CHAPTER 7

Abstract Classes and Interfaces

In Chapter 6, we have looked at the keyword `final`, let us now look at another keyword in Java called `abstract`. The keyword `abstract` in a sense is opposite of `final`. It is a modifier only for methods and classes. The modifiers `abstract` and `final` cannot be used together. A method or a class cannot be declared to be both `abstract` and `final`. A `final` method prevents overriding, whereas an `abstract` method promotes overriding. A `final` class prevents inheritance, whereas an `abstract` class promotes inheritance.

7.1 ABSTRACT CLASSES AND ABSTRACT METHODS

What is the meaning of declaring a method as `abstract`. When we declare a method as `abstract`, then we do not provide an implementation of the method, and the method would be required to be overridden in a non-abstract sub-class. A non-abstract class cannot have any abstract methods, whether inherited or declared.

An abstract class is a class that is like all normal classes, but has two differences. One, an application cannot create an instance of this class. Only instances of its non-abstract sub-classes can be created. Two, an abstract class can have abstract methods, which are not allowed in a non-abstract class.

Let us try to understand the use of abstract methods. Methods are like the operators of a data type. Now, if we look at the specification for the C language, we find that it says that the arithmetic operators are defined for all the numeric types. There is no type called numeric type, which is being used directly, but byte, short, int, long, float, double and char are the numeric types. These are the sub-types of the numeric type. Here numeric type is an abstract type, which has the operators like addition, subtraction, multiplication and division. The specification for the numeric type does not specify how these operations are to be performed, it is simply a specification that such operators need to be supported by any sub-type (kind) of numeric type. There will be different ways of doing addition, subtraction, multiplication and division for different kinds of numeric types. The addition for integral types differs from the way addition is done for the floating point types. Similarly, an abstract class is an abstract data type (like the numeric type above), which can have abstract methods (like addition, subtraction, etc. above), these abstract methods are simply declaration, and does not have any implementation.

Let us take another example of an abstract type. Let us say we have a class called `Vehicle`, then an instance of `Vehicle` is never created. The instances created are of some sub-class of `Vehicle`. Every `Vehicle` can have a method called `start()`, `run()`, `brake()`, etc. But how to start a vehicle cannot be specified in the `Vehicle` class itself. There is no implementation of these methods in the `Vehicle` class. It has just a declaration about the return type and the kind of parameters required for the method. Each sub-class will have its own different implementation of the start, run and brake methods. So, these methods would be specified as `abstract` in the `Vehicle` class, indicating that all non-abstract sub-classes of vehicles must have these methods implemented.

There are two cases in Java where a method is simply declared and implementation is not written. One is when the method is declared as `abstract`, and the other is when the method is declared as native. The native methods are methods whose implementation is not written in Java, but is written in some native language, mostly C or C++.

```
abstract class Vehicle {  // an abstract class may have abstract
      methods.
      abstract void start();  // abstract methods do not have an
            implementation.
      abstract void run();
      abstract void brake();
      ...
}
```

In the class hierarchy given in Figure 7.1, we have `Vehicle` as a super-class, and since this class has abstract methods, this class would have to be declared as `abstract`. It has a sub-class called `Car`. The class `Car` may not declare any new abstract methods and may implement some of the abstract methods from the super-class, but if it has still some inherited abstract methods (abstract method not overridden), then the class `Car` would also have to be declared as `abstract`. Now let us say that we have the sub-class `MarutiCar`, which is a sub-class of `Car`, and it may not have any abstract methods declared, and it overrides all the abstract methods from the super-class to be non-abstract. So, the class `MarutiCar`, does not have any abstract methods of any form, but we may still decide to declare it `abstract`, since we would not like anyone to directly create instances of this class. Instances of `MarutiCar` are not created; what are created are instances of various models like the `Maruti800`, which is a type of `MarutiCar`. Perhaps, `Maruti800`, which is a basic model simply inherits all the functionality from the super-class and may not even add or override methods from the super-class, and the overriding or adding of new methods may be done in other sub-classes like `MarutiZen`, etc. Here we do not create instances of `MarutiCar`, but we do create instances of `Maruti800`, `MarutiZen` and other types of `MarutiCar`; so it is necessary that the classes `Maruti800`, `MarutiZen` and other types of `MarutiCar` do not have any abstract methods. Therefore, the following conditions can be summarized for declaring a class as `abstract`:

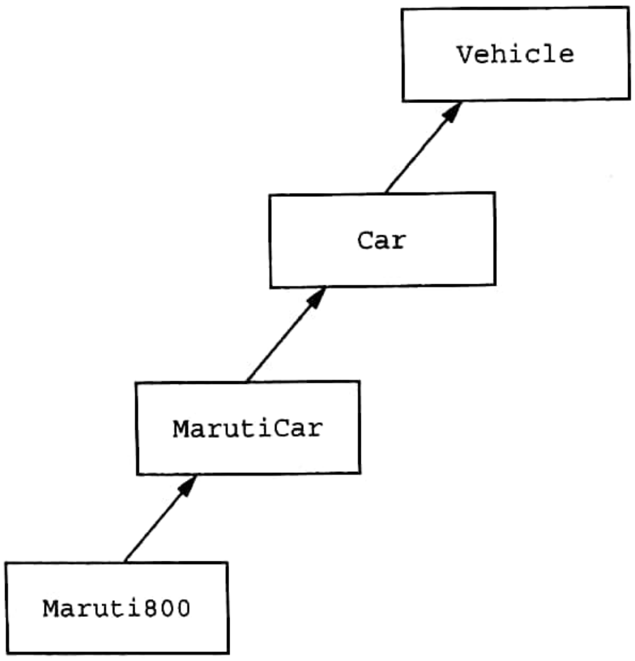

Figure 7.1 Sub-classes of Vehicle class

1. A class would be `abstract` if we have declared some abstract methods in a class. A non-abstract class cannot have any `abstract` methods.
2. A class would be `abstract` even if there are no abstract methods declared in the class, but it has some abstract methods that have not been overridden to be non-abstract.
3. A class may be declared to be `abstract` even if the class has no `abstract` methods declared, and there are no `abstract` methods that have been inherited. This may be done to prevent creating instances of the class and it enforces the creation of sub-classes.

Even though we do not create instances of an `abstract` class, still, an `abstract` class can always have a constructor. The constructors of an `abstract` class will be used from the constructor of its sub-classes.

A common question at this stage is what is the use of an `abstract` method and why declare a method when we are not writing an implementation of the method? Let us look at an example where an `abstract` method is useful. Let us say we have an `abstract` class `Vehicle` as given below:

```
abstract class Vehicle {
    abstract double getLoad(); // a method return the load of a vehicle
    abstract void start();
    abstract void run();
    abstract void brake();
}
```

and we also have a class called `Bridge` as given below:

```
class Bridge {
    double maxLoad;
    Bridge(double maxLoad) {
        this.maxLoad = maxLoad;
    }
    boolean canCross(Vehicle v) {
        return v.getLoad() < maxLoad;
    }
}
```

In the above example, whenever we create a `Bridge` instance, we would have to specify the `maxLoad` that can go over the `Bridge`. Now, whenever a `Vehicle` approaches the `Bridge`, it may first check whether the `Vehicle` is allowed to go over the `Bridge` by checking with the `canCross()` method of the `Bridge` class. The `canCross()` method uses the `getLoad()` method from the `Vehicle`. If we did not have the `abstract` method `getLoad()` in the `Vehicle` then the `canCross()` method would have to be overloaded for each non-abstract sub-class of the `Vehicle` class where the `getLoad()` method would be defined. Here, it is sufficient to know that any `Vehicle` instance would always have a method `getLoad()`, since an instance can only be created from a non-abstract class, so it would also imply that an implementation of the `getLoad()` method would always be available for any instance of a non-abstract sub-class of `Vehicle`.

