CHAPTER 6
Inheritance and Sub-classing

In Chapter 5, we have seen how to define a class and the various kinds of members that can be defined in a class. Let us now look at how we can create a sub-class of a given class in Java. A sub-class always inherits the functionality of the super-class and can always add more functionality. A sub-class is created when it can be represented by an 'is a' relation. There are many examples of super-class and sub-class relationships in the real world, e.g. Car is a sub-class of Vehicle. This is so since Car is a Vehicle. The two types Vehicle and Car are related using the 'is a' relationship. Let us look at how to define a sub-class in Java.

6.1 DEFINING SUB-CLASSES

For the purpose of understanding the sub-classing rules and syntaxes, let us define a sub-class of the class `Rectangle` used in Chapter 5. Let us define a class `Cuboid` as a sub-class of `Rectangle`. Here we have certain additional members required in the sub-class. This example is not a proper example of inheritance, but we use it here for the purpose of understanding the syntax and rules related to inheritance in Java. The relationship between Cuboid and Rectangle according to the real world is not a proper sub-class and super-class relationship since we cannot say that Cuboid is a Rectangle. Here the relation of Cuboid being a sub-class of Rectangle is only a programmatic relation. It is normally advised that such relations should be avoided though we might still come across such relations being used by many developers. Let's now get started by defining the new class `Cuboid` as a sub-class of an existing class `Rectangle`. To define a sub-class of an existing class, we use the keyword `extends`. `extends` is used in a class definition to specify the super-class of the class being defined. This is also shown in Listing 6.1.

Listing 6.1. Using `extends` to define a sub-class

```
class Cuboid extends Rectangle {
    ...
    // The Cuboid class would inherit methods and
    // instance variables from the super-class Rectangle
    // and additional members required in the sub-class
    // may be added in the sub-class
}
```

Now, the new class `Cuboid` inherits the instance variables and methods from the super-class. So, it has already inherited two instance variables `length` and `width` from the super-class `Rectangle`. Now we would like to additionally have a `height` for every instance of Cuboid. This can be done by adding an instance variable `height` in the class `Cuboid` as shown in Listing 6.2.

Listing 6.2. Adding instance variables in a sub-class

```
class Cuboid extends Rectangle {
    int height; // this is the additional instance variable
}
```

The class `Cuboid` defined above has inherited the methods from the `Rectangle` class, and now we would like to have an additional method for the `Cuboid` class to return the volume of any `Cuboid` instance. This can be done by adding the volume method as given in Listing 6.3.

Listing 6.3. Defining the volume method for `Cuboid`

```
int volume() {
    return area() * height;
}
```

The above method uses the area method, which is available in the class `Cuboid`, as it has been inherited from the super-class `Rectangle`.

6.2 USING `super` TO USE THE CONSTRUCTOR OF A SUPER-CLASS

A sub-class does not inherit constructors from the super-class. So, we need to define the constructor for our class `Cuboid`. Now for creating a `Cuboid`, we would like to have three values to create an instance of a `Cuboid`—one each for the length, width and height. We now define the constructor with three parameters as given in Listing 6.4.

Listing 6.4. Using `super` to invoke a super-class constructor

```
Cuboid(int l, int w, int h) {
    super(l, w); // invoking the constructor of the super-class.
    height = h;
}
```

The first statement in the constructor above is **super**(l, w). `super` is a keyword in Java. The `super` keyword followed by a list of parameters (which may be empty) is used in a constructor to invoke the constructor of a super-class. Whenever we use `super` to invoke the constructor of the super-class, in a constructor, it has to be the first statement in the constructor. Here, in the constructor of the Cuboid class, we use it to invoke the constructor with two parameters in the super-class, Rectangle. The first statement in any constructor is either this or super. When we do not specify either this or super as the first statement in a constructor, it is assumed to be a super with no parameters. In this case, if we do not specify the first statement in the constructor as super, then it would assume super() as the first statement, i.e. if we write the above constructor as

```
Cuboid(int l, int w, int h) {
    height = h;
}
```

it would be equivalent to

```
Cuboid(int l, int w, int h) {
    super();
    height = h;
}
```

In this case since we do not have a no-arg constructor available in the super-class, the super() will result in a compilation error. So, whenever any constructor is invoked, it would always use the constructor of the super-class first. If we are creating a Cuboid then we would first be creating a Rectangle and do the additional things required for creating a Cuboid later. So, now let's say we define and overload the constructor for a Cuboid as in Listing 6.5.

