CHAPTER

19

INTERFACES

In the last chapter, we learned that an abstract class is a class which contains some abstract methods as well as concrete methods also. Imagine there is a class that contains only abstract methods and there are no concrete methods. It becomes an interface. This means an interface contains methods which are all abstract and hence none of the methods will have body. Only method prototypes will be written in the interface. So an interface can be defined as a specification of method prototypes. Since, we write only abstract methods in the interface, there is possibility for providing different implementations (body) for those abstract methods depending on the requirements of objects.

Interface

An interface contains only abstract methods which are all incomplete methods. So it is not possible to create an object to an interface. In this case, we can create separate classes where we can implement all the methods of the interface. These classes are called implementation classes. Since, implementation classes will have all the methods with body, it is possible to create objects to the implementation classes. The flexibility lies in the fact that every implementation class can have its own implementation of the abstract methods of the interface. See Figure 19.1.

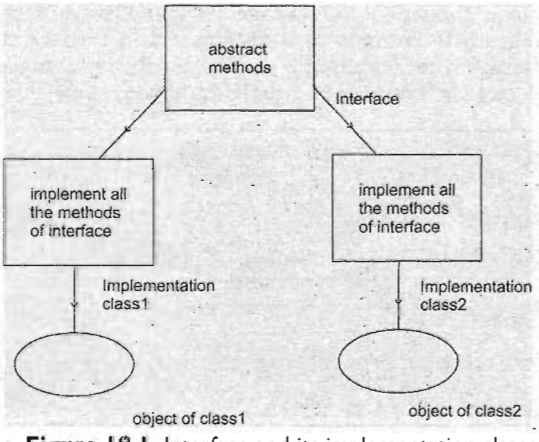

Figure 19.1 Interface and its implementation classes

Important Interview Question

What is an interface?

An interface is a specification of method prototypes. All the methods of the interface are public and abstract.

Since, none of the methods have body in the interface, we may tend to think that writing an interface is mere waste. This is not correct. In fact, an interface is more useful when compared to the class owing to its flexibility of providing necessary implementation for the objects. Let us elucidate this point further with an example. You have some rupees in your hands. You can spend in rupees only by going to a shop where billing is done in rupees. Suppose you have gone to a shop where only dollars are accepted, you can not use your rupees there. This money is like a 'class'. A class satisfies the only requirement intended for it. It is not useful to handle a different situation.

Suppose, you have a credit card. Now, you can pay by using your credit card in rupees in a shop. If you go to another shop where they expect you to pay in dollars, you can pay in dollars. The same credit card can be used to pay in pounds also. Here, the credit card is like an interface which performs several tasks. In fact, the credit card is a plastic card and does not hold any money physically. It contains just your name, your bank name and perhaps some number. But how the shop keepers are able to draw the money from the credit card? Behind the credit card, you got your bank account which holds the money from where it is transferred to the shop keepers. This bank account can be taken as an implementation class which actually performs the task. See Figure 19.2.

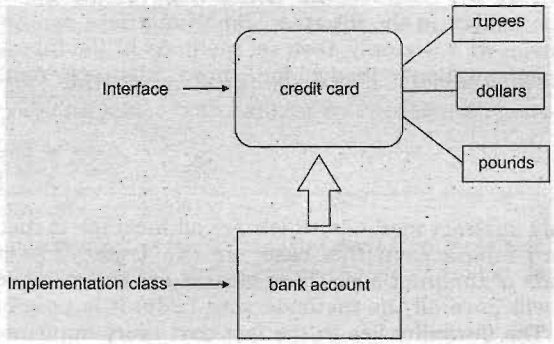

Figure 19.2 Interface and Implementation class

Let us see how the interface concept is advantageous in software development. A programmer is asked to write a Java program to connect to a database and retrieve the data from the database, process the data and display the results in the form of some reports. For this purpose, the programmer has written a class to connect to Oracle database, some thing like this:

```
//class to connect and disconnect from oracle
class Myclass
{
      void connect()
      {
            //write code to connect to oracle database
      }

      void disconnect()
      {
            //disconnect from oracle database
      }
}
```

This class has a limitation. It can connect only to Oracle database. If a client (user) using any other database (for example, Sybase database) uses this code to connect to his database, this code will not work. So, the programmer is asked to design his code in such a way that it is used to connect to any database in the world. How is it possible?

