Packages and Interfaces

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 *pkg;*
Here, *pkg* is the name of the package.

For example, the following statement creates a package called **MyPackage**:

```
package MyPackage;
```

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

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

```
package MyPack;

/* Now, the Balance class, its constructor, and its
   show() method are public. This means that they can
   be used by non-subclass code outside their package.
*/
public class Balance {
  String name;
  double bal;

  public Balance(String n, double b) {
    name = n;
    bal = b;
  }

  public void show() {
    if(bal<0)
      System.out.print("--> ");
    System.out.println(name + ": $" + bal);
  }
}
```

```
import MyPack.*;

class TestBalance {
  public static void main(String args[]) {
        /* Because Balance is public, you may use Balance
           class and call its constructor. */
        Balance test = new Balance("J. J. Jaspers", 99.88);

        test.show(); // you may also call show()
    }
}
```

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:

• 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 modifiers, private, public, and protected, provide a variety of ways to produce the many levels of access required by these categories.

	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

```
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**:

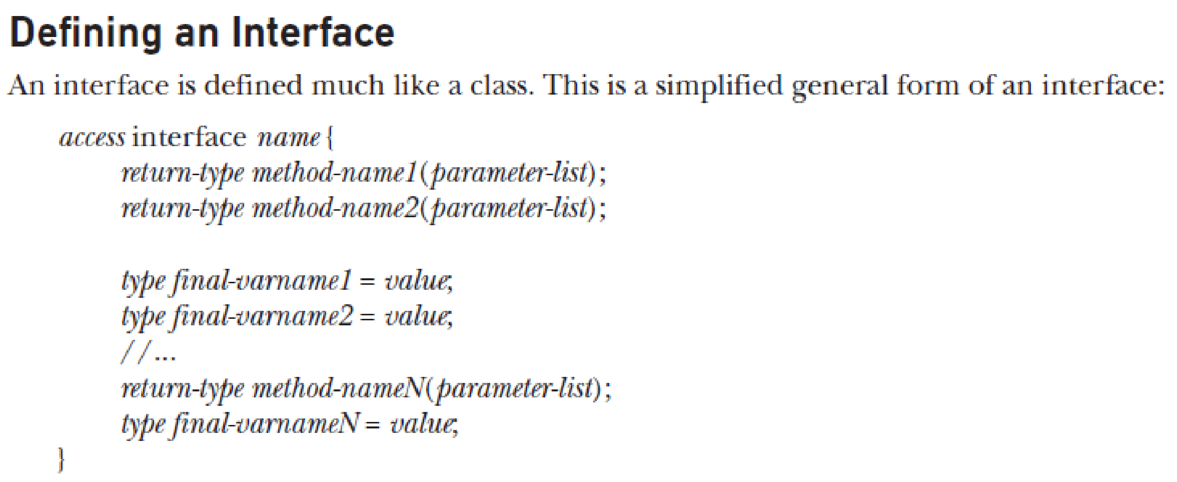
```
package p1;

class Derived extends Protection {
  Derived() {
    System.out.println("derived constructor");
    System.out.println("n = " + n);

