Chapter 12

Interfaces

An interface allows a group of unrelated objects to be treated in a similar manner. This simplifies the writing of code which processes the objects in some way. This chapter describes the concept of an interface and shows how it is implemented in Java. It also gives an appreciation of the kinds of polymorphic behavior that are possible when multiple interfaces are implemented.

12.1 The Concept of an Interface

In order to understand the concept of an interface in object-oriented programming, it is helpful to consider the following question, what do the following objects have in common?

- A door
- A bank account
- A book
- A drawer in a filing cabinet

It is clear that the four objects above are not structurally related. However, they are related through a set of common behaviors namely, `open()` and `close()`. This set of behaviors can be grouped into a concept called `Openable`. In object-oriented programming, the concept `Openable` is referred to as an *interface*.

It should be noted that `open()` and `close()` specify behaviors that are common to the objects above but these behaviors take place in different ways. For example, opening an account is a very different operation from opening a door.

12.2 A Real-World Example

Consider an alarm clock. There are hundreds of models of alarm clocks available on the market produced by different manufacturers. These models have a wide variety of shapes and sizes yet they provide the same basic functionality to users:

- Set current time
- Set alarm time
- Set alarm (radio or alarm ring)
- Display current time
- Sound alarm

The above functionality can be considered the interface of an alarm clock. Manufacturers of alarm clocks have the freedom to use different technology to design their alarm clocks. They may also choose to include features in addition to those that are required by the interface. However, manufacturers always ensure that the functionality of the interface is provided.

The user of an alarm clock expects to find features to set the current time, the alarm time, and the type of alarm to wake up to. They also expect the alarm clock to display the current time and to sound the alarm when the specified time is reached. Because manufacturers ensure that the functionality of the interface is provided, persons can use different models and brands of alarm clocks without being concerned that they will be able to set the current time, the alarm time, and the type of alarm to wake up to.

A similar benefit is derived in object-oriented programming by using interfaces. A group of classes that agree to provide features based on an agreed protocol (interface) simplifies the work of client objects which can now deal with instances of these classes in a standard way without being concerned about the differences in implementation.

12.3 Defining an Interface

As mentioned earlier, an *interface* specifies a set of common behaviors or a *protocol* for classes. The interface does not specify how the behavior is provided; this is the responsibility of the classes that implement the interface. A class that implements the interface must provide method implementations for all the behaviors specified in the protocol, unless that class is abstract.

In Java, an interface is defined using a special structure known as an `interface`. The behaviors of an interface are specified using a set of abstract methods.

To define the `Openable` interface in Java, the following code should be used:

```
public interface Openable
{
   public abstract void open();   // abstract keyword can be omitted
   public abstract void close();  // abstract keyword can be omitted
}
```

12.4 Implementing an Interface

Any class that wishes to obey the protocol specified by an interface must provide method implementations for all the abstract methods in the interface (unless that class is abstract). There is no restriction on how the methods are implemented in the class. The only requirement is that the method signatures of the interface are maintained. Thus, an interface is a "method signature contract" and there is no guarantee that the methods are implemented in any given manner. Note that a class that implements an interface can have additional methods of its own.

Consider the `Door` class. If the `Door` class wishes to conform to the `Openable` interface, it must first indicate that it `implements` the `Openable` interface. Next, it must provide method implementations for the `open()` and `close()` methods. Methods can also be written which provide behaviors for the `Door` class which are not required by the interface, e.g., `knock()`. An outline of the code for the `Door` class is given below:

```
public class Door implements Openable
{  // indicates that Door will conform to the Openable interface

   // instance variables of Door

   // constructor/s of Door

   // other methods of Door

   public void knock(int times) {
      // method implementation
   }
```

```
    public void open() {
        // method implementation
    }

    public void close() {
        // method implementation
    }
}
```

implementation of methods from `Openable` interface

To correctly implement the `Openable` interface, the `open()` and `close()` methods in `Door` must have the same signatures as specified in the interface. If `Door` claims to implement the `Openable` interface (by means of the `implements` keyword) and does not provide method implementations for one or more methods in the interface, it must be declared as an abstract class. Like any other abstract class, one or more descendants of `Door` (abstract or concrete subclass) will now be expected to provide the missing method implementations.