Listing 6.5. Constructors in the Cuboid class

```
Cuboid(int l, int w, int h) {
    super(l, w);
    height = h;
}
Cuboid(int l) {
    this(l, l, l);
}
```

The first statement in the constructor with one parameter is this, which invokes a constructor of the same class. It would in turn invoke the constructor of the super-class first. So whenever any constructor of a class is invoked the first thing done from the constructor is to use the constructor of the super-class. This is done to carry out the initialization required for the instance variables inherited from the super-class first. This is then followed by initialization for the instance variables declared in the current class. When we do not have a no-arg constructor in the super-class available, as is the case with the Cuboid class here where the super-class Rectangle does not have a default constructor available, it becomes compulsory for a constructor in the class to explicitly invoke an existing constructor of the super-class by using super with a list of appropriate parameters. Now suppose we do not specify any constructor in the Cuboid class (a class whose super-class does not have a no-arg constructor available), then what is the implicit constructor used by the compiler? The compiler in this case would assume a constructor as follows:

```
public Cuboid() {
      super();
}
```

In case of the Cuboid class, this constructor would also fail to compile since the super-class Rectangle does not have a no-arg constructor available. So, if a class does not have a no-arg constructor available, it forces the sub-classes to have a constructor, which explicitly calls the super-class constructor using the super keyword.

6.3 METHOD OVERRIDING AND THE USE OF super

In the class Rectangle, we have a method called area(), which is implemented to return length * width. Now for the sub-class Cuboid, let's say we would like to have area() method implemented differently. ie. area for Rectangle is length * width, but in case of Cuboid, which is a kind of Rectangle(being a sub-class) we want to have area() method as length * height. ie. in general for all Rectangles the area() method is length * width, but Cuboid is a special kind of Rectangle where area is length * height. A sub-class, either overrides, or inherits, methods from the super-class. Now, in the Cuboid class we may override the area() method instead of inheriting it from the Rectangle class. Since, we are not interested in inheriting the method, from the super-class, we do it by defining the area() method in the Cuboid class as shown in Listing 6.6.

Listing 6.6. Cuboid class

```
class Cuboid extends Rectangle {
     int height;
     int volume() {
          return area() * height;
     }
     Cuboid(int l, int w, int h) {
          super(l, w);
          height = h;
     }
     Cuboid(int l) {
          this(l, l, l);
     }
     int area() { // method overriding
          return length * height;
     }
}
```

When a sub-class defines a method with the same name and same method signature, i.e. same number and types of parameters, then this method is an overriding method. The method in the super-class is said to have been overridden in the sub-class. The return type of the overriding method should be the same as the return type of the method being overridden. This was the strict rule upto Java 1.4, but from Java 5 onwards the rule about the return type for the overriding method has changed and the overriding method may now have a return type, which is the same as the return type of the method being overridden or a sub-type of the return type in the overridden method, i.e. overriding methods can have co-variant return types. Now after the overriding of the `area()` method in the `Cuboid` class, what happens to the computation of volume using the `volume()` method, which returns area() * height? Which `area()` method will be used from the `volume()` method? Since `Cuboid` class overrides the `area()` method to return length * height, the `volume()` method, which uses `area()` method in `Cuboid` class will result in length * height * height. Here, we are now interested in using the `area()` method according to the super-class in the calculation of volume. In class `Cuboid`, we can use an overridden method as it is available in the super-class by using the keyword `super` This can be done as shown in the `volume()` method in Listing 6.7.

Listing 6.7. Use of `super` to access the overridden method

```
int volume() {
    return super.area() * height;
}
```

Here, the `super` keyword is used for the method `area()` as is available on instance of `Rectangle` (super-class), i.e. length * width, and not according to the `area()` method as available for instances of the `Cuboid` class, which is length * height. When we prefix a method invocation with the keyword `super`, it would use the method as it is available for instances of the super-class.

`static` methods are not inherited by a sub-class. When we have a `static` method in the sub-class that uses the same name and method signature as the method in the super-class, then it is known as method hiding and not method overriding. Inheritance is about the behaviour and structure of instances of a class. So, static methods and static variables are never inherited.