// class only
// System.out.println("n_pri = "4 + 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);
  }
}
```

Interfaces

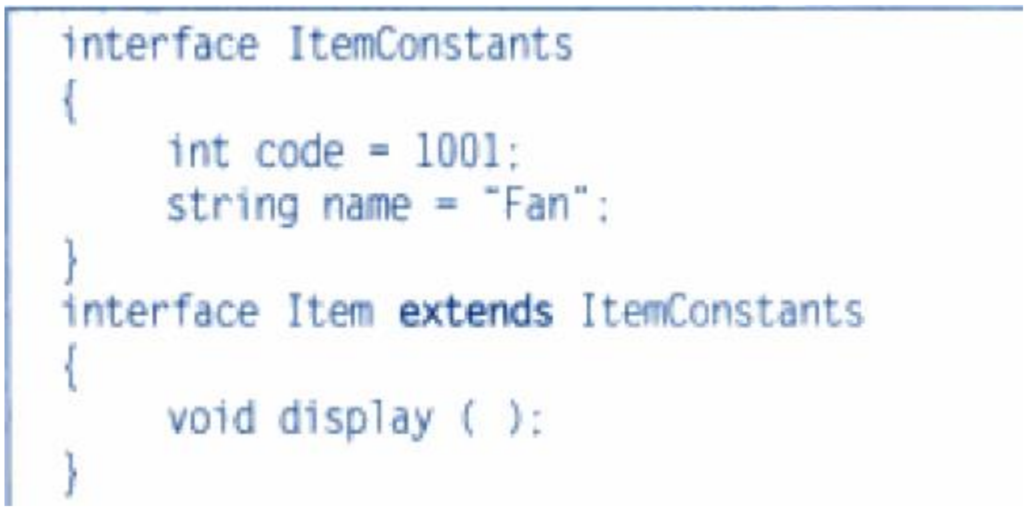
- Using the keyword interface, you can fully abstract a class' interface from its implementation.
- That is, using interface, you can specify what a class must do, but not how it does it.
- Interfaces are syntactically similar to classes, but they lack instance variables, and, as a general rule, their methods are declared without any body.
- In practice, this means that you can define interfaces that don't make assumptions about how they are implemented.
- Once it is defined, any number of classes can implement an interface. Also, one class can implement any number of interfaces.
- By providing the interface keyword, Java allows you to fully utilize the "one interface,
- multiple methods" aspect of polymorphism.
- Interfaces are designed to support dynamic method resolution at run time.
- Normally, in order for a method to be called from one class to another, both classes need to be present at compile time so the Java compiler can check to ensure that the method signatures are compatible



Defining an Interface

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

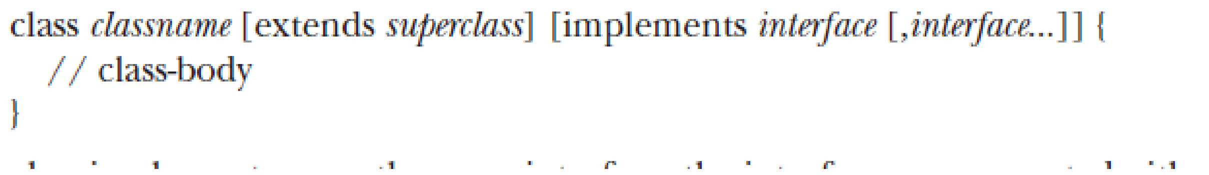
```
access interface name {
        return-type method-name1(parameter-list);
        return-type method-name2(parameter-list);

