CHAPTER 8
The `Object` Class

In Chapter 7, we have seen that there is single inheritance of classes in Java, i.e. a class cannot have more than one direct super-class. Every class in Java has a direct super-class, if we do not declare a super-class while defining a class then the compiler assumes the `Object` class to be the super-class. Only the `Object` class does not have any super-class. In Java any instance which is created has certain minimum behaviour. When an instance of any reference type is created, it has certain methods always available. These are the methods which are inherited from the `Object` class. These methods are also available on instances of arrays. `Object` class is the root of the class hierarchy, defining the minimum set of methods which should be available on any instance in the JVM. `Object` is the super-type for all reference types, i.e. any reference type is assignable to `Object`.

8.1 THE `Object` CLASS AS THE SUPER-CLASS OF ALL CLASSES

Let's look at The `Rectangle` class in the earlier example, and try to execute the statement in the application given in Listing 8.1.

Listing 8.1. `Rectangle` class code segment

```
1 Rectangle r1 = new Rectangle(7, 5);
2 System.out.println("r1 is"+ r1);
```

What does it print? Here we are trying to concatenate a `String` with an instance, we know that when + operator is used, and any of the operands is a `String`, then the other operand will be converted to a `String` and it would result in a `String` concatenation. So in this case it would convert the instance of `r1` to a `String`. Converting a `Rectangle` into a String!. Well we see some output which looks like `r1 is Rectangle@xxxxxxxx` where `xxxxxxxx` are some hexadecimal digits. Therefore, we see that the instance of `Rectangle` is converted to `String` as <class-name>@<some-hex-digits>. Let us now also try changing the second line in Listing 8.1 as follows:

```
System.out.println("r1 is"+ r1.toString());
```

That should be a compilation error? We have not yet defined a method called `toString()` in the `Rectangle` class. But we find that this code compiles. What is the reason? This is because the method `toString()` has been inherited by the `Rectangle` class. But we have not mentioned any super-class for `Rectangle` in the class definition of `Rectangle`. Where does it inherit from? It inherits the `toString()` method from a class called `Object`. In Java every instance is an `Object`. When we define a class and do not mention any super-class using the `extends` keyword, then the class would inherit from the `Object` class. So every class in Java except for the `Object` class has a super-class. If we do not mention a super-class using the `extends` keyword, then the compiler would use `Object` class as the super-class. In fact `Object` is the super-type for all reference types. Even the array instances are inheriting from `Object`. In Java everything is `Object` (except for primitive types). So it is important for us to know what is inherited and available for all classes from this `Object` class. All methods of the `Object` class would work on all instances, including the array types as well. All reference types are assignable to a variable of `Object` type, including arrays and interfaces, e.g.

```
1    Object obj;
2    Comparable c = ...; // instance of some class which is
        Comparable.
3    obj = c;            // is valid interface is assignable to obj.
4    obj = new int[10];  // is valid array is assignable to obj
```

8.1.1 Methods Inherited from the `Object` Class

`hashCode`: Every object in Java has an integer value associated with it, which is known as its hashCode value. This method returns the hashCode value of the Object, on which the method is invoked. This method may be overridden. The hashCode value is used by some of the data structures. The method has the following signature:

```
public int hashCode()
```

This method is used in the `Map` implementation classes, which are discussed in Chapter 13. This method is also supposed to be consistent with the `equals()` method discussed later in this chapter.

`toString`: In Java every data type can be converted to `String` (has a String representation). For all instances the `toString()` method is supposed to return the `String` representation of the instance. This method may be overridden to define how instances of a given class should be converted to `String`. The method has the following signature:

```
public String toString()
```

In case of the `Rectangle` class, let us say, we override the `toString()` method as follows:

```
1     public String toString() {
2         return "Rectangle:"+length+"x"+width;
3     }
```

Then the output of Listing 8.1 will be something like `Rectangle:7x5`, which is more meaningful. So for every class, it may be a better idea to override this method to give some meaningful `String` conversion. The `toString()` method which is defined in the `Object` class (this would be inherited in case a class does not override it) returns a `String` giving the name of the class followed by @ then followed by the hashCode value in hexadecimal, e.g. in case we did not override the `toString()` method of the `Rectangle` class, then the output could be `Rectangle@213f3323`. The hashCode value may be different in the output.