→ Instance variable hiding

6.4 VARIABLE SHADOWING AND THE USE OF super

Looking at the class definition of a `Cuboid` as given in Listing 6.6, how many instance variables are there in an instance of `Cuboid`? There are three instance variables, the `length` and `width`, which are inherited from the super-class (`Rectangle`), and the `height`, which is declared in the `Cuboid` class. Now let's introduce another instance variable in the `Cuboid` class as given in Listing 6.8.

Listing 6.8. Shadow variable in the `Cuboid` class

```
class Cuboid extends Rectangle {
    int length;      // shadow variable
    int height;
    ... // other members in cuboid class
}
```

Now, with this definition, how many instance variables do we have in an instance of `Cuboid`? The answer is four—`length` and `width` that are inherited from super-class and the `length` and `height` that are declared in the `Cuboid` class. How do we access the two instance variables with the same name, the one that is inherited and the one that is declared in the current class? The `length` variable that is declared in the `Cuboid` class is known as a shadow variable for the `length` variable that has been inherited from the super-class. So, now within the class `Cuboid`, when we refer to `length`, it would be considered as `length` that is declared in the `Cuboid` class and to refer to the instance variable that is inherited from the super-class, we would use super.length. So, in a class definition, we have two implicit reference variables, `this` and `super`, where `this` refers to the current instance and is of the same type as the current class, and `super` refers to the current instance, but is used as an instance of the super-class, i.e. it is of the type, which is a super-class of the class in which it is used. So, within the class `Cuboid`, `this` is of the type `Cuboid` and `super` is of the type `Rectangle`, but both still refer to the same instance. Another difference is that while `this` can be used independently to refer to the current instance, `super` can only be used as a prefix to access members that are available from the super-class. In the definition of the `Cuboid` class, when we declare the shadow variable `length`, which `length` instance variable will be initialized from the constructor. It is going to be the `length` variable that is declared in the super-class and not the one that is declared in the `Cuboid` class, since we use super(l, w) in the constructor, and there is no explicit initialization for the `length` that is declared in the `Cuboid` class. Now, if we look at the `area()` method overridden in the `Cuboid` class, the `length` here would be using the `length` that is declared in the current class, i.e. the shadow variable and not the one that has been initialized in the super-class, so with this, the `area()` method on `Cuboid` instance would always return a zero. Here we may be interested in accessing the `length` that is declared in the super-class, then we could change the `area()` method in the `Cuboid` class as given in Listing 6.9.

Listing 6.9. Using `super` to access a variable from super-class

```
int area() {
    return super.length * height; // accessing the length from
        super-class
}
```

6.5 METHOD AND VARIABLE BINDING

Now let us assume the class definition for the `Cuboid` class given in Listing 6.10.

Listing 6.10. `Cuboid` class with method overriding and variable shadowing

```
class Cuboid extends Rectangle {
    int length; // shadow variable
                // remains zero since it is not initialized from
                   constructor
    int height;
    int volume() {
        return super.area() * height;
    }
    Cuboid(int l, int w, int h) {
        super(l, w);
        height = h;
    }
    Cuboid(int l) {
        this(l, l, l);
    }
    int area() {    // overriding method
        return super.length * height;
    }
}
```

What will be the output of the application in Listing 6.11?

Listing 6.11. Tesing method and variable binding

```
1  class TestCuboid {
2      public static void main(String[] args) {
3          Rectangle r1, r2;
4          Cuboid c1, c2;
5          r1 = new Rectangle(7, 5);
6          r2 = new Cuboid(7, 5, 4);
7          c1 = new Cuboid(7, 5, 4);
8          System.out.println("area of r1:"+r1.area());|| invoking
9          System.out.println("area of r2:"+r2.area());
10         System.out.println("area of c1:"+c1.area());
11         System.out.println("length of r1:"+r1.length);
12         System.out.println("length of r2:"+r2.length);
13         System.out.println("length of c1:"+c1.length);
14     }
15 }
```

