package or class that we 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 Java programs. So, there will be no effect if we include following line at the start of our program.

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

If a class with the same name exists in two different packages that we import using the asterisk form, the compiler will do nothing, unless we try to use one of the classes. In that case, we will get a compile-time error and have to explicitly name the class specifying its package. Any place we use a class name, we can use its fully qualified name, which includes its full package hierarchy.

For example, we have created a class named 'Vector' in our package 'myPack'. We have imported that class from the package. The entire 'util' package is also imported in the program. Remember the 'util' package also contains Vector class. So while using the classes in the program we have to refer them by their fully qualified name such as,

```
myPack.Vector       //for user defined class
```
and
```
java.util.Vector    //for Java library class
```

Hiding the classes

The asterisk form of import statement imports all the classes from the package in program. However, we may prefer to "not import" certain classes. That is, we may like to hide these classes from accessing outside of the package. So, we can give the default access to these classes. Such classes will not be used outside of the package.

For example:

```
package myPack;
public class A
{
     //body of A
}
class B
{
     //body of B
}
```

Here, after compilation of the program both the classes A and B will become the part of package 'myPack'. When a program has imported the package 'myPack' then class A will be available in that program but B will not be available. So we can hide the class from accessing it outside the package by specifying the default access.

Static import

The JDK5 has added a new feature to the Java called static import to extend the capabilities of the import keyword. By following import keyword with static, an import statement can be used to import static members of the class or interface. By using static import, it is possible to refer static members of the class directly by their names. They need not have to refer the class name with them. This simplifies the syntax required to use the static member.

Let's begin this concept with an example which uses three different static methods from three different classes. The 'Math' class from lang package contains a static method called ‘sqrt( )’ which finds the square root of the number.

```
//Non static import method
  class NonStaticImport
  {
     public static void main(String args[])
     {
        double x = Math.sqrt(52);       //method1
        System.out.print("Square root of 52: ");
        System.out.println(x);          //method2

        char ch = 'D';
        if(Character.isUpperCase(ch))   //method3
          System.out.println(ch+" is uppercase");
     }
  }
```

Program using static methods

Output:

```
Square root of 52: 7.211102550927978
D is uppercase
```

Program uses three static methods that are sqrt( ) of Math class, println( ) of System class and isUpperCase( ) of Character class. All these are the part of java.lang package. While using these methods in program we have to refer them by using the class name along with them. We can eliminate this by importing these methods using static import as shown in program below. This is modified version of program shown above.

```
//Static import
  import static java.lang.Math.sqrt;
  import static java.lang.Character.isUpperCase;
  import static java.lang.System.out;
  class StaticImport
  {
     public static void main(String args[])
     {
        double x = sqrt(52);  //method1
        out.print("Square root of 52: ");
        out.println(x);       //method2

        char ch = 'D';
        if(isUpperCase(ch))   //method3
          out.println(ch+" is uppercase");
     }
  }
```

Program using the static import statement

Output:

```
Square root of 52: 7.211102550927978
D is uppercase
```

Here all three methods are brought into view by the static import statements:

```
import static java.lang.Math.sqrt;
import static java.lang.Character.isUpperCase;
import static java.lang.System.out;
```

After these statements, it is no longer necessary to qualify sqrt( ), println( ) and isUpperCase( ) with their class name. Therefore the statements method1, method2 and method3 are more simplified. It is considerably more readable.

There are two general form of the static import statement. The first used in program 4.4 which brings a particular method into view. Its general form is:

```
import static packagename.classname.methodname;
```

Second form of static import imports all static members of given class or interface into view. Its general form is:

```
import static packagename.classname.*;
```

This will import all static members of particular classname from packagename into view of the program. For example:

```
import static java.lang.Math.*;
import static java.lang.Character.*;
```

By writing these statements in the program we can use pow( ), log( ), sin( ) as well as other static methods of Math class to refer directly by their names. The Character class static methods such as isLowerCase( ), toUpperCase( ) can also be used referred directly.