EXERCISE 7.1 Declare the `Account` class of Exercise 6.1 to be `abstract`. Also recompile the `TestAccount` class to test that the instances of `Account` are not created directly, i.e. new `Account(...)`, should give a compilation error.

The updated `Account` class is now given in Listing 7.1.

Listing 7.1. `Account` class as abstract class

```

abstract class Account {

    int accountNumber;
    String name;
    double balance;

    Account(int acno, String n, double openBal) {
        this.accountNumber = acno;
        this.name = n;
        this.balance = openBal;
    }

    int getAccountNumber() {
        return this.accountNumber;
    }

    String getName() {
        return this.name;
    }

    double getBalance() {
        return this.balance;
    }

    void deposit(double amt) {
        this.balance += amt;
    }

    boolean withdraw(double amt) {
        if (this.balance < amt) {
            return false;
        }
        this.balance -= amt;
        return true;
    }

    void display() {
        System.out.println("Account:"+this.accountNumber+","+this.
            name+","+this.balance);
    }

    static int lastAccountNumber = 1000;

    Account(String n, double openBal) {
        this(++lastAccountNumber, n, openBal);
    }
}
```

7.2 SINGLE INHERITANCE OF CLASSES

In Java, we do not have multiple inheritance of classes. A class cannot have more than one direct super-class. When we define the class, we can specify only one class name in the `extends` clause of the class definition.

Now looking at the initial class inheritance given for the `Vehicle` class, we also identify another sub-class of `Vehicle` class called `Aeroplane`. Now this class is not only a `Vehicle`, but it is also a `FlyingObject`. There are many other things that are flying objects.

So here, we have `Aeroplane`, which has two direct super-types. One, `Vehicle` and another, `FlyingObject`. What is a `FlyingObject`? `FlyingObject` is anything that can fly. It is identified by the functionality which it must support. There are many things which can be `FlyingObject`, but, each type of `FlyingObject` will have its own implementation of the `fly()` method, so, the `fly()` method in the `FlyingObject` will be `abstract`. We cannot implement the `fly()` method for the `FlyingObject`, also, we cannot have any instance variables in the `FlyingObject`; different types of `FlyingObject` will use different things for flying. Some may use wings, and yet some may not be using wings. There is no structure available for `FlyingObject`. `FlyingObject` will be a type, which has no instance variables and has only `abstract` method (here it has the `fly()` method as an abstract method). When

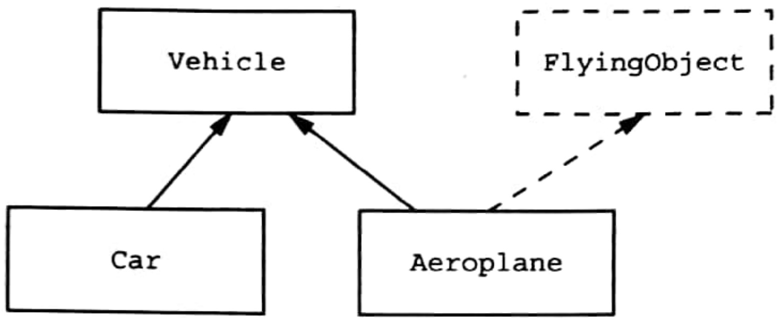

Figure 7.2 Sub-classes of `Vehicle` class

we have such data types that have no instance variables and only `abstract` methods (no, non-abstract methods), then these types are defined as interfaces (Figure 7.2).