        type final-varname1 = value;
        type final-varname2 = value;
        //...
        return-type method-nameN(parameter-list);
        type final-varnameN = value;
}
```



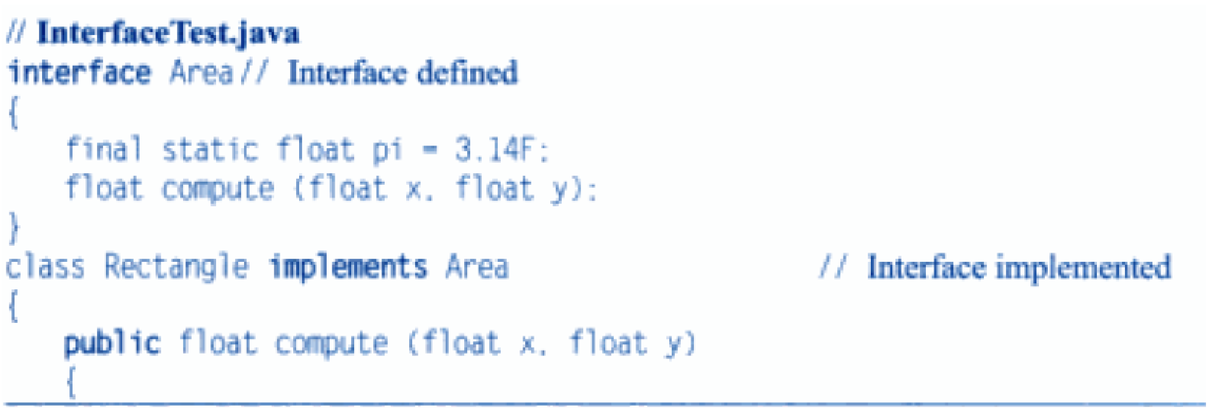
```
interface ItemConstants
{
      int code = 1001;
      string name = "Fan";
}
interface Item extends ItemConstants
{