`equals`: In Java for the reference types the == operator does not define equality, e.g. If we have two `Rectangle` instances created as follows:

```
1     Rectangle r1, r2;
2     r1 = new Rectangle(7, 5);
3     r2 = new Rectangle(7, 5);
4     System.out.println(r1 == r2);
```

Then the output in the above listing is false. The == operator only tells us whether `r1` and `r2` refer to the same instance or not. To know whether any two objects referred by references a and b are equal or not, the Java API uses the `equals()` method, like a.equals(b). When anyone defines a class it is upto the developer of the class to define the equality operation for instances of the class by overriding the `equals()` method. The default `equals()` method inherited from the `Object` class simply checks if the `Object` is the same or not. It has been implemented in the `Object` class as below:

```
1     public boolean equals(Object obj) {
2         return (this == obj);
3     }
```

Let us define the equality for the instances of `Rectangle`, such that, two Rectangles are equal if their lengths and the widths are equal. In that case, we may override the `equals()` method for the `Rectangle` class as given below:

```
1     public boolean equals(Object obj) {
2         if (!(obj instanceof Rectangle)) {
3             return false;
4         }
5         if (this.length != ((Rectangle)obj).length) {
```

```
6               return false;
7           }
8           if (this.width != ((Rectangle)obj).width) {
9               return false;
10          }
11          return true;
12      }
```

Here the first condition is used to ensure that the instance being compared is a `Rectangle` only, and not any other object. A `Rectangle` instance would not be considered equal with any instance which is not a `Rectangle`.

Similarly we also override the `equals()` method for the `Cuboid` class as below:

```
1       public boolean equals(Object obj) {
2           if (!(obj instanceof Cuboid)) {
3               return false;
4           }
5           if (this.length != ((Cuboid)obj).length) {
6               return false;
7           }
8           if (this.width != ((Cuboid)obj).width) {
9               return false;
10          }
11          if (this.height != ((Cuboid)obj).height) {
12              return false;
13          }
14          return true;
15      }
```

There are a few consistencies which are expected from the implementation of `equals()` method. So when we override the `equals()` method, we try to ensure the following:

1. Given any instance `a`, ensure that a.equals(a) is always `true`.
2. Given any two instances `a` and `b`, a.equals(b) should be the same as b.equals(a).
3. Given instances `a`, `b` and `c`, if a.equals(b) is `true` and b.equals(c) is `true`, then a.equals(c) should also be `true`.

   The `equals()` method is also expected to be consistent with the `hashCode()` method.
4. Given two instances `a` and `b` such that a.equals(b) is `true`, then a.hashCode() == b.hashCode() must be `true`.

   So, normally, whenever we override `equals()` method, we would also override the `hashCode()` method.

Now, let us check whether the `equals()` method, which we have overridden for the `Rectangle` and the `Cuboid` classes are consistent or not. They seem to be consistent, but

check if a.equals(b) is the same as b.equals(a) in all cases. Consider the following code:

```
1    Rectangle r1, r2;
2    r1 = new Rectangle(7, 5);
3    r2 = new Cuboid(7, 5, 4);
4    System.out.println(r1.equals(r2));
5    System.out.println(r2.equals(r1));
```

What is the output? It prints `true` and `false`, i.e. r1.equals(r2) is `true`, while r2.equals(r1) returns `false`. What is the reason for this? The culprit is the `instanceof` operator. In the `Rectangle` class, the `instanceof` operator only checks if the `obj` instance passed as parameter is an instance of the `Rectangle` or not. Here even the `Cuboid` instance also qualifies. The `instanceof` operator does not check the exact class of the Object. Here we are insterested in checking if the `obj` parameter is exactly a `Rectangle` or not and we are not interested in sub-classes of the `Rectangle` class. We want to compare `length` and `width` for exactly a `Rectangle` instance only, and we are not interested in the sub-classes of the `Rectangle` class.

`getClass`: In Java we can get the exact class of an instance by using the `getClass()` method. The method signature is as given below:

```
public final Class getClass()
```

What is the return type for this method? `Class`. What is `Class`? `Class` is the name of a class. Yes, we have a class called `Class` in Java. The instances of this class have information about a class definition. An instance of the `Class` class is not created by using a constructor. When we start the Java runtime by using the Java command, the class loader for the JVM creates instances of the `Class` class for each of the data types, which is used in the application. Every class would be loaded before it can be used. What is meant by loading a class? Loading a class involves the creation of the instance of the `Class` class for the class which is loaded. There is one instance of the `Class` class for each of the data types. When Java starts, it loads certain minimum classes at the startup, whether we use them in our application or not. It would always load the `Object` class, `Class` class, `String` class, `System` class and many such classes. The instance of `Class` encapsulates information about a data type. Normally, a class is loaded only once. So one instance of `Class` is created for each data type, including the primitive types. So, when we invoke a Java application which uses the `Rectangle` and the `Cuboid` classes, then there would be instances of `Class` class for `Rectangle` class as well as the `Cuboid` class also. When any instance of a reference type is created, then we may use the `getClass()` method to get the `Class` instance for the type of the instance. So, if we created an instance of `Rectangle` class and the `Cuboid` class as in earlier listing, then r1.getClass() would return the instance of the `Class` class corresponding to the `Rectangle` class, whereas r2.getClass() would return the instance of the `Class` class corresponding to the `Cuboid` class. The class instance of any data type is also accessible by using a notation as <datatype-name>.class. Therefore, we can also access the instance of the `Class` class of the `Rectangle` class as Rectangle.class. Similarly, we can also access `Class` instance of `Cuboid` as Cuboid.class, even `Class` instances for the primitive types or the array types are also accessible, e.g. int.class is the `Class`