7.3 INTERFACES is a block of code in Java

What is the difference between an `abstract` class and an interface? An interface is a special case of an `abstract` class, which can only have `abstract` methods and static constants and can be used in multiple inheritance of types. Out of the various members for a class which we know about, only `abstract` methods and static constants are allowed in an interface. There are a few other members which can also go in an interface, which we will talk about when we look at the nested types in Chapter 12. An interface cannot contain any kind of method implementations (non-abstract methods). All the members in an interface are always `public`. We will be looking at the meaning of various access specifiers in Chapter 9. An interface for the `FlyingObject` may be defined as follows:

```
interface FlyingObject {
    void fly();
}
```

All interfaces are implicitly `abstract` and we need not explicitly declare an interface to be `abstract`. Also any method declaration in an interface is always `public` and `abstract`, we need not explicitly declare them as `abstract`. When we declare a variable in an interface then it is implicitly `public static` and `final`, it may not be explicitly declared so. Therefore, the above definition of `FlyingObject` may also have been declared as given below:

```
abstract interface FlyingObject {
    public abstract void fly();
}
```

The inheritance rules for classes and interfaces are as follows:

1. A class can inherit from only one class, but can inherit multiple types from multiple interfaces, which are mentioned using the `implements` clause.

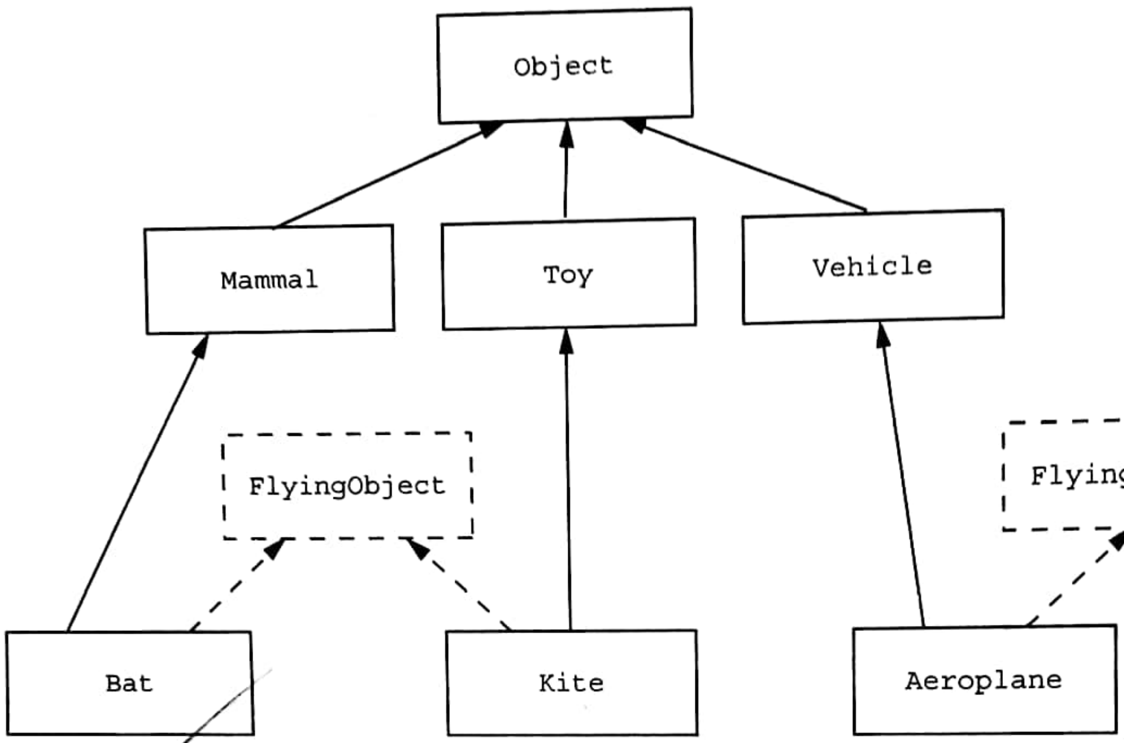

Figure 7.3 Interface FlyingObject implemented by unrelated classes

2. An interface can inherit multiple types only from other interfaces, it cannot inherit from a class.
3. When we do not mention a super-class in a class definition using the extends keyword, then the Object class (which we will look at, in Chapter 8) is considered as the super-class.