      void display ( );
}
```

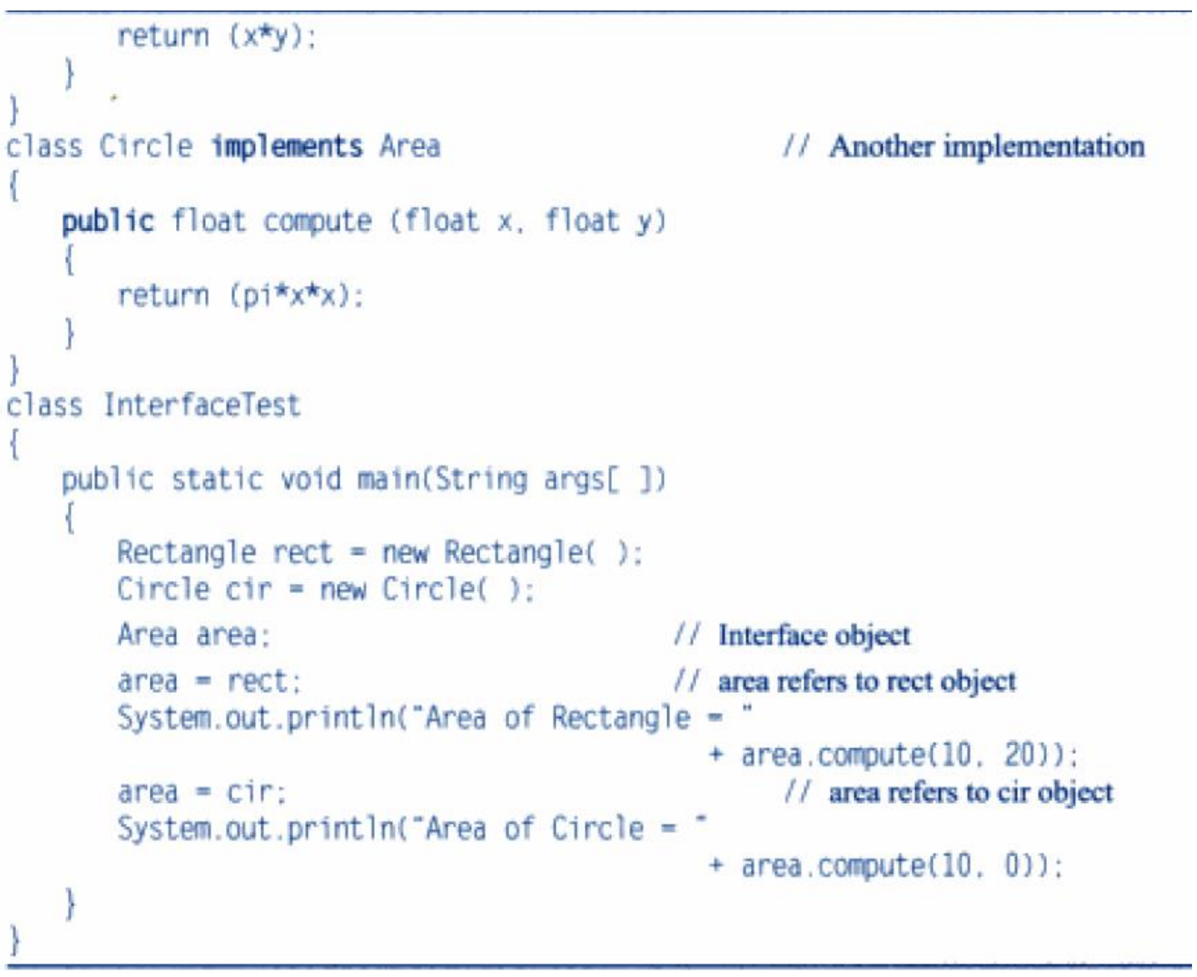
Implementing Interfaces

- 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 required by the interface.
- The general form of a class that includes the implements clause looks like this:



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



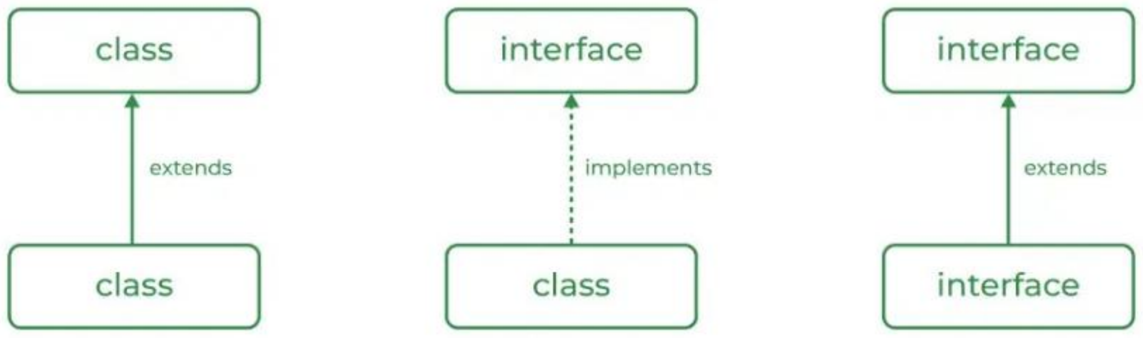


