CHAPTER: 13

Interfaces

Objectives:

By the end of this chapter you should be able to:

- *explain what is meant by the term **interface** in Java;*
- *create and use your own interfaces.*

13.1 Introduction

Up to now, the applications we created were fairly simple, consisting for the most part of only one or two classes. In reality, applications that are developed for commercial and industrial use comprise a large number of classes, and are developed not by one person, but by a team. Members of the team will develop different modules which can later be integrated to form a single application. When groups of programmers work together in this way, they often have to agree upon how they will develop their own individual classes in order for them to be successfully integrated later. In this chapter we will see how these agreements can be formalized in Java programs by making use of a special kind of class called an **interface**. To demonstrate this we will begin by considering once again the testing process that we discussed in chapter 11.

13.2 An example

It is a very common occurrence that the attributes of a class should be allowed to take only a particular set of values. Think, for example, of the `BankAccount` class that we developed in the last semester. It is likely that the account number should be restricted to numbers that contain, say, precisely eight digits. Similarly, a `Customer` class might require that the customer number comprises a letter followed by four digits. In some cases there are constraints that exist not because we choose to impose them, but because they occur "naturally" - in the `Rectangle` class, for example, it would make no sense if an object of this class were to have a length or height of zero or less.

In such cases, every effort must be made when developing the class to prevent invalid values being assigned to the attributes. Constructors and other methods should be designed so that they flag up errors if such an attempt is made - and one of the advantages of object-oriented programming languages is precisely that they allow us to restrict access to the attributes in this way.

The above remarks notwithstanding, it is the case that in industrial sized projects, classes will be very complex and will contain a great many methods. It is therefore possible that a developer will overlook a constraint at some point, and allow objects to be created that break the rules. It might therefore be useful if, for testing purposes, every object could contain a method, which we might call `check`, that could be used to check the integrity of the object.

In a particular project, people could be writing test routines independently of the people developing the modules, and these routines will be calling the `check` method. We need, therefore, to be able to *guarantee* that every object contains such a `check` method.

You learnt in chapter 9 that the way to guarantee that a class has a particular method is for that class to inherit from a class containing an **`abstract`** method - when a class contains an **`abstract`** method, then any subclass of that class is *forced* to override that method - if it does not do so, a compiler error occurs.

In our example, we need to ensure that our classes all have a `check` method that tests an object's integrity, so one way to do this would be to write a class as follows:

The *Checkable* class

```
public abstract class Checkable
{
   public abstract boolean check();
}
```

Now all our classes could extend `Checkable`, and would compile successfully only if they each had their own `check` method.

While this would work, it does present us with a bit of a problem. What would happen, for example, if our `Rectangle` class were going to be part of a graphical application and needed to extend another class such as `JFrame`? This would be problematic, because, in Java, a class is not allowed to inherit from more that one superclass. Inheriting from more than one class is known as **multiple inheritance** and is illustrated in figure 13.1.

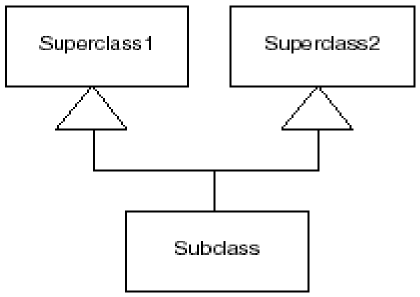

Fig 13.1. Multiple inheritance - not allowed in Java

One reason that multiple inheritance is disallowed is that it can easily lead to ambiguity, and hence to programming errors. Imagine for example that the two superclasses shown in figure 13.1 both contained a method of the same name - which version of the method would be inherited by the subclass?

Luckily, there is a way around this, because Java allows a kind of *lightweight inheritance*, made possible by a construct to which you were introduced in chapter 11 - the **interface**.

13.3 Interfaces

As you learnt in chapter 10, an interface is a class in which *all* the methods are **`abstract`**. As you will recall, when we want a class to inherit the methods of an interface we use the word **`implements`**, rather than **`extends`**. Just as with inheritance, once a class implements an interface it has the same type as that interface, as well as being of the type of its own class. So, for example, if a class implements `ActionListener`, then any object of that class is a *kind of* `ActionListener` - in other words it is of type `ActionListener`, as well as being of the type of its own class.

So far `ActionListener` is the only interface you have come across. The Java Foundation Classes, particularly those associated with graphical applications, contain a great many more. Later in this chapter you will come across two of these - `MouseListener` and `MouseMotionListener`. Of course it is perfectly possible for us to write our own interfaces as we will do in a moment when we turn our `Checkable` class into an interface.

Figure 13.2 shows the UML notation for the implementation of interfaces - you can see that interfaces are marked with the <<interface>> tag, and that a circle is used to indicate a class implementing an interface.

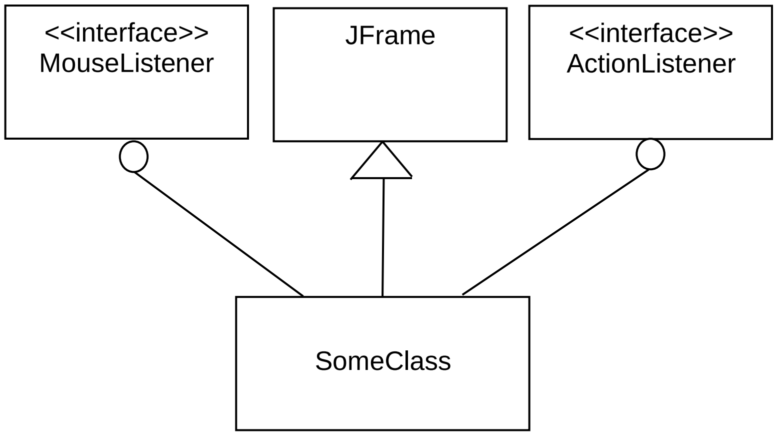

Fig 13.2. A class can inherit from only one superclass, but can implement many interfaces

As can be seen in figure 13.2, while it is possible to *inherit* from only one class, it is perfectly possible to implement any number of interfaces. In this case `SomeClass` **`extends`** `JFrame` and **`implements`** `MouseListener` and `ActionListener` - because the methods are not actually coded in the interface means that the problems with multiple inheritance that we described earlier do not arise.