instance for the primitive type int, and int[].class is the Class instance for the array of int. So, now we may correct the equals() method for the Rectangle and the Cuboid classes as in Listings 8.2 and 8.3.

Listing 8.2. equals method for Rectangle class

```
1       public boolean equals(Object obj) {
2           if (this.getClass() != obj.getClass()) {
3               return false;
4           }
5           if (this.length != ((Rectangle)obj).length) {
6               return false;
7           }
8           if (this.width != ((Rectangle)obj).width) {
9               return false;
10          }
11          return true;
12      }
```

Listing 8.3. equals method for Cuboid class

```
1       public boolean equals(Object obj) {
2           if (!super.equals(obj)) {
3               return false;
4           }
5           if (this.height != ((Cuboid)obj).height) {
6               return false;
7           }
8           return true;
9       }
```

EXERCISE 8.1

Override the toString() method of Account class in exercise 7.1 to return the <class-name>:<acc-no>,<name>,<balance>. Also override the equals() and the hashCode() methods such that two accounts are equal if their account numbers match, do not match any other fields in equals method. Override hashCode() method so that it is consistent with the equals() method. Update the display() method to use the toString() method.

The updated Account class is given in Listing 8.4.

Listing 8.4. Account class after overriding Object class methods

```
1
2       abstract class Account {
3
4           int accountNumber;
5           String name;
```

```
6           double balance;
7
8           Account(int acno, String n, double openBal) {
9               this.accountNumber = acno;
10              this.name = n;
11              this.balance = openBal;
12          }
13
14          int getAccountNumber() {
15              return this.accountNumber;
16          }
17
18          String getName() {
19              return this.name;
20          }
21
22          double getBalance() {
23              return this.balance;
24          }
25
26          void deposit(double amt) {
27              this.balance += amt;
28          }
29
30          boolean withdraw(double amt) {
31              if (this.balance < amt) {
32                  return false;
33              }
34              this.balance -= amt;
35              return true;
36          }
37
38          void display() {
39      //          System.out.println("Account:"+this.accountNumber+","+this
            .name+","+this.balance);
40              System.out.println(this);
41          }
42
43          static int lastAccountNumber = 1000;
44
45          Account(String n, double openBal) {
46              this(++lastAccountNumber, n, openBal);
47          }
48
49          public String toString() {
50              return this.getClass().getName()+":"+this.accountNumber+","
                    +this.name+","+this.balance;
51          }
52
53          public boolean equals(Object obj) {
```

```
54              if (this.getClass() != obj.getClass()) {
55                  return false;
56              }
57              return this.accountNumber == ((Account)obj).accountNumber;
58          }
59
60          public int hashCode() {
61              return this.accountNumber;
62          }
63      }
```

`finalize`: In Java we allocate new instances of classes and arrays using the new operator. How do we deallocate an instance? In Java the programmer does not ever deallocate any object. The deallocation is handled by the JVM for us. Java is a multi-threaded machine. When we invoke a Java application by using the `java` command, then in the JVM, over and above the thread for execution of our application (method `main()`), there are other threads in the JVM, which take care of detecting objects which cannot be used by the application (those instances which are not being referred any more), and then deallocates them. These threads are known as the garbage collector. The garbage collector is the only thread which can deallocate any instance at runtime. When the garbage collector identifies the instances eligible for garbage collection, it would first invoke the method `finalize()` on the instance which is to be deallocated, just before deallocation. `finalize()` is a `protected` method in the `Object` class and by default does not do anything. In the `Object` class it is defined as follows:

```
protected void finalize() { }
```

It may be overridden to specify the task to be done by the garbage collector before it can deallocate an instance of the given class. This method is not normally expected to be invoked by the application. It is defined, for the garbage collector, to carry out some task before deallocation. Normally we override this method to release any system resources that the instance of this class may be using.

