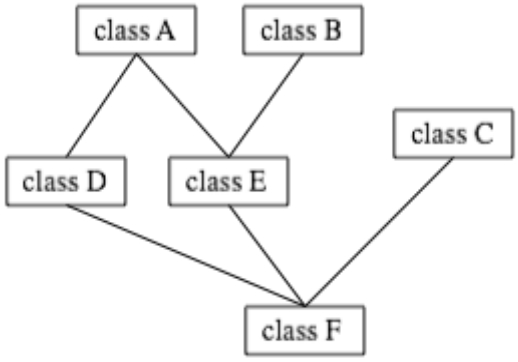
INTERFACES

Some object-oriented programming languages, such as C++, allow a class to extend two or more superclasses. This is called multiple inheritance. In the illustration below, for example, class E is shown as having both class A and class B as direct superclasses, while class F has three direct superclasses.



Multiple inheritance (NOT allowed in Java)

Such multiple inheritance is **not** allowed in Java. The designers of Java wanted to keep the language reasonably simple, and felt that the benefits of multiple inheritance were not worth the cost in increased complexity. However, Java does have a feature that can be used to accomplish many of the same goals as multiple inheritance: interfaces.

We've encountered the term "interface" before, in connection with black boxes in general and subroutines in particular. The interface of a subroutine consists of the name of the subroutine, its return type, and the number and types of its parameters. This is the information you need to know if you want to call the subroutine. A subroutine also has an implementation: the block of code which defines it and which is executed when the subroutine is called.

In Java, `interface` is a reserved word with an additional, technical meaning. An "`interface`" in this sense consists of a set of instance method interfaces, without any associated implementations. (Actually, a Java interface can contain other things as well, but we won't discuss them here.) A class can implement an `interface` by providing an implementation for each of the methods specified by the interface. Here is an example of a very simple Java `interface`:

```
public interface Drawable {
   public void draw(Graphics g);
}
```

This looks much like a class definition, except that the implementation of the `draw()` method is omitted. A class that implements the `interface` *Drawable* must provide an implementation for this method. Of course, the class can also include other methods and variables. For example,

```
public class Line implements Drawable {
    public void draw(Graphics g) {
        . . . // do something -- presumably, draw a line
    }
    . . . // other methods and variables
}
```

Note that to implement an interface, a class must do more than simply provide an implementation for each method in the interface; it must also **state** that it implements the interface, using the reserved word`implements` as in this example: "`public class Line` **`implements`** `Drawable`". Any class that implements the *Drawable* interface defines a `draw()` instance method. Any object created from such a class includes a `draw()` method. We say that an **object** implements an `interface` if it belongs to a class that implements the interface. For example, any object of type *Line* implements the *Drawable* interface.

While a class can **extend** only one other class, it can **implement** any number of interfaces. In fact, a class can both extend one other class and implement one or more interfaces. So, we can have things like

```
class FilledCircle extends Circle
                        implements Drawable, Fillable {
   . . .
}
```

The point of all this is that, although interfaces are not classes, they are something very similar. An interface is very much like an abstract class, that is, a class that can never be used for constructing objects, but can be used as a basis for making subclasses. The subroutines in an interface are abstract methods, which must be implemented in any concrete class that implements the interface. You can compare the*Drawable* interface with the abstract class

```
public abstract class AbstractDrawable {
   public abstract void draw(Graphics g);
}
```

The main difference is that a class that extends *AbstractDrawable* cannot extend any other class, while a class that implements *Drawable* can also extend some class. As with abstract classes, even though you can't construct an object from an interface, you can declare a variable whose type is given by the interface. For example, if *Drawable* is an interface, and if *Line* and *FilledCircle* are classes that implement*Drawable*, then you could say:

```
Drawable figure;  // Declare a variable of type Drawable.  It
can
                  //    refer to any object that implements the
                  //    Drawable interface.
```

```
figure = new Line();   // figure now refers to an object of
class Line
figure.draw(g);   // calls draw() method from class Line

figure = new FilledCircle();     // Now, figure refers to an
object
                                    //   of class FilledCircle.
figure.draw(g);        //   calls   draw()   method   from   class
FilledCircle
```

A variable of type *Drawable* can refer to any object of any class that implements the *Drawable* interface. A statement like `figure.draw(g)`, above, is legal because `figure` is of type *Drawable*, and **any***Drawable* object has a `draw()` method. So, whatever object `figure` refers to, that object must have a `draw()` method.

Note that a type is something that can be used to declare variables. A type can also be used to specify the type of a parameter in a subroutine, or the return type of a function. In Java, a type can be either a class, an interface, or one of the eight built-in primitive types. These are the only possibilities. Of these, however, only classes can be used to construct new objects.

You are not likely to need to write your own interfaces until you get to the point of writing fairly complex programs. However, there are several interfaces that are used in important ways in Java's standard packages. You'll learn about some of these standard interfaces in the next few chapters, and you will write classes that implement them.

Source : http://math.hws.edu/javanotes/c5/s7.html