It is possible for `Door` to have other `open()` or `close()` methods with different signatures. For example,

```
public void open(int count) {
    // method implementation
}
```

This is an example of method overloading and is not related to the interface mechanism discussed in this chapter.

12.5 UML Notation for Interfaces

There are two ways to depict interfaces in the UML. One way is to draw an interface using a rectangular symbol similar to a class except that a *stereotype descriptor* `<<interface>>` is added above the interface name. If a class implements an interface, a line with a closed arrow tip is used to connect the class to the interface. This line is similar to the one used to depict an inheritance relationship, except that the line is broken.

Figure 12.1 is a UML diagram of the `Openable` interface and the `Door`, `Account`, and `Book` classes that implement the interface.

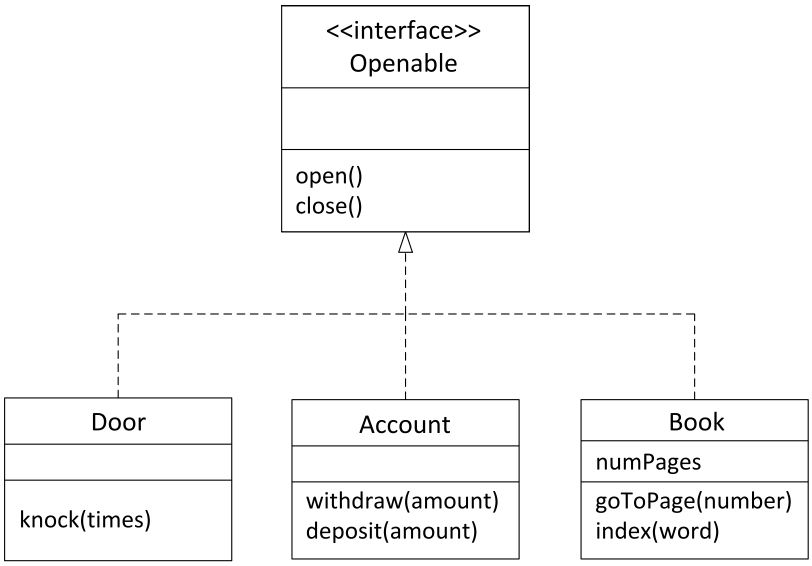

Figure 12.1 Depicting Interfaces in the UML

Another way to depict an interface in the UML is to use a circle to denote the interface. A class that implements the interface connects to the circle with a straight line. Figure 12.2 shows this notation:

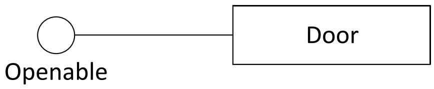

Figure 12.2: Another Way to Depict Interfaces in the UML

12.6 Properties of Interfaces

It is possible for an abstract class to have only abstract methods. It is also possible for an abstract class to have neither instance variables nor class variables. Under these two circumstances, an interface is identical to an abstract class. A method declared in an interface is implicitly abstract; also, any variable declared is implicitly a `static final` variable (i.e., a constant).

A constant `MAX_TRIES` can be explicitly declared in an interface as follows:

```
public static final int MAX_TRIES = 3;
            // declares MAX_TRIES as a constant (type is int)
```

The constant `MAX_TRIES` then becomes available to any class implementing the interface.

Like an abstract class, it is not possible to create an instance of an interface. However, unlike an abstract class, an interface cannot contain instance variables, class variables, or method implementations.

An interface and a concrete class can be viewed as two opposite ends of an implementation continuum. The interface is highly abstract and declares methods but provides no implementation of these methods. On the other hand, a concrete class must have method implementations of all its methods (directly or indirectly through inheritance). An abstract class is like a bridge between these two concepts in the continuum and may have method implementations as required.

