2.6 Modifying Inherited Methods

KEY IDEA

A subclass can make limited modifications to inherited methods as well as add new methods.

Besides adding new services to an object, sometimes we want to modify existing services so that they do something different. We might use this facility to make a dancing robot that, when sent a `move` message, first spins around on its current intersection and then moves. We might build a kind of robot that turns very fast even though it continues to move relatively slowly, or (eventually), a robot that checks to see if something is present before attempting to pick it up. In a graphical user interface, we might make a special kind of component that paints a picture on itself. In all of these situations, we replace the definition of a method in a superclass with a new definition. This replacement process is called **overriding**.

2.6.1 Overriding a Method Definition

To override the definition of a method, you create a new method with the same name, return type, and parameters in a subclass. These constitute the method's **signature**. As an example, let's create a new kind of robot that can turn left quickly. That is, we will override the `turnLeft` method with a new method that performs the same service differently.

You may have noticed that the online documentation for `Robot` includes a method named `setSpeed`, which allows a robot's speed to be changed. Our general strategy will be to write a method that increases the robot's speed, turns, and then returns the speed to normal. Turning quickly doesn't seem to be something we would use often, so it has not been added to `RobotSE`. On the other hand, it seems reasonable that fast-turning robots need to turn around and turn right, so our new class will extend `RobotSE`.

As the first step in creating the `FastTurnBot` class, we create the constructor and the shell of the new `turnLeft` method, as shown in Listing 2-11.

FIND THE CODE

ch02/override/

PATTERN

Extended Class

Listing 2-11: *An incomplete class which overrides* `turnLeft`

```
import becker.robots.*;

/** A FastTurnBot turns left very quickly relative to its normal speed.
 *  @author Byron Weber Becker  */
public class FastTurnBot extends RobotSE
{
   /** Construct a new FastTurnBot.
    * @param aCity         The city in which the robot appears.
    * @param aStreet       The street on which the robot appears.
    * @param anAvenue      The avenue on which the robot appears.
    * @param aDirection    The direction the robot initially faces. */
```

Listing 2-11: *An incomplete class which overrides* `turnLeft` (continued)

```
    public FastTurnBot(City aCity, int aStreet, int anAvenue,
                       Direction aDirection)
    { super(aCity, aStreet, anAvenue, aDirection);
    }

    /** Turn 90 degrees to the left, but do it more quickly than normal. */
    public void turnLeft()
    {
    }
}
```

PATTERN

Constructor

When this class is instantiated and sent a `turnLeft` message, it does nothing. When the message is received, Java starts with the object's class (`FastTurnBot`) and looks for a method matching the message. It finds one and executes it. Because the body of the method is empty, the robot does nothing.

How can we get it to turn again? We *cannot* write `this.turnLeft();` in the body of `turnLeft`. When a `turnLeft` message is received, Java finds the `turnLeft` method and executes it. The `turnLeft` method then executes `this.turnLeft`, sending *another* `turnLeft` message to the object. Java finds the same `turnLeft` method and executes it. The process of executing it sends *another* `turnLeft` message to the object, so Java finds the `turnLeft` method again, and repeats the sequence. The program continues sending `turnLeft` messages to itself until it runs out of memory and crashes. This problem is called **infinite recursion**.

LOOKING AHEAD

Recursion occurs when a method calls itself. Although recursion causes problems in this case, it is a powerful technique.

What we really want is the `turnLeft` message in the `FastTurnBot` class to execute the `turnLeft` method in a superclass. We want to send a `turnLeft` message in such a way that Java begins searching for the method in the superclass rather than the object's class. We can do so by using the keyword `super` instead of the keyword `this`. That is, the new definition of `turnLeft` should be as follows:

KEY IDEA

Using `super` *instead of* `this` *causes Java to search for a method in the superclass rather than the object's class.*

```
public void turnLeft()
{ super.turnLeft();
}
```

We have returned to where we started. We have a robot that turns left at the normal speed. When a `FastTurnBot` is sent a `turnLeft` message, Java finds this `turnLeft` method and executes it. This method sends a message to the superclass to execute its `turnLeft` method, which occurs at the normal speed.

To make the robot turn faster, we add two calls to `setSpeed`, one before the call to `super.turnLeft()` to increase the speed, and one more after the call to decrease the speed back to normal. The documentation indicates that `setSpeed` requires a single parameter, the number of moves or turns the robot should make in one second.

The default speed of a robot is two moves or turns per second. The following method uses `setSpeed` so the robot turns 10 times as fast as normal, and then returns to the usual speed.

```
public void turnLeft()
{ this.setSpeed(20);
  super.turnLeft();
  this.setSpeed(2);
}
```

The `FastTurnBot` class could be tested with a small program such as the one in Listing 2-12. Running the program shows that `speedy` does, indeed, turn quickly when compared to a `move`.

