Overriding and Hiding Methods ← Polymorphism → Object as a Superclass

Polymorphism

Polymorphism

The dictionary definition of polymorphism refers to a principle in biology in which an organism or species can have many different forms or stages. This principle can also be applied to object-oriented programming and languages like the Java language. Subclasses of a class can define their own unique behaviors and yet share some of the same functionality of the parent class.

Polymorphism can be demonstrated with a minor modification to the `Bicycle` class. For example, a `printDescription()` method could be added to the class that displays all the data currently stored in an instance.

```java
public void printDescription(){
    System.out.println("\nBike is " + "in gear " + this.gear
        + " with a cadence of " + this.cadence +
        " and travelling at a speed of " + this.speed + ". ");
}
```

To demonstrate polymorphic features in the Java language, extend the `Bicycle` class with a `MountainBike` and a `RoadBike` class. For `MountainBike`, add a field for suspension, which is a `String` value that indicates if the bike has a front shock absorber, `Front`. Or, the bike has a front and back shock absorber, `Dual`.

Here is the updated class:

```java
public class MountainBike extends Bicycle {
    private String suspension;

    public MountainBike(
                int startCadence,
                int startSpeed,
                int startGear,
                String suspensionType){
        super(startCadence,
              startSpeed,
              startGear);
        this.setSuspension(suspensionType);
    }

    public String getSuspension(){
      return this.suspension;
    }

    public void setSuspension(String suspensionType) {
        this.suspension = suspensionType;
    }

    public void printDescription() {
        super.printDescription();
        System.out.println("The " + "MountainBike has a" +
            getSuspension() + " suspension.");
    }
}
```

Note the overridden `printDescription()` method. In addition to the information provided before, additional data about the suspension is included to the output.

Next, create the `RoadBike` class. Because road or racing bikes have skinny tires, add an attribute to track the tire width. Here is the `RoadBike` class:

```java
public class RoadBike extends Bicycle{
    // In millimeters (mm)
    private int tireWidth;

    public RoadBike(int startCadence,
                    int startSpeed,
                    int startGear,
                    int newTireWidth){
```

```java
        super(startCadence,
              startSpeed,
              startGear);
        this.setTireWidth(newTireWidth);
    }

    public int getTireWidth(){
      return this.tireWidth;
    }

    public void setTireWidth(int newTireWidth){
        this.tireWidth = newTireWidth;
    }

    public void printDescription(){
        super.printDescription();
        System.out.println("The RoadBike" + " has " + getTireWidth() +
            " MM tires.");
    }
}
```

Note that once again, the `printDescription()` method has been overridden. This time, information about the tire width is displayed.

To summarize, there are three classes: `Bicycle`, `MountainBike`, and `RoadBike`. The two subclasses override the `printDescription()` method and print unique information.

Here is a test program that creates three `Bicycle` variables. Each variable is assigned to one of the three bicycle classes. Each variable is then printed.

```java
public class TestBikes {
  public static void main(String[] args){
    Bicycle bike01, bike02, bike03;

    bike01 = new Bicycle(20, 10, 1);
    bike02 = new MountainBike(20, 10, 5, "Dual");
    bike03 = new RoadBike(40, 20, 8, 23);

    bike01.printDescription();
    bike02.printDescription();
    bike03.printDescription();
  }
}
```

The following is the output from the test program:

```
Bike is in gear 1 with a cadence of 20 and travelling at a speed of 10.

Bike is in gear 5 with a cadence of 20 and travelling at a speed of 10.
The MountainBike has a Dual suspension.

Bike is in gear 8 with a cadence of 40 and travelling at a speed of 20.
The RoadBike has 23 MM tires.
```

The Java virtual machine (JVM) calls the appropriate method for the object that is referred to in each variable. It does not call the method that is defined by the variable's type. This behavior is referred to as virtual method invocation and demonstrates an aspect of the important polymorphism features in the Java language.

Hiding Fields

Within a class, a field that has the same name as a field in the superclass hides the superclass's field, even if their types are different. Within the subclass, the field in the superclass cannot be referenced by its simple name. Instead, the field must be accessed through super, which is covered in the next section. Generally speaking, we do not recommend hiding fields as it makes code difficult to read.

Using the Keyword Super

Accessing Superclass Members

If your method overrides one of its superclass's methods, you can invoke the overridden method through the use of the keyword `super`. You can also use `super` to refer to a hidden field (although hiding fields is discouraged). Consider this class, `Superclass`:

```java
public class Superclass {

    public void printMethod() {
        System.out.println("Printed in Superclass.");
    }
}
```

Here is a subclass, called `Subclass`, that overrides `printMethod()`:

```java
public class Subclass extends Superclass {

    // overrides printMethod in Superclass
    public void printMethod() {
        super.printMethod();
        System.out.println("Printed in Subclass");
    }
    public static void main(String[] args) {
        Subclass s = new Subclass();
        s.printMethod();
    }
}
```

Within `Subclass`, the simple name `printMethod()` refers to the one declared in `Subclass`, which overrides the one in `Superclass`. So, to refer to `printMethod()` inherited from `Superclass`, `Subclass` must use a qualified name, using `super` as shown. Compiling and executing `Subclass` prints the following:

```
Printed in Superclass.
Printed in Subclass
```

Subclass Constructors

The following example illustrates how to use the super keyword to invoke a superclass's constructor. Recall from the `Bicycle` example that `MountainBike` is a subclass of `Bicycle`. Here is the `MountainBike` (subclass) constructor that calls the superclass constructor and then adds initialization code of its own:

```java
public MountainBike(int startHeight,
                    int startCadence,
                    int startSpeed,
                         int startGear) {
    super(startCadence, startSpeed, startGear);
    seatHeight = startHeight;
}
```

Invocation of a superclass constructor must be the first line in the subclass constructor.

The syntax for calling a superclass constructor is

```
super();
```

or

```
super(parameter list);
```

With `super()`, the superclass no-argument constructor is called. With `super(parameter list)`, the superclass constructor with a matching parameter list is called.

> *Note: If a constructor does not explicitly invoke a superclass constructor, the Java compiler automatically inserts a call to the no-argument constructor of the superclass. If the super class does not have a no-argument constructor, you will get a compile-time error. Object does have such a constructor, so if `Object` is the only superclass, there is no problem.*

If a subclass constructor invokes a constructor of its superclass, either explicitly or implicitly, you might think that there will be a whole chain of constructors called, all the way back to the constructor of `Object`. In fact, this is the case. It is called *constructor chaining*, and you need to be aware of it when there is a long line of class descent.

Writing Final Classes and Methods

You can declare some or all of a class's methods final. You use the `final` keyword in a method declaration to indicate that the method cannot be overridden by subclasses. The `Object` class does this—a number of its methods are final.

You might wish to make a method final if it has an implementation that should not be changed and it is critical to the consistent state of the object. For example, you might want to make the `getFirstPlayer()` method in this `ChessAlgorithm` class final:

```java
class ChessAlgorithm {
    enum ChessPlayer { WHITE, BLACK }
    ...
    final ChessPlayer getFirstPlayer() {
        return ChessPlayer.WHITE;
    }
    ...
}
```

Methods called from constructors should generally be declared final. If a constructor calls a non-final method, a subclass may redefine that method with surprising or undesirable results.

Note that you can also declare an entire class final. A class that is declared final cannot be subclassed. This is particularly useful, for example, when creating an immutable class like the `String` class.

Last update: *September 14, 2021*

Overriding and Hiding Methods ← **Polymorphism** → Object as a Superclass

Home > Tutorials > Inheritance > Polymorphism