In the code in Listing 6.11, we are invoking the area() methods on r1, r2 and c1 from Lines 8–10. Now, in each of these cases, which area() method would be used (whether the area() method would work according to the definition available for Rectangle or for Cuboid)? In case of Lines 8 and 10, it is very obvious that for Line 8, the area() method of the Rectangle class will be used, and in Line 10, the area() method of the Cuboid class will be used. In case of the invocation of the area() method on r2 in Line 9, we have r2 declared to be of type Rectangle, but at the runtime, r2 refers to an instance of the Cuboid. So, which area() method will be used? Will it be according to the type declaration, which is known at the compile time, or will it be according to the type of instance r2 that is refered at the runtime? In Java method binding is done at the runtime. So, the area() method that will be used is not determined at compile time? Rather, the method invocation is determined at the runtime by the instance on which a method is being invoked. So, in Line 9 of Listing 6.11, since r2 refers to an instance of Cuboid, the method used on r2 is according to the one available from the Cuboid class. The output generated from Lines 8–10 will be as given below:

```
area of r1:35
area of r2:28
area of c1:28
```

Now continuing with the next few lines (Lines 11–13) from Listing 6.11, we are accessing the length member on the instances referred by r1, r2 and c1. In each of these cases, which length is being accessed? In Line 11, r1 is of type Rectangle, and it refers to instance of Rectangle class only, so the r1.length in Line 11 refers to the length from the Rectangle class. Again in Line 13, c1 is of type Cuboid and refers to an instance of Cuboid, so, here c1.length would refer to the shadow variable length in the Cuboid. In Line 12, we have r2, which is of type Rectangle, but at the runtime it refers to an instance of Cuboid. In Java, variable binding is done at compile time. So, the r2.length is decided at the compile time, where only the type of r1 is known. So, in Line 12 r2.length is according to the Rectangle. The output generated from Lines 11–13 is as given below:

```
length of r1:7
length of r2:7
length of c1:0
```

In the above application code if we replaced Line 13 by

```
System.out.println("length of c1:"+((Rectangle)c1).length);
```

then the output generated by this line of code will be

```
length of c1:7
```

When we do any casting the compiler uses cast to determine the type for the instance of c1.

6.6 USING final WITH VARIABLES, METHODS AND CLASSES

In Java, we have a keyword final, which is used as a modifier for variables, methods as well as for classes. Let us look at the uses of this keyword.

The keyword final is used with a variable to declare a constant, i.e. if a variable is declared to be final then that variable cannot be modified. All final variables must be initialized, before they can be used.

The final modifier may be used with any kind of variable:

static variable: A static variable, if it is declared to be final, can be initialized either at the place of declaration or in a static block.

The keyword final is most commonly used with static variables. For example, we have a class Math, which has some constants like PI and E. These values have one copy only (they are static), and they are not modifiable. So, in the Math class these variables are static and final.

Instance variable: An instance variable, if it is declared as final, can be initialized at the place of declaration or in the initializer block or in the constructor.

This is not very common, but there could be times when we may declare an instance variable to be final. For example, in case of the Rectangle class, if we are required to maintain a color code for each Rectangle and the color code for the Rectangle will be decided at the time of creating, and once a Rectangle is created with a particular color code, it should not be possible to change the color code of the Rectangle. Here, since each `Rectangle` can have a different color code, the color code would be an instance variable, every instance of `Rectangle` should maintain its own value of color code. Since the color code for the `Rectangle` class cannot be modified after it has been created, it would be declared to be `final`.

Parameter variable: A parameter variable may be declared to be `final`. It is not explicitly initialized. It is initialized by the invoker of the method or the constructor.

This is also not so commonly used. Though in most cases the parameter variables never get modified in the method, the parameter variables may be declared to be `final` to indicate that they are not allowed to be modified in the method.

Local variable: A local variable if it is declared as `final` can be initialized only once before it can be used.

The `final` modifier may be used for variables of any type, i.e. primitive or reference types. In case of a variable that is a reference type, the reference cannot be changed, i.e. once a reference `final` variable is initialized, and refers to an instance, then that reference variable cannot refer to any other instance, though the state of the instance referred by the reference variable may keep on changing. Say, for example, we have a reference variable `r` declared to be `final` and refers to an instance of `Rectangle`, then one can always use *r.setDimensions(...)* to change the state of the `Rectangle` instance referred by `r`, but one cannot change the reference `r` to refer to a different instance of `Rectangle`.

The keyword `final` may be used as a modifier for a method to prevent overriding of the method. So if a method is declared as `final`, then it cannot be overridden in a sub-class. For example, if we had declared the method `area()` in the `Rectangle` class as `final`, then the method overriding, which we have done in the class `Cuboid`, would not have been possible. It would result in a compilation error.