An interface defines a new type and this type can be used to refer to objects created from classes implementing the interface, directly or indirectly. For example, the following code is legal in a client:

```
Openable o;    // declare variable of type Openable (interface name)
Door d;        // declare object variable of type Door (class name)

d = new Door();
               // create instance of Door

o = d;         // variable of type Openable refers to Door instance
```

In the above example, after assigning the variable of type `Door`, `d`, to a variable of type `Openable`, `o`, only methods declared in the `Openable` interface and those of the `Object` class[1] can be invoked on `o`. In other words, it is only possible to use the methods `open()` and `close()` and those of the `Object` class:

```
o.toString(); // OK, as well as other methods of Object
o.open();     // OK, since open() is a method in Openable
o.close();    // OK, since close() is a method in Openable
```

An attempt to invoke the `knock()` method on `o` will generate a compilation error since the `knock()` method is not present in the `Object` class and it is not declared in the `Openable` interface:

[1] Implicitly, the `Openable` object is at least an instance of `Object`, so all the methods of the `Object` class can be used.

```
o.knock(3);    // will generate a compilation error
```

Note that the error will be generated even though `o` is currently referring to an instance of `Door` which has a `knock()` method implemented. Since the interface defines a type, the compiler will only allow method invocations on `o` that are defined in the `Object` class or are declared in the *static* type, i.e., the method invocations that obey the interface specification.

12.7 The Comparable Interface

Consider the `Comparable` interface in Java[2]. This interface declares a single method, `compareTo()`, that is responsible for comparing one object with another and determining their relative order, according to some specified ordering for that class of objects. The following is the signature of the `compareTo()` method:

```
public abstract int compareTo (Object o);
```

The `compareTo()` method compares the current object (`this`) with the object passed as a parameter (`o`). If the current object is less than the supplied object or should appear before the supplied object in a sorted list, `compareTo()` should return a negative number. If it is greater than the supplied object or should come after the supplied object in a sorted list, `compareTo()` should return a positive integer. If the two objects are equivalent or their relative order in a sorted list does not matter, `compareTo()` should return zero. If `compareTo()` returns zero with a supplied object, the `equals()` method should typically return `true` with the supplied object.

To allow `Account` objects to be compared with each other (e.g., to allow sorting of accounts in a report), the `Account` class must implement the `Comparable` interface. This requires the `Account` class to provide a method implementation for the `compareTo()` method, in addition to its usual set of methods. The `compareTo()` method in `Account` can be implemented as follows.

```
public class Account implements Comparable
{
   // instance variables

```

[2] The `Comparable` interface is discussed in this chapter without the use of generics. In Chapter 13, a version of the `Comparable` interface using parameterized types is presented.

```
   // Account methods

   public int compareTo(Object o) {
      if (o instanceof Account) {
         Account a = (Account) o;
         if (this.num < a.num)
            return -1;
         else {
            if (this.num == a.num)
               return 0;
            else
               return 1;
         }
      }
      else
         throw new IllegalArgumentException ("Expected an Account");
   }
}
```

Since the `compareTo()` method accepts objects of the general type `Object`, it is necessary to check that the object being compared is indeed an instance of `Account` before proceeding with the comparison. If this is the case, the object is cast to an `Account` object and the comparison is then made.

The `Comparable` interface defines a type so the following code is legal in a client:

```
Comparable c1, c2;

Account a1 = new Account(10, 1000.00);
Account a2 = new Account(20, 2000.00);

c1 = a1;
c2 = a2;

int compare = c1.compareTo(c2);
```

Since the `compareTo()` method is present in the `Account` class, it is also possible to do the following in a client, without using a variable of type `Comparable`:

```
compare = a1.compareTo(a2);
```

The `Comparable` type makes it possible to take advantage of pre-defined methods that work on only `Comparable` objects. For example, the class `Collections` in Java provides a class method `sort()` that sorts the objects in a collection. An `ArrayList`, `al`, can be populated with a set of `Account` instances implementing the `Comparable` interface as shown above. It is easy to sort this `ArrayList` using the `sort()` method:

```
Collections.sort(al);
```

Now, assume that a set of `Door` instances implementing the `Comparable` interface is stored in the `ArrayList` instead of the set of `Account` instances. These instances can be sorted (based on the implementation of the `compareTo()` method in `Door`) using a similar method invocation:

```
Collections.sort(al);
```

Since the `sort()` method operates on objects of type `Comparable`, it is possible to sort a collection containing objects of any type, without knowing anything about the objects to be sorted. The only requirement is that the corresponding object class must implement the `compareTo()` method of `Comparable` interface. This is one of the most powerful uses of interfaces: being able to write code to deal with objects which are not known but which are guaranteed to provide a minimal set of behavior specified by an interface.