`clone` method and the `Cloneable` interface: In the `Object` class we also have a method called `clone()`. This is used to create a copy of any given instance. The `clone()` method has the following signature:

```
1     protected Object clone()
```

What can be cloned? In Java only instances of `Cloneable` can be cloned using the `clone()` method. What is `Cloneable`? It is an interface in Java. So, the `clone()` method which is inherited from the `Object` class creates a clone for instances of the class which implement the `Cloneable` interface, and would give an error at runtime in case the `clone()` is invoked on an instance of a class which does not implement the `Cloneable` interface. What are the methods in the `Cloneable` interface? There are no methods in the `Cloneable` interface. It is a blank interface. We have many such interfaces in Java which are blank. These interfaces are

known as marker interfaces. Here, for example, if we want to allow cloning of the Rectangle instances, then we simply have to mark the Rectangle class to be Cloneable by just using the implements clause in the class definition. We do not have to implement any methods because of this interface. This method may be overridden to customize the cloning, by default the clone() method in the Object class performs a shallow cloning. If we are interested in deep cloning for any class, then we need to override this method and customize the cloning. Also note that all array types are Cloneable, i.e. all array instances can be cloned using the clone() method.

Other methods: Apart from the methods mentioned earlier, we have five more methods in the Object class. These methods are more related to multi-threading, and we will look at them in Chapter 16. Here is a listing of the remaining methods of the Object class:

```
1    public final void wait()
2    public final void wait(long millis)
3    public final void wait(long millis, int nanoseconds)
4    public final void notify()
5    public final void notifyAll()
```

8.2 SOME METHODS OF THE Class CLASS

We know the instance of Class class is created in the JVM by the class loader for every data type being used in the JVM. What are the methods in the Class class? The instance of the Class gives information about the data type whose definition it encapsulates. The following are some of the common methods available in the Class class:

```
1    public String getName() // returns the name of the data type
2    public boolean isPrimitive() // check whether data type is
         primitive
3    public boolean isArray() // check whether it is an array type
4    public boolean isInterface() // check whether the data type is
         interface
5    public Class getComponentType() // in case of array types,
         returns the element type of the array
6    public Class getSuperClass() // returns the super class data
         type
7    public Class[] getInterfaces() // returns an array of the
         implemented interfaces
```

There are many more methods in the Class class; they give information about any data type at runtime. This feature about getting information about data types at runtime is known as RTTI. Some other languages also have such a feature. RTTI is Run Time Type Information. There are also methods in the Class class which can give information about the various members defined in a class. To explore the information about members of a class and use them, we have a set of API, which is known as reflection API.

```
1    public int getModifiers() // returns an int which has bits set
         for the various modifiers
2    public Constructor[] getDeclaredConstructors()
3    public Field[] getDeclaredFields()
4    public Method[] getDeclaredMethods()
```

The reflection API has classes like the `Field`, `Constructor` and `Method`, which have methods to give information about the field, constructor or method defined in a class.

LESSONS LEARNED

- `Object` class is the super-class of all the classes in Java. It is a super-type for all the reference data types, including the interfaces and the array types.
- All instances in Java have a minimum behaviour which is the set of methods available in the `Object` class.
- The string conversion for reference data types uses the `toString()` method.
- The == operator for reference data types does not define equality of instances, it is only used to check if the two references refer to the same instance or not, but the `equals()` method is used to determine the equality for reference data types.
- All instances, whenever they are garbage collected by the garbage collector, invoke the `finalize()` method on the instance being garbage collected.
- Whenever any class is loaded by the JVM, an instance of the `Class` class is created by the class loader. An instance of `Class` class also exists for every primitive data type. The `Object` class has a method `getClass()` which returns the `Class` instance for any given specified object.

EXERCISES

1. State which of the following are true or false:

(a) Every class has a super-class.

(b) Every reference type is assignable to `Object` type.

(c) `toString()` cannot be invoked on array instances.

(d) Objects of classes which do not override the `finalize()` method cannot be garbage collected.

(e) The `finalize()` method is declared to be `final` in the `Object` class.

(f) We can override the `getClass()` method to return `null`.

(g) Any instance can be deallocated from the application by calling the `finalize()` method on the instance.

(h) There is a `Class` instance for all data types in Java including the primitive types and void (e.g. `int.class`).

(i) The `clone()` method is an abstract method declared in the `Cloneable` interface.

2. **Fill in the blanks in the following:**
   (a) finalize method is called by the garbage collector just before deallocation of an instance.
   (b) Object class is the super-class of all the classes.
   (c) When `toString()` method is invoked using a reference variable having `null` reference, then it returns null.
   (d) The `getClass()` method returns an object of the class called Class
   (e) The clone method gives an error at runtime, if invoked on instances of class which do not implement the `Cloneable` interface.
   (f) All arrays are assignable to variables of Object and Cloneable interfaces.
3. **Explain the purpose of `equals()` method in the `Object` class.**
4. **Explain the consistency rules to be observed whenever we override `equals()` method.**