```
// InterfaceTest.java
interface Area// Interface defined
{
    final static float pi = 3.14F;
    float compute (float x, float y);
}
class Rectangle implements Area                    // Interface implemented
{
    public float compute (float x, float y)
    {
        return (x*y);
    }
}
class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

Relationship Between Class and Interface

A class can extend another class similar to this an interface can extend another interface.

But only a class can extend to another interface, and vice-versa is not allowed.



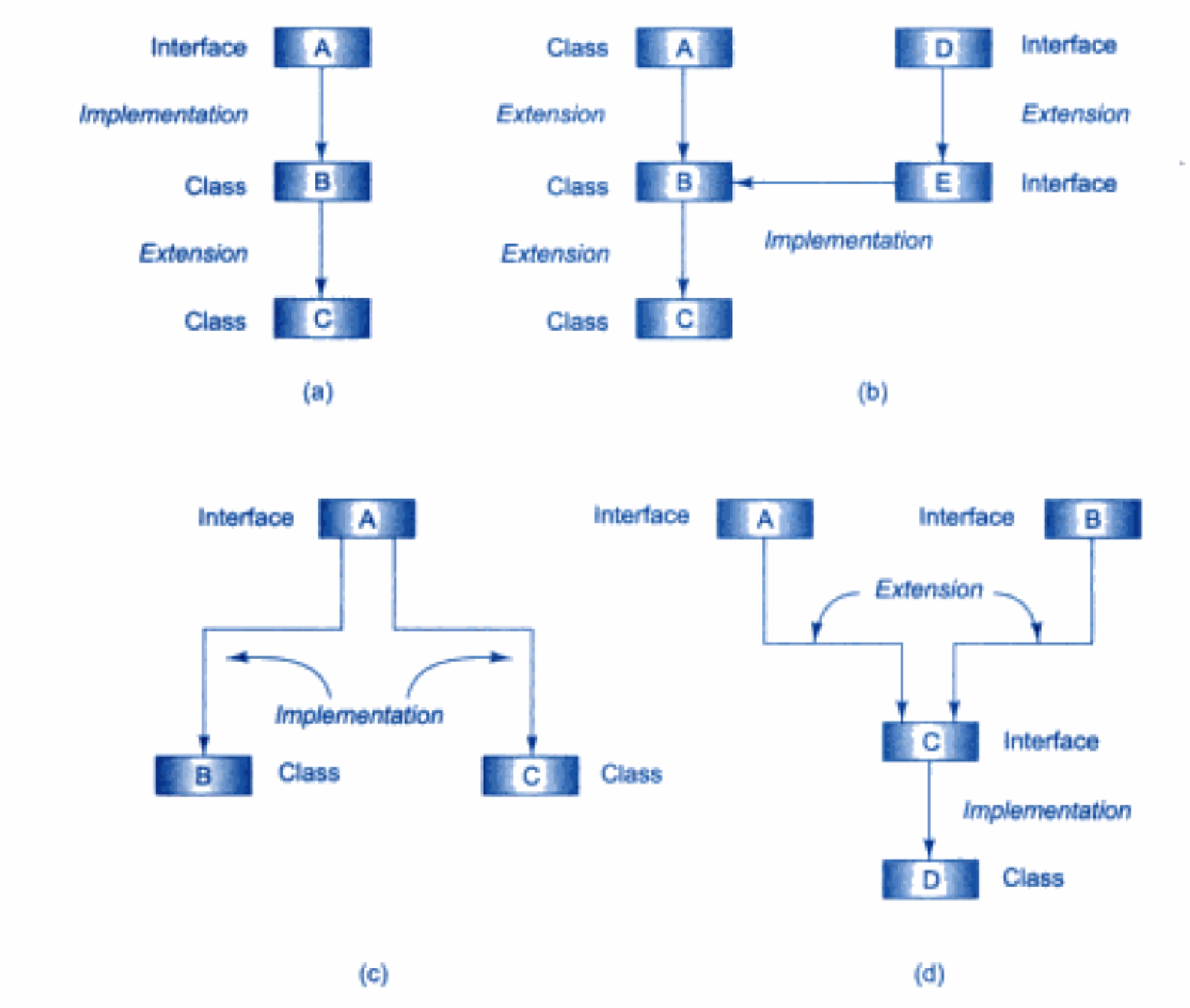


Fig. 10.1 *Various forms of interface implementation*

Difference Between Class and Interface

Although Class and Interface seem the same there have certain differences between Classes and Interface. The major differences between a class and an interface are mentioned below:

Class	Interface
In class, you can instantiate variables and create an object.	In an interface, you can't instantiate variables and create an object.
A class can contain concrete(with implementation) methods	The interface cannot contain concrete(with implementation) methods
The access specifiers used with classes are private, protected, and public.	In Interface only one specifier is used- Public.

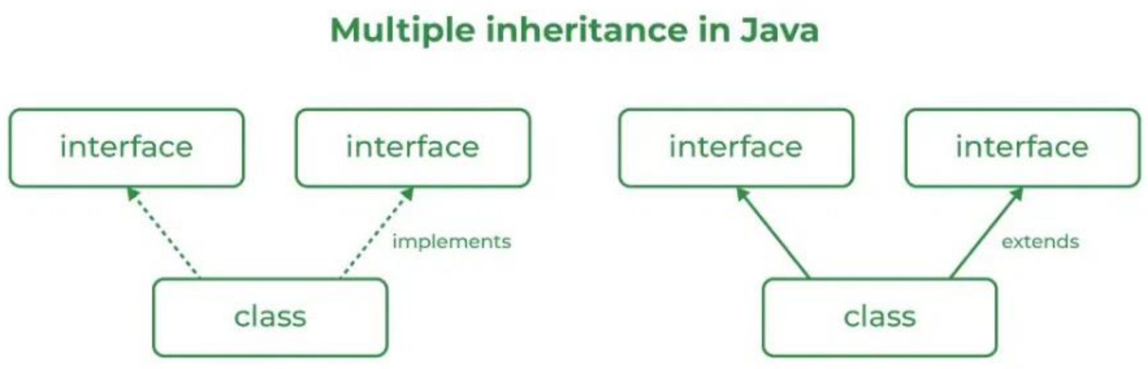
Advantages of Interfaces in Java

The advantages of using interfaces in Java are as follows:

1. Without bothering about the implementation part, we can achieve the security of the implementation.
2. In Java, multiple inheritances are not allowed, however, you can use an interface to make use of it as you can implement more than one interface.

Multiple Inheritance in Java Using Interface

Multiple Inheritance is an OOPs concept that can't be implemented in Java using classes. But we can use multiple inheritances in Java using Interface. let us check this with an example.



Program 10.2 illustrates the implementation of the concept of multiple inheritance using interfaces.

Program 10.2 ***Implementing multiple inheritance***

```
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("part 1 = " + 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
```