One way is to write several classes, each to connect to a particular database. Thus, considering all the databases available in the world, the programmer has to write a lot of classes. This takes a lot of time and effort. Even though, the programmer spends a lot of time and writes all the classes, by the time the software is released into the market, all the versions of the databases will change and the programmer is supposed to rewrite the classes again to suit the latest versions of databases. This is very cumbersome.

Interface helps to solve this problem. The programmer writes an interface with abstract methods as shown here:

```
interface MyInter
{
      //All the methods of an interface are by default public and abstract.
      //So, we need not declare them explicitly as public abstract.
      void connect();
      void disconnect();
}
```

Important Interview Question

Why the methods of interface are public and abstract by default?

Interface methods are public since they should be available to third party vendors to provide implementation. They are abstract because their implementation is left for third party vendors.

We cannot create objects to the interface. So, we need implementation classes where all these methods of the interface are implemented to suit the needs of objects. So, the programmer releases a notification inviting the programmers of all other companies (third party vendors) to provide implementation classes for his `MyInter` interface. He also specifies in the terms of usage to the user that the user should purchase the implementation class related to his database and use it along with his program.

Now, the third party vendors will provide implementation classes to MyInter interface. These implementation classes are also called database drivers. For example, Oracle Corp people may provide an implementation class (driver) where the code related to connecting to the oracle database and disconnecting from the database will be provided, as:

```
class OracleDB implements MyInter
{
      public void connect()
      {
            //code to connect to oracle database
      }

      public void disconnect()
      {
            //code to disconnect from oracle
      }
}
```

Note that OracleDB is an implementation class of MyInter interface. `implements` keyword should be used with implementation class, as:

```
class Classname implements Interfacename
```

Similarly, the Sybase people may provide another implementation class SybaseDB, where code related to connect to Sybase database and disconnect from Sybase will be provided, as:

```
class SybaseDB implements MyInter
{
      public void connect()
      {
            //code to connect to sybase database
      }

      public void disconnect()
      {
            //code to disconnect from sybase
      }
}
```

Now, it is possible to create objects to the implementation classes and call the `connect()` method and `disconnect()` methods from a main program. This program is also written by the same programmer who develops the interface. Since the programmer does not know the names of implementation classes at the time of writing the main program, he should create object to implementation class without knowing its name. For this, the following steps can be followed:

- Take the driver name (e.g., OracleDB) from the client (user) through command line arguments and pass it to `forName()` method. This method belongs to the class 'Class' which is in `java.lang` package. `forName()` accepts the name of a class as a string, creates an object and stores the class name in that object. This object belongs to the class `Class`, as:

```
Class c = Class.forName(args[0]);
```

Simply speaking, the object 'c' preceding contains the driver name (e.g., OracleDB).

- Now create an object to the class (e.g., OracleDB) whose name is in c. This can be done by `newInstance()` method of the class `Class`. We can create a reference to the interface MyInter which can be used to refer to this object.

```
MyInter mi = (MyInter)c.newInstance();
```

Simply speaking, mi refers to the object of driver class (e.g., OracleDB). This object is created by `newInstance()` method as an object of Object class type. To use MyInter reference to refer to this object, the object type should be converted as MyInter type by using casting.

- Call the `connect()` and `disconnect()` methods of the driver class using mi, as:

```
mi.connect();
mi.disconnect();
```

`connect()` method establishes connection with the particular database and `disconnect()` method disconnects from the database. The complete program is shown in Program 1.

Program 1: Write a program to create an interface MyInter that connects to a database and retrieves the data from the database.

```
//interface example - Connecting to any Database
interface MyInter
{
      void connect(); //abstract public
      void disconnect();
}

class OracleDB implements MyInter
{
```

```
        public void connect()
        {
                System.out.println("Connecting to Oracle database...");
        }
        public void disconnect()
        {
                System.out.println("Disconnected from Oracle.");
        }
}
class SybaseDB implements MyInter
{
        public void connect()
        {
                System.out.println("Connecting to Sybase database...");
        }
        public void disconnect()
        {
                System.out.println("Disconnected from Sybase.");
        }
}
class InterfaceDemo
{
        public static void main(String args[ ]) throws Exception
        {
                //accept the implementation classname from Commandline argument
                //and store it in the object c.
                Class c = Class.forName(args[0]);

                //create an object to the class whose name is in c.
                //let the reference variable of interface point to it.
                MyInter mi = (MyInter)c.newInstance();
                //call methods of the object using mi.
                mi.connect();
                mi.disconnect();
        }
}
```