Thus, the inheritance relation between classes forms a tree structure, with Object class as the root, and the interfaces are some kind of specification, which can be put anywhere in the hierarchy of the classes. Therefore, for example we have FlyingObject interface being implemented, in totally unrelated classes.

Here, the FlyingObject is implemented by different types of classes, which are kind of Mammals, Toys or Vehicle, etc. Interface is like a pure abstract class, which can never have any kind of implementation (non-abstract methods) (Figure 7.3). Note that static methods cannot be declared to be abstract. In Java we do not have multiple inheritance of classes, but we have multiple inheritance of data types using interfaces. Some languages allow multiple inheritance of classes but then they also have cases, which can lead to ambiguity in case of inheritance. There is also a very famous diamond problem, related to multiple inheritance of classes. In case of Java, since the interfaces do not have implementation, and a class can have only one direct super-class, it would not be possible to inherit more than one kind of implementation of any method from the super-class. Abstract methods may be inherited from multiple interfaces that the class implements. Only abstract classes can inherit abstract methods from an interface, a non-abstract class needs to override all the abstract methods declared in the interface that it implements, such an overriding is allowed to be inherited from the super-class.

Interfaces are also used to check if a certain operation can be performed on an instance or not; example, Cloneable interface, which is discussed in Chapter 8.

Another very good use of interface is the Comparable interface, which is discussed in Chapter 10.

MISCONCEPTION NOTE:

Misconception Abstract class must have at least one abstract method.

Fact Any class may be declared to be abstract to prevent instantiation of the class without sub-classing.

Misconception Abstract class cannot have a constructor.

Fact Every class has at least one constructor. The constructor in an abstract class will be used from the constructors of its sub-class.

LESSONS LEARNED

- An abstract class cannot be instantiated, only its non-abstract sub-class can be instantiated.
- Non-abstract classes cannot have any kind of abstract methods, either declared or inherited. It needs to override all the abstract methods in its super-class.
- A class cannot inherit directly from more than one super-class, but can have more than one super-interfaces. The keyword `extends` is used to inherit from one super-class, whereas the keyword `implements` is to indicate the super-interfaces for a class.
- An interface cannot inherit from a class, but can inherit from more than one interface. The keyword `extends` is used by an interface to inherit from other interfaces.
- An interface can have any non-abstract methods, instance variables and blocks. It can only have abstract methods and `static` constant(`final`) variables. All members in an interface are always `public`.

EXERCISES

1. State which of the following are true or false:
   (a) An `abstract` class must have at least one `abstract` method.
   (b) Constructors cannot be defined in an `abstract` class.
   (c) An interface can contain instance variable declarations.
   (d) An interface in Java can inherit from at the most one non-final class.
   (e) `abstract` methods may not be declared in a non-abstract class, but may be inherited from an abstract super-class.
   (f) Multiple inheritance of types is not supported by Java.
2. Fill in the blanks in the following:
   (a) __________ keyword is used by an interface to inherit from multiple interfaces.
   (b) The constructors of __________ classes cannot be used to create instances of the given class.

(c) If a class contains one or more abstract methods, it is an abstract class.

(d) A concrete (non-abstract) class cannot have abstract methods.

(e) Implements keyword is used in a class definition to inherit multiple interface types.

3. Explain the uses of the keywords `abstract` and `final`.
4. Differentiate between abstract classes and interfaces.
5. Explain each of the following terms: inheritance, multiple inheritance, `interface`, super-class and sub-class.
6. Explain the diamond problem associated with the programming languages supporting multiple inheritance of classes.
7. Explain the difference between inheritance of types by interfaces and the inheritance from a class.