12.8 Polymorphism with Interfaces

Suppose that the `Door` and `Book` classes mentioned earlier implement the `Openable` interface together with the `Account` class. Next, assume that an instance of each class is created as follows:

```
Door d = new Door();
Book b = new Book();
Account a = new Account(10, 1000.00);
```

Now, suppose that these three instances are inserted into an array of type `Openable`:

```
Openable[] o = new Openable[3];
o[0] = d;
o[1] = b;
o[2] = a;
```

It is now possible to go through the collection of objects in the array and treat the objects as if they were all `Openable`, ignoring the fact that they are really a `Door`, a `Book`, and an `Account`:

```
for (int i=0; i<o.length; i++) {
   o[i].open();
   o[i].close();
}
```

This is similar to the case where `B` and `C` are subclasses of a superclass `A`, and variables of type `A` are used to refer to instances of `B` and `C` (based on the Principle of Substitutability).

It is another example of polymorphism at work since it allows objects of matching interfaces to be substituted for one another at run-time (using a common interface type such as `Openable` in the above example).

12.9 Implementing Multiple Interfaces

A class can implement more than one interface by simply separating the interfaces with commas after the `implements` keyword. For example, the `Account` class can implement both the `Comparable` interface and the `Openable` interface as follows:

```
public class Account implements Comparable, Openable
{
   :
}
```

If the `Account` class is not abstract, it must provide method implementations for the only method in the `Comparable` interface (`compareTo()`) and the two methods in the `Openable` interface (`open()` and `close()`), as discussed before. Assuming that this is the case, it is possible to create an `Account` object and assign its reference to variables of type `Comparable` and `Openable`, respectively:

```
Account a = new Account(10, 1000.00);

Comparable c = a;
Openable o = a;
```

The variable `c` can now be used wherever a variable of type `Comparable` is expected. Similarly, the variable `o` can be used wherever a variable of type `Openable` is expected. Of course, the variable `a` can be used wherever a variable of type Account is expected (including situations where a variable of type Object is expected). Thus, there are at least four situations where the single Account instance can be used, each one having a different view of the Account object. In essence, if a class implements multiple interfaces, instances of the class can be viewed in different ways at different times, depending on the "lens" (i.e., interface type) being used. This feature allows instances of a class to perform different roles at different times.

12.10 Inheritance of Interfaces

Some object-oriented programming languages such as Java only allow single inheritance, i.e., a child class can have at most one parent class. However, as mentioned in the previous section, it is possible for a class to implement more than one interface, allowing a restricted form of multiple inheritance. It is also possible to define an interface which inherits from more than one parent interface. For example,

```
public interface Multiple extends   Interface1,
                                    Interface2,
                                    Interface3 {
   // declaration of methods in Multiple
}
```

Suppose a concrete class MultipleImpl implements the Multiple interface. MultipleImpl must provide method implementations for all the methods declared in the Multiple interface as well as all the methods declared in Interface1, Interface2, and Interface3. Using substitutability, it will then be possible to assign a variable of type MultipleImpl to a variable of type Multiple, Interface1, Interface2, or Interface3. Of course, only the methods of the latter types can be used when this is done.

Consider the UML diagram of Figure 12.3 which shows the relationship between three interfaces, I1, I2, and I3, and three concrete classes, A, B, and C.