Output:

```
C:\> javac InterfaceDemo.java
C:\> java InterfaceDemo OracleDB
Connecting to Oracle database...
Disconnected from Oracle.
C:\> java InterfaceDemo SybaseDB
Connecting to Sybase database...
Disconnected from Sybase.
```

In preceding program, the implementation classes `OracleDB` and `SybaseDB` are supposed to be provided by third party vendors. An object to `OracleDB` or `SybaseDB` is created and `connect()` and `disconnect()` methods are called by using MyInter reference mi. There are two more concerns in the preceding program. The first one, why third party vendors provide implementation classes for our interface? They definitely provide implementation classes because it is profitable for them. Every client who ever purchases our software (interface) will definitely purchase the implementation classes from the third party vendor because they want to use our software in connection with the database. Thus, the third party vendors can sell the implementation classes. Also, the database vendor can promote their business by providing the implementation classes.

The second concern is how the third party vendors know which methods to implement? The third party people know which interface we wrote and which methods were written there along with their description through API (Application Programming Interface) document. API document is a hyper text mark up language (html) file that contains the description of all the features of a software, a product or a technology. After creating the software, we create an API document which works like a

reference manual to the third party vendor to understand which methods to implement. ... document creation is explained in Chapter 20. This entire discussion is represented in Figure 19...

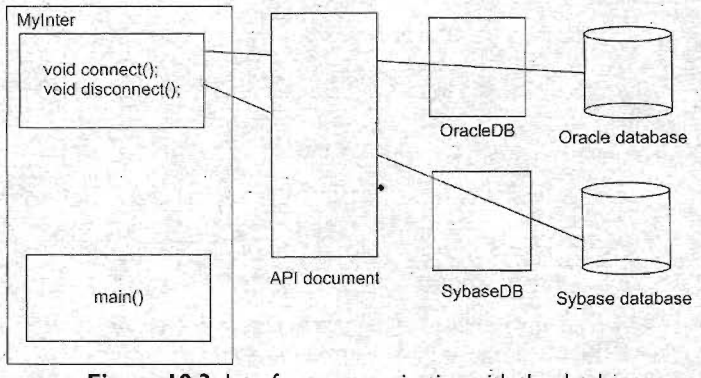

Figure 19.3 Interface communicating with the databases

Let us take another example where interface is used. We want to write Printer interface ... used to send data to different printers. This interface has a method `printit()` that sends ... the printer and `disconnect()` method that disconnects the printer after printing is ... course, this program is a model how the printer interface can be used. It does not send the ... real printer. On the other hand, it displays the text on the screen.

Printer interface is implemented by IBM people such that it sends text to IBM printer. ... Epson people provide a different implementation to the Printer interface such that it sends ... Epson printer. These implementation classes (printer drivers) are written as IBMPrinter ... EpsonPrinter.

To use a printer, first of all we should know which printer is used by the client. We assume ... printer driver name is generally stored in the `config.txt` file at the time of installing the ... (implementation class). So, open notepad (or any text editor) and create a file with the ... `config.txt` and store a single line that represents the printer driver name, as:

```
EpsonPrinter
```

And then save the file. Alternately, you can store the name IBMPrinter in the `config.txt` file.

Now, this `config.txt` file should be connected to an input stream which can read data ... file. We can use `FileReader` for this purpose. We can connect another stream ... `LineNumberReader` to read line by line from the stream. This stream can read the first line ... in `config.txt` file. This line will be, of course, the line stored by us, i.e., `EpsonPrinter`.

Store the printer name 'EpsonPrinter' in an object using `Class.forName()` method. ... EpsonPrinter as a class name (which represents implementation class), create another object ... by using `newInstance()` method. Using this object, we can call and use the methods of ... interface.