FIND THE CODE

ch02/override/

Listing 2-12: *A program to test a* `FastTurnBot`

```
import becker.robots.*;

/** A program to test a FastTurnBot.
 *  @author Byron Weber Becker */
public class Main extends Object
{
  public static void main(String[] args)
  { City cairo = new City();
    FastTurnBot speedy = new FastTurnBot(
                cairo, 1, 1, Direction.EAST);

    speedy.turnLeft();
    speedy.move();
    speedy.turnLeft();
    speedy.turnLeft();
    speedy.turnLeft();
    speedy.turnLeft();
    speedy.turnLeft();
    speedy.move();
  }
}
```

2.6.2 Method Resolution

So far, we have glossed over how Java finds the method to execute, a process called **method resolution**. Consider Figure 2-13, which shows the class diagram of a `FastTurnBot`. Details not relevant to the discussion have been omitted, including constructors, attributes, some services, and even some of the `Robot`'s superclasses (represented by an empty rectangle). The class named `Object` is the superclass, either directly or indirectly, of every other class.

When a message is sent to an object, Java always begins with the object's class, looking for a method implementing the message. It keeps going up the hierarchy until it either finds a method or it reaches the ultimate superclass, `Object`. If it reaches `Object` without finding an appropriate method, a compile-time error is given.

Let's look at several different examples. Consider the following code:

```
FastTurnBot speedy = new FastTurnBot(...);
speedy.move();
```

To execute the `move` method, Java begins with `speedy`'s class, `FastTurnBot`, in the search for the method. When Java doesn't find a method named `move` in `FastTurnBot`, it looks in `RobotSE` and then in `Robot`, where a method matching the `move` method is found and executed.

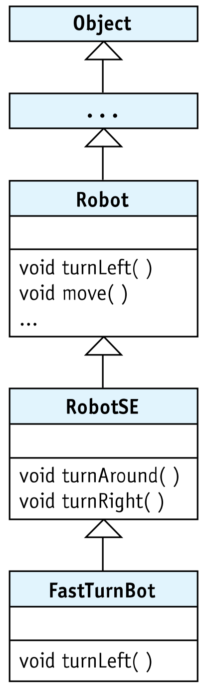
As another example, consider the following code:

```
RobotSE special = new RobotSE(...);
special.move();
```

The search for a `move` method begins with `RobotSE`, the class that instantiated `special`. It doesn't matter that `RobotSE` has been extended by another class; what matters is that when `special` was constructed, the programmer used the constructor for `RobotSE`. Therefore, searches for methods begin with `RobotSE`.

(figure 2-13)

Class diagram of a `FastTurnBot`



Once again, consider `speedy`. What happens if `speedy` is sent the `turnAround` message? The search for the `turnAround` method begins with `speedy`'s class, `FastTurnBot`. It's found in `RobotSE` and executed. As it is executed, it calls `turnLeft`. Which `turnLeft` method is executed, the one in `FastTurnBot` or the one in `Robot`?

KEY IDEA

Overriding a method can affect other methods that call it, even methods in a superclass.

The `turnLeft` message in `turnAround` is sent to the implicit parameter, `this`. The implicit parameter is the same as the object that was originally sent the message, `speedy`. So Java begins with `speedy`'s class, searching for `turnLeft`. It finds the method that turns quickly and executes it. Therefore, a subclass can affect how methods in a superclass are executed.

LOOKING AHEAD

Written Exercise 2.4 asks you to trace similar examples.

If `turnAround` is written as follows, the result would be different.

```
public void turnAround()
{ super.turnLeft();
  super.turnLeft();
}
```

KEY IDEA

The search for the method matching a message sent to `super` *begins in the method's superclass.*

Now the search for `turnLeft` begins with the superclass of the class containing the method, or `Robot`. `Robot` contains a `turnLeft` method. It is executed, and the robot turns around at the normal pace.

Suppose you occasionally want `speedy` to turn left at its normal speed. Can you somehow skip over the new definition of `turnLeft` and execute the normal one, the one that was overridden? No. If we really want to execute the original `turnLeft`, we should not have overridden it. Instead, we should have simply created a new method, perhaps called `fastTurnLeft`.

2.6.3 Side Effects

`FastTurnBot` has a problem, however. Suppose that Listing 2-12 contained the statement `speedy.setSpeed(20);` just before line 12. This statement would speed `speedy` up dramatically. Presumably, the programmer wanted `speedy` to be speedier than normal all of the time. After its first `turnLeft`, however, `speedy` would return to its normal pace of 2 moves per second.

This phenomenon is called a **side effect**. Invoking `turnLeft` changed something it should not have changed. Our programmer will be very annoyed if she must reset the speed after every command that turns the robot. Ideally, a `FastTurnBot` returns to its previous speed after each turn.

KEY IDEA

Not only are side effects annoying, they can lead to errors. Avoid them where possible; otherwise, document them.

The programmer can use the `getSpeed` query to find out how long the robot currently takes to turn. This information can be used to adjust the speed to its original value after the turn is completed. The new version of `turnLeft` should perform the following steps:

set the speed to 10 times the current speed
turn left
set the speed to one-tenth of the (now faster) speed

The query `this.getSpeed()` obtains the current speed. Multiplying the speed by 10 and using the result as the value to `setSpeed` increases the speed by a factor of 10. After the turn, we can do the reverse to decrease the speed to its previous value, as shown in the following implementation of `turnLeft`:

LOOKING AHEAD