The keyword `final` may be used as a modifier for a class to prevent inheritance of the class. If a class is declared as `final`, then it cannot be sub-classed. For example, if we declare the class `Cuboid` as a `final` class, then it would not be possible to create a sub-class of the `Cuboid` class. In Java, we have used the `String` class. This class is a `final` class. Therefore, it is not possible for anyone to create a sub-class of the `String` class.

EXERCISE 6.1 Define two sub-classes of the `Account` class of Exercise 5.2, called `SavingsAccount` and `CurrentAccount`. The sub-classes should have appropriate constructors. The `SavingsAccount` class may inherit all the methods from the super-class and works just like the base class `Account`, but in case of `CurrentAccount`, we have a fixed value of minimum balance. In case of withdrawal from a `CurrentAccount`, it works the same way as the super-class, but after withdrawal according to the super-class, the balance should be checked against the minimum balance in case the balance is less than the minimum balance. Then a penalty may also be deducted from the account. The value of the penalty is fixed and same amount is used for all `CurrentAccount` instances. Also update the `TestAccount` class to test the creation of the new classes called `SavingsAccount` and `CurrentAccount`, test the method binding at the runtime by declaring a variable of `Account` and assigning an instance of `CurrentAccount` and invoking the `withdraw()` method, which should result in a penalty.

The `SavingsAccount` and `CurrentAccount` may be defined as given in Listings 6.12 and 6.13.

Listing 6.12. SavingsAccount.java

```java

class SavingsAccount extends Account {
    SavingsAccount(int acno, String n, double openBal) {
        super(acno, n, openBal);
    }

    SavingsAccount(String n, double openBal) {
        super(n, openBal);
    }
}
```

Listing 6.13. CurrentAccount.java

```java

class CurrentAccount extends Account {

    static final double minimumBalance = 5000;
    static final double penalty = 100;

    CurrentAccount(int acno, String n, double openBal) {
        super(acno, n, openBal);
    }

    CurrentAccount(String n, double openBal) {
        super(n, openBal);
    }

    boolean withdraw(double amt) {
        if (!super.withdraw(amt)) {
            return false;
        }
        if (this.balance < minimumBalance) {
            this.balance -= penalty;
        }
        return true;
    }
}
```

LESSONS LEARNED

- A class can be created by inheriting from any existing class by using the keyword `extends`.
- A sub-class only inherits the instance variables and the instance methods from the super-class.
- A sub-class either inherits or overrides the methods from its super-class.
- The keyword `super` can be used in a constructor to invoke the constructor of a super-class. Whenever any constructor is called, the super-class constructor is always called first.

It is also used as a reference to the current instance, treating it like an instance of the super-class.

- The `final` variable cannot be modified once it is initialized.
- The `final` methods cannot be overridden.
- The `final` classes cannot be sub-classes.

EXERCISES

1. State which of the following are true or false:
   - **(a)** In Java method binding is done at the runtime. ✓
   - **(b)** Methods declared to be `final` cannot be overloaded. X - overriding
   - **(c)** In Java variable binding is done at the runtime. X compile time
   - **(d)** `final` variables cannot be modified. ✓
   - **(e)** Both `super` and `this` refer to the current Object, but their types are different. ✓
2. Fill in the blanks:
   - **(a)** The __final__ qualifier is used to declare read-only variables.
   - **(b)** The keyword __final__ is used to indicate that a class cannot be inherited further.
   - **(c)** The keyword __final__ is used with a method to prevent overriding.
   - **(d)** If the class Alpha inherits from the class Beta, class Alpha is called the __super/Base__ class and the class Beta is called the __sub/child/derived__ class.
   - **(e)** The Java keywords that cannot be used inside the body of a static method, but may be used without problems in a non-static method are __this__ and __super__.
   - **(f)** The __final__ keyword is used in Java to indicate that the class cannot be sub-classed.
3. Explain the uses of keywords `extends`, `super` and `final`.
4. Explain with example what is meant by the statement "method binding is done at the runtime". polymorphism
5. Explain with example what is meant by variable shadowing.
6. Explain the difference between overloading and overriding of methods.
7. Explain the difference between uses of keywords `final` and `static`.