Program 2: Write a program which contains a Printer interface and its implementation class ... send text to any printer.

```
//An interface Printer to send text to any printer
import java.io.*;

//creating an interface for printing
 interface Printer
{
     //to print the text sent to printer
```

```
        void printit(String text);  //public abstract

        //to disconnect from printer
        void disconnect();
}

//Implementing Printer interface for IBM printer
 class IBMPrinter implements Printer
{
        public void printit(String text)
        {
                System.out.println(text);
        }
        public void disconnect()
        {
                System.out.println("Printing completed");
                System.out.println("Disconnected from IBM Printer");
        }
}

//Implementing Printer interface for Epson printer
 class EpsonPrinter implements Printer
{
        public void printit(String text)
        {
                System.out.println(text);
        }
        public void disconnect()
        {
                System.out.println("Printing completed");
                System.out.println("Disconnected from Epson printer");
        }
}

//Using a printer
 class UsePrinter
{
        public static void main(String args[ ]) throws Exception
        {
                //attach FileReader to config.txt file to read data from it
                FileReader fr = new FileReader("config.txt");

                //connect LineNumberReader to FileReader to read one line at a time
                LineNumberReader lnr= new LineNumberReader(fr);

                //read the first line from config.txt file
                String printername =  lnr.readLine();

                //The read line represents the printer name
                System.out.println("Loading the driver for: "+printername);

                //store the printername in an object c
                Class c =Class.forName(printername);

                //create an object to that class represented by printername in c
                Printer ref = (Printer)c.newInstance();

                //send text to printit using Printer reference
                ref.printit("Hello, This is printed on the printer");

                //disconnect afeter printing
                ref.disconnect();
        }
}
```

Output:

```
C:\> javac UsePrinter.java
C:\> java UsePrinter
Loading the driver for: EpsonPrinter
Hello, This is printed on the printer
Printing completed
Disconnected from Epson printer
```

Summary

Now, let us summarize the following points on interfaces, before we proceed further:

- An interface is a specification of method prototypes. This means, only method [illegible] written in the interface without method bodies.
- An interface will have 0 or more abstract methods which are all public and abstract by [illegible]
- An interface can have variables which are public static and final by default. This means [illegible] variables of the interface are constants.
- None of the methods in Interface can be private, protected or static.
- We cannot create an object to an interface, but we can create a reference of interface type [illegible]
- All the methods of interface should be implemented in its implementation classes. [illegible] method is not implemented, then that implementation class should be declared as 'abstract'.
- Interface reference can refer to the objects of its implementation classes.
- When an interface is written, any third party vendor can provide implementation classes [illegible]
- An interface can extend another interface.
- An interface cannot implement another interface.
- It is possible to write a class within an interface.
- Interface forces the implementation classes to implement all of its methods compulsorily. [illegible] compiler checks whether all the methods are implemented in the implementation class or not.
- A class can implement (not extend) multiple interfaces. For example, we can write:

```
class Myclass implements Interface1, Interface2,…
class Myclass extends Class1 implements Interface1, Interface2
```

Important Interview Question

Can you implement one interface from another?

No, we can't. Implementing an interface means writing body for the methods. This can not be done again in an interface, since none of the methods of the interface can have body.

Can you write a class within an interface?

Yes, it is possible to write a class within an interface.

Multiple Inheritance using Interfaces

We know that in multiple inheritance, sub classes are derived from multiple super classes. If super classes have same names for their members (variables and methods) then which member is inherited into the sub class is the main confusion in multiple inheritance. This is the reason why Java does not support the concept of multiple inheritance. This confusion is reduced by using multiple interfaces to achieve multiple inheritance. Let us take two interfaces as:

```
interface A
{
      int X = 20;  //public static final
      void method();  //public abstract
}

 interface B
{
      int X= 30;
      void method();
}
```

And there is an implementation class Myclass as:

```
class Myclass implements A,B
```

Now, there is no confusion to refer to any of the members of the interfaces from Myclass. For example, to refer to interface A's X, we can write:

```
A.X
```

And to refer to interface B's X, we can write:

```
B.X
```

Similarly, there will not be any confusion regarding which method is available to the implementation class, since both the methods in the interfaces do not have body, and the body is provided in the implementation class, i.e., Myclass.

The way to achieve multiple inheritance by using interfaces is shown in Program 3. In this program, interface Father has a constant HT which represents the height of father and interface Mother has another constant HT which represents the height of mother. Both the interfaces have an abstract method `height()`. If Child is the implementation class of these interfaces, we can write:

```
class Child implements Father, Mother
```

Now, in `Child` class, we can use members of Father and Mother interfaces without any confusion. The `height()` method can be implemented in Child class to calculate child's height which we assume being the average height of both of its parents.