As with inheritance, a class is *obliged* to override all the methods of the interfaces that it implements. By implementing an interface we are *guaranteeing* that the subclass will have certain methods. In the case of the `ActionListener` interface, you are aware that this contains a single method, `actionPerformed` - so in figure 13.2, `SomeClass` is guaranteed to have an `actionPerformed` method. It is also guaranteed to have all the methods declared in `MouseListener`, an interface that you will meet in the next section.

So, in cases where we need a class such as `Checkable` in which all the methods are **`abstract`** we don't create a class - instead we create an interface.

Let's turn our `Checkable` class into an interface! The code looks like this:

The *Checkable* interface

```
public interface Checkable
{
   public boolean check();
}
```

Notice the word **`interface`** instead of **`class`** - and notice also that we don't have to declare our method as **`abstract`**, because by definition all methods of an **`interface`** are **`abstract`**.

Let's make the `Rectangle` class from chapter 8 checkable by implementing this interface - all we need to do is change the header, and code the `check` method. The class will now look like this:

The *Rectangle* class implementing the *Checkable* interface

```
public class Rectangle implements Checkable
{
  private double length;
  private double height;

  public Rectangle(double lengthIn, double heightIn)
  {
      length = lengthIn;
      height = heightIn;
  }

  public double getLength()
  {
      return length;
  }

  public double getHeight()
  {
      return height;
  }

  public void setLength(double lengthIn)
  {
      length = lengthIn;
  }

  public void setHeight(double heightIn)
  {
   height = heightIn;
  }

  public double calculateArea()
  {
      return length * height;
  }

  public double calculatePerimeter()
  {
      return 2 * (length + height);
  }

  public boolean check() // we must implement this method as it is declared in the Checkable interface
  {
      return length > 0 && height > 0;
  }
}
```

You can see that the class now implements our `Checkable` interface:

```
public class Rectangle implements Checkable
```

The `check` method, which the class is forced to override, returns a value of **`true`** if the attributes are both greater than zero, and **`false`** otherwise:

```
   public boolean check() // the check method of Checkable must be overridden
   {
      return length > 0 && height > 0;
   }
```

Other classes can implement the `Checkable` interface in a similar way. For example, a partial implementation of the `Customer` class that we talked about in section 13.2 might look like this:

The *Customer* class implementing the *Checkable* interface

```
public class Customer implements Checkable
{
   private String number;

   // other attributes

   public Customer(String numberIn)
   {
      number = numberIn;
   }

   public String getNumber()
   {
      return number;
   }


   public boolean check()
   {
      if(number.length() != 5)
      {
         return false;
      }
      else if(!Character.isLetter(number.charAt(0)))
      {
         return false;
      }
      else
      {
         for(int i = 1; i <= 4;  i++)
         {
            if(!Character.isDigit(number.charAt(i)))
            {
               return false;
            }
         }
      }
      return true;
   }

   // other methods
}
```

You can see here how, in the `check` method, we check firstly that the string contains exactly five characters, and then check if the first character is a letter by making use of the `isLetter` method of the `Character` class. Finally we check that each of the remaining characters is a digit by using the `isDigit` method of the `Character` class.

In program 13.1 below we create four objects - two `Rectangle` objects and two `Customer` objects. In each case the first object breaks the rules that we have set for these two classes, whereas the second does not - the first `Rectangle` object is given a height of zero, while the first `Customer` object has a customer number containing only three characters.

Program 13.1

```
public class Checker
{
   public static void main(String[] args)
   {
      // create two rectangles
      Rectangle rectangle1 = new Rectangle(0, 8); // invalid: first attribute is zero
      Rectangle rectangle2 = new Rectangle(10, 8); // valid

      // create two customers
      Customer customer1 = new Customer("A37"); // invalid: number must be 1 letter and 4 digits
      Customer customer2 = new Customer("S1234"); // valid

      // send objects to the check method
      checkObject(rectangle1);
      checkObject(rectangle2);
      checkObject(customer1);
      checkObject(customer2);
   }

   static void checkObject(Checkable objectIn) // note that the type of the parameter is Checkable
   {
      if(objectIn.check())  // call the check method
      {
         System.out.println("Valid object");
      }
      else
      {
         System.out.println("Invalid object");
      }
   }

}
```

As you can see, we send the four objects in turn into a method called `checkObject,` which calls the object's `check` method.

The `checkObject` method accepts a parameter of type `Checkable` - and of course both the `Rectangle` objects and the `Customer` objects are of type `Checkable` because they implement the `Checkable` interface.

As expected, the output from the program is as follows:

```
Invalid object
Valid object
Invalid object
Valid object
```

Implementing an interface is rather like making a contract with the user of a class - it *guarantees* that the class will have a particular method or methods. In the above case, a developer will know that any object that implements `Checkable` will have a `check` method. This enables the developer to write methods such as `checkObject` that expect to receive an object of type `Checkable`, in the certain knowledge that the object - whether it is a `Rectangle`, `Customer` or any other class that implements this interface - will have a method called `check`.