Interface

We have discussed the concept of classes and the inheritance. We also learned about the various forms of inheritance. Here one thing is pointed out that the Java does not support the concept of multiple inheritance. That is, classes in Java will not have more than one super class. The definition like this,

```
class A extends B extends C
{
        -------;
}
or
class A extends B, C
{
        -------;
}
```

is not permitted in Java. But the Java designers do not ignore this fact. In many applications it is required to use multiple inheritance. For these purpose Java has added an additional functionality called Interface to support the concept of multiple inheritance.

Interface is Java's abstract class implementation. They can not be instantiated. That is, using interface, we can specify what a class must do, but not how it does it. Interfaces are syntactically similar to classes, but they do not have instance variables, and their methods are declared without having any body. This means that we can define interfaces which don't make assumptions about how they are implemented. Once the interface is defined, any number of classes can implement an interface. Also, one class can implement any number of interfaces. To implement an interface, a class must create the complete set of methods defined by the interface. However, each class can define the methods with different definitions. By providing the interface keyword, Java allows us to fully utilize the "one interface, multiple methods" aspect of polymorphism. Interfaces are designed to support dynamic method resolution at run time.

Defining an interface

An interface is basically a kind of the class. Like classes, interfaces contain methods and variables but with the major difference. The difference is that interfaces define only abstract methods and final variable fields. That is, interfaces do not specify any code to implement these methods and the data fields contain only constants. The syntax of defining an interface is very similar to defining a class. The general form of defining an interface is:

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

Here the ‘access’ is either public or not used (i.e. default). ‘InterfaceName’ is valid Java identifier name which is the name given to interface. Notice that the methods which are declared having 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. For example:

```
interface College
{
     int studs = 120;
     void play();
     int countUp(int x);
}
```

Here ‘College’ is the name of interface having default accessibility. Variable ‘studs’ is initialized with value 120. By default, this variable is static and final. The interface contains two methods declaration i.e. void play( ) and countUp(int x)

Implementing interfaces

Once an interface has been defined, one or more classes can implement that interface. To implement an interface, we have to 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-body
}
```

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 declaration of the implementing method must match exactly the type declaration specified in the interface definition. For example:

```
class Kkwp implements College
{
     public void play()
     {
          System.out.println("Let's play !!!");
     }
     public int countUp(int x)
     {
          x++;
          return(x);
     }
}
```

Here the class 'Kkwp' has implemented the interface 'College'. So this class must define both the methods declared in the interface. Note that these methods are using public access specifier.

It is both permissible and common for classes that implement interfaces to define additional members of their own also. For example:

```
class Kkwp implements College
{
     void display() //own method of kkwp
     {
          System.out.println("My method");
     }
     public void play()
     {
          System.out.println("Let's play !!!");
     }
     public int countUp(int x)
     {
          x++;
          return(x);
     }
}
```

Accessing implementations by interface references

We can declare variables as object references that use an interface rather than a class type. Any instance of any class that implements the declared interface can be referred to by such a variable. When we call a method through one of these references, the correct version of the method 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, allows classes to be created later than the code which calls methods on them. The "calling" code can dispatch through an interface without having to know anything about the "callee." This process is just similar to use a super class reference to access a subclass object, as described in Chapter 2.

Program 4.5 illustrates the concept of accessing implementations by interface references.

```
// Accessing the class through interface reference
  interface Mango
  {
      void display();
  }
  class Summer implements Mango
  {
      public void display()
      {
         System.out.println("First Display method... ");
      }
  }
  class Winter implements Mango
  {
      public void display()
      {
         System.out.println("Second Display method...");
      }
  }
  class MyInterface
  {
      public static void main(String args[])
      {
          Mango m = new Summer();  //statement1
          Winter n = new Winter();

          m.display();

          m = n;    //assigning the reference
          m.display();
      }
  }
```

Program: Accessing the class through interface reference

Output:

```
First Display method...
Second Display method...
```

In program, the reference variable ‘m’ of the interface ‘Mango’ has been created in statement1. For the first time it is referring the class ‘Summer’. We can assign the object of another class ‘Winter’ to this reference variable. Because ‘Winter’ is also implemented from ‘Mango’. After this assignment, ‘m’ can be used to refer the class ‘Winter’. This concept is most useful when large hierarchies of the inheritance are created.

Partial implementation of methods

When we implement the interface into the class, we need to define or implement all the methods of interface inside the class. If we want to eliminate this the keyword abstract is used in front of the class name. For example:

```
interface Mango
{
     void display();
     void show();
}
class Summer implements Mango
{
     public void display()
     {
          System.out.println("Display method");
     }
}
```

When we compile the code containing such implementation the compiler will flash the error:

```
Summer is not abstract and does not override abstract method
show() in Mango
```

In this case we have to make the class ‘Summer’ as abstract which means the class can implement any number of methods from the interface as,

```
abstract class Summer implements Mango
```

Creating multiple inheritance

In order to understand the power of interfaces, let’s look at the practical example which creates the multiple inheritance.

```
// Implementing multiple inheritance
  class Player
  {
      String name;
      void getName(String n)
      {
          name = n;
      }
      void putName()
      {
          System.out.println("Name: "+name);
      }
  }
  class Records extends Player
  {
      float avg, sRate;
      void getData(float a, float s)
      {
         avg = a;
         sRate = s;
      }
      void putData()
      {
         System.out.println("Records :- ");
         System.out.println("Average: "+avg);
         System.out.println("Strike rate: "+sRate);
      }
  }
  interface IPL
  {
      void displayEarning();
  }
  interface Information
  {
      void getTeam(String t);
  }
  class Match extends Records implements IPL, Information
  {
      long earn;
      Match(long e)
      {
          earn = e;
      }
      public void displayEarning()
      {
         System.out.print("His earning : "+earn);
         System.out.println(" Rupees");
      }
      public void getTeam(String t)
      {
         System.out.println("Team: "+t);
      }
  }
  class MultiInheritance
  {
      public static void main(String args[])
      {
         Match s = new Match(5600000);
         s.getName("Shane Warne");
         s.putName();
         s.getTeam("Rajasthan Royals");
         s.getData(30.52f, 123.45f);
         s.putData();
         s.displayEarning();
      }
  }
```

Program: Implementation of multiple inheritance

Output:

```
Name: Shane Warne
Team: Rajasthan Royals
Records :-
Average: 30.52
Strike rate: 123.45
His earning : 5600000 Rupees
```

In program, it contains four classes and two interfaces. The hierarchy shown in figure below has been created. The class 'Match' has been inherited from class 'Records' and implemented from interfaces 'IPL' and 'Information'. This creates a multiple inheritance. The interfaces 'IPL' and 'Information' can also be implemented in other classes such as 'Player' and 'Records'. So after compiling the source file six different .class files will be created including classes and interfaces.

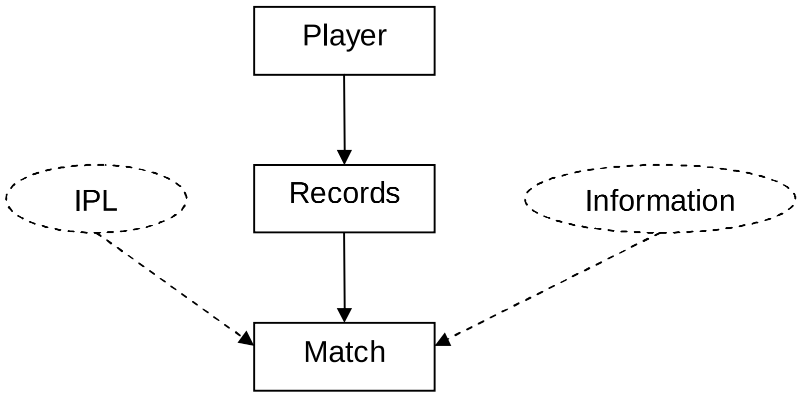

Fig. multiple inheritance using interface

Nested interfaces

An interface can be declared as the part of a class or another interface. Such interface is called as nested interface or member interface. Unlike general interface, a nested interface can be declared as public, private or protected. When a nested interface is used outside of its enclosing scope, it must be qualified by the name of the class or interface of which it is member. Thus, outside of the class or interface in which a nested interface is declared, its name must be fully qualified.

```
// Nesting of the interfaces.
  class CPU
  {
      public interface MicroProcessor
      {
          boolean isMultiCore(int core);
      }
  }
  class Computer implements CPU.MicroProcessor
  {
      public boolean isMultiCore(int core)
      {
          if(core>1)
             return true;
          else
             return false;
      }
  }
  class NestInterface
  {
      public static void main(String args[])
      {
          CPU.MicroProcessor comp = new Computer();
          int noOfCores = 4;

          if(comp.isMultiCore(noOfCores))
            System.out.println("It is multi-core");
          else
            System.out.println("It is single core");
      }
  }
```

Program: Nested interface

Output:

```
It is multi-core
```

Program illustrates the concept of nested interface. Here, the interface ‘MicroProcessor’ is defined inside the class ‘CPU’. When we are implementing it in class ‘Computer’ we have used its fully qualified name that is, CPU.MicroProcessor. While creating the reference variable also, we used the same name. For this program, four .class files will be created that is CPU.class, Computer.class, NestInterface.class and fourth will be CPU$MicroProcessor.class.

Variables in the interfaces

An interface can also define named constants. These can be used by all the classes that implement particular interface. So they are called as shared constants. This is similar to using a header file in C/C++ to create a large number of **#define**d constants or **const** declarations. If an interface contains no methods, then any class that includes such an interface doesn't actually implement anything.

The constants defined by interface are considered to be public, static and final. These modifiers are usually omitted from the declaration. They must be initialized with the value. By implementing the interface the constants can directly be accessed. When interface is not implemented then these can be referred by their fully qualified name. Program 4.8 illustrates the use of variables in the interfaces.

```
//Using constants in interface
interface Constants
{
    double PI = 3.14;
    String unit = " sq.cm";
}
interface Values
{
    int cir = 2;
}
public class InterfaceVar implements Constants
{
    public static void main(String args[])
    {
         double rad = 5.83;
         System.out.print("Area of circle: ");
         System.out.println(PI * rad * rad + unit);

         System.out.print("Perimeter of circle: ");
         System.out.println(Values.cir*PI*rad);
    }
}
```

Program: Using variables in the interfaces

Output:

```
Area of circle: 106.72514600000001 sq.cm
Perimeter of circle: 36.6124
```

In program, two interfaces are defined ‘Values’ and ‘Constants’. But our class has implemented only interface ‘Constants’. So, all the values available in this interface will automatically be made available in our class. In order to use the constant from interface ‘Values’ we have referred it by using the interface name directly as in the last line of the program.

Interface inheritance

An interface can extend or inherit other interfaces, using the ‘extends’ keyword. Unlike extending classes, an interface can extend several interfaces. The interfaces extended by an interface (directly or indirectly), are called super-interfaces. Conversely, the interface is a sub-interface of its super-interfaces. Since interfaces define new reference types, super-interfaces and sub-interfaces are also super-types and subtypes, respectively.

A sub-interface inherits all methods from its super-interfaces, as their method declarations are all implicitly public. A sub-interface can override method prototype declarations from its super-interfaces. Here, overridden methods are not inherited. Method prototype declarations can also be overloaded, analogous to method overloading in the classes.

```
//Extending the interface
  interface SYIT
  {
      int strength = 68;
      void displaySecond();
  }
  interface TYIT extends SYIT
  {
      int strength = 55;
      void displayThird();
      void displayAll();
  }
  class InfoTech implements TYIT
  {
      int total;
      public void displaySecond()
      {
         System.out.println("SYIT: "+SYIT.strength);
      }
      public void displayThird()
      {
         System.out.println("TYIT: "+TYIT.strength);
      }
      public void displayAll()
      {
         total = SYIT.strength + TYIT.strength;
         System.out.println("Total: "+total);
      }
  }
  class ExtendInterface
  {
      public static void main(String args[])
      {
           InfoTech x = new InfoTech();
           x.displaySecond();
           x.displayThird();
           x.displayAll();
      }
  }
```

Program: Extending the interfaces

Output:

```
SYIT: 68
TYIT: 55
Total: 123
```

In program, the interface 'TYIT' is inherited from 'SYIT'. Class 'InfoTech' has implemented sub-interface 'SYIT'. So, it is necessary for that class to implement all the three methods of both interfaces. If any one of these methods is not defined then compiler will flash run-time error. Remember interfaces support all types of inheritance.