Program 3: Write a program to illustrate how to achieve multiple inheritance using multiple interfaces.

```
//Multiple inheritance using interfaces
interface Father
{
      float HT = 6.2f;
      void height();
}
interface Mother
{
      float HT = 5.8f;
      void height();
}
class Child implements Father, Mother
{
      public void height()
      {
            //child got average height of its parents
            float ht = (Father.HT+ Mother.HT)/2;
            System.out.println("Child's height= "+ ht);
```

```
        }
}
class Multi
{
        public static void main(String args[ ])
        {
                Child ch = new Child();
                ch.height();
        }
}
```

Output:

```
C:\> javac Multi.java
C:\> java Multi
Child's height= 6.0
```

Abstract Classes vs. Interfaces

It is the discretion of the programmer to decide when to use an abstract class and when to go for an interface. Generally, abstract class is written when there are some common features shared by all the objects as they are. For example, take a class WholeSaler which represents a whole sale shop with text books and stationery like pens, papers and note books, as:

```
class WholeSaler
{
        void text_books()
        {
                //text books of X class
        }
        void stationery();  //this can be pens, papers or note books.
}
```

Let us take Retailer1, a class which represents a retail shop. Retailer1 wants text books of X class and some pens. Similarly, Retailer2 also wants text books of X class and some papers. In this case we can understand that the `void text_books()` is the common feature shared by both the retailers. But the stationery asked by the retailers is different. This means, the stationery has different implementations for different retailers but there is a common feature, i.e., the text books. So in this case, the programmer designs the `WholeSaler` class as an abstract class. Retailer1 and Retailer2 are sub classes.

On the other hand, the programmer uses an interface if all the features need to be implemented differently for different objects. Suppose, Retailer1 asks for VII class text books and Retailer2 asks for X class text books, then even the `text_books()` method of WholeSaler class needs different implementations depending on the retailer. It means, the `void text_books()` method and also `void stationery()` methods should be implemented differently depending on the retailer. So, in this case, the programmer designs the `WholeSaler` as an interface and Retailer1 and Retailer2 become implementation classes.

There is a responsibility for the programmer to provide the sub classes whenever he writes an abstract class. This means the same development team should provide the sub classes for the abstract class. But if an interface is written, any third party vendor will take the responsibility of providing implementation classes. This means, the programmer prefers to write an interface when he wants to leave the implementation part to the third party vendors.

In case of an interface, every time a method is called, JVM should search for the method in the implementation classes which are installed elsewhere in the system and then execute the method. This takes more time. But when an abstract class is written, since the common methods are defined within the abstract class and the sub classes are generally in the same place along with the

software, JVM will not have that much overhead to execute a method. Hence, interfaces are slow when compared to abstract classes.

From this discussion, you can understand that it is possible to convert an abstract class into an interface and vice versa.

Complete differences between an abstract class and interface are provided here.

Table 19.1

Abstract class	Interface
1. An abstract class is written when there are some common features shared by all the objects.	An interface is written when all the features are implemented differently in different objects.
2. When an abstract class is written, it is the duty of the programmer to provide sub classes to it.	An interface is written when the programmer wants to leave the implementation to the third party vendors.
3. An abstract class contains some abstract methods and also some concrete methods.	An interface contains only abstract methods.
4. An abstract class can contain instance variables also.	An interface can not contain instance variables. It contains only constants.
5. All the abstract methods of the abstract class should be implemented in its sub classes.	All the (abstract) methods of the interface should be implemented in its implementation classes.
6. Abstract class is declared by using the keyword `abstract`.	Interface is declared using the keyword `interface`.

Important Interview Question

What is the difference between an abstract class and an interface?

See the Table 19.1.

Conclusion

Interfaces are very important, especially when the programmer wants to customize the features of the software differently for different objects. In this case, an interface should be written so that the features will not have any specific implementation. These features are then implemented in the implementation classes by the third party vendors according to the task to be performed in a particular context. The third party vendors get the description of features of the interface from a document, called API document. So, it is the duty of the programmer to create an API document after completion of his software.