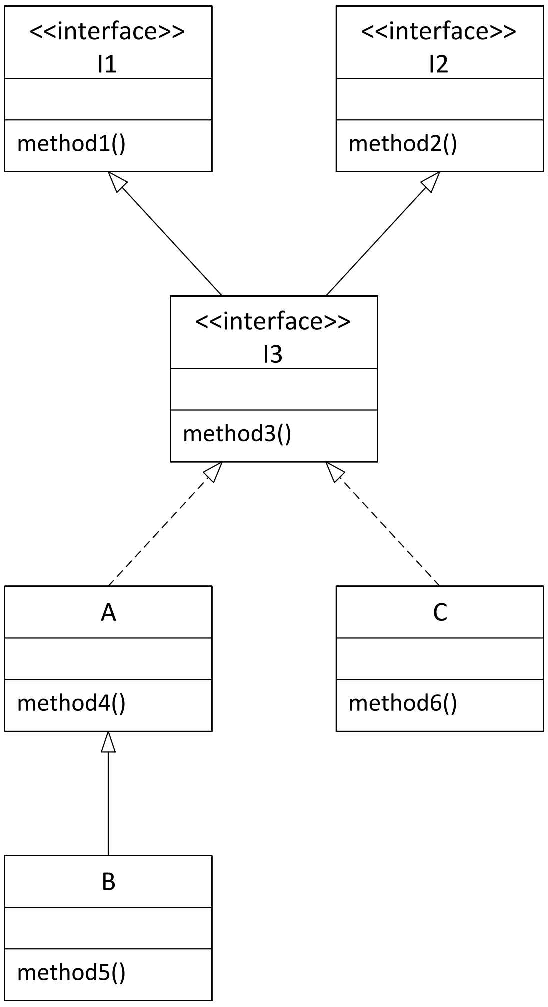

Figure 12.3: Some Interfaces and Concrete Classes

`I1` and `I2` are interfaces which declare the methods `method1()` and `method2()`, respectively. `I3` is an interface that inherits from both `I1` and `I2` and declares an additional method, `method3()`. `A`, `B`, and `C` are concrete classes. Both `A` and `C` implement the `I3` interface, providing methods of their own. `B` is a concrete subclass of `A` and adds a method of its own.

Consider the following segment of code based on Figure 12.3:

Line 1:
```
I3 b = new B();
```

Line 2:
```
b.method1();
```

Line 3:
```
b.method3();
```

Line 4:
```
b.method4();
```

We would like to determine if the code above will compile and run successfully. There is no problem in Line 1 since `A` implements `I3` and `B` is a concrete subclass of `A`. This is normal polymorphic assignment with interfaces described in Section 12.8. Now, `I3` extends `I1` and `I2`. Thus, the type `I3` is associated with the methods of `I3` (i.e., `method3()`) as well as those of `I1` and `I2` (`method1()` and `method2()`). Since `b` is of type `I3`, it is legal to invoke `method1()` on `b` as well as `method3()`. Thus, there is no problem in Line 2 and Line 3.

Finally, it should be noted that `method4()` is a valid operation on an instance of `B` since `B` inherits this method from class `A`. However, even though the dynamic type of `b` is `B`, only the methods of the *static* type of `b` (i.e., `I3`) can be used. Since `method4()` is not present in `I3` (or any of its inherited interfaces), Line 4 will not compile.

Consider also the following segment of code:

Line 1:
```
I1 c = new C();
```

Line 2:
```
I2 i = (I2) c;
```

Now, `C` implements `I3` which extends `I1` and `I2`. So, `C` must implement the methods of `I2` and `I2` since it is a concrete class. By the polymorphic assignment of interface types, it is legal to assign a concrete instance of `C` to one of its interface types such as `I1`. So, Line 1 will compile and run. At run time, the dynamic type of `c` is `C`. Since `C` implements `I1`, `I2,` and `I3`, it is legal to cast `c` to one of the interface types such as `I2`. So, Line 2 will also compile and run.

Exercises

1. What does it mean to implement an interface?

2. What are the benefits of an interface?

3. Discuss two situations where it might be better to use an interface instead of an abstract class.

4. A certain interface declares three methods. `A` is an abstract class which implements the interface and `C` is a concrete class which also implements the interface. Is it necessary to implement the three methods in both `A` and `C`? Explain.

5. A class `C` implements an interface `I`. What are three types of object variables that can be used to refer to an instance of `C`? Are there any other possibilities?

6. A concrete class `C` implements the `Comparable` interface. Suppose the implementation of the `compareTo()` methods returns zero when a certain object is passed as an argument to the method. Why is it a good idea to override the `equals()` method in `C`? Explain how the `equals()` method should be implemented.

7. Two interfaces `I1` and `I2` declare a method with the same signature. A class `C` intends to implement the two interfaces. Investigate if there will be any problem writing the code for `C`, given the common method in both interfaces. What conclusion can you make about situations like these?

8. Chapter 9 explains how to use delegation to achieve a limited form of multiple inheritance in an object-oriented application. Compare this approach with implementing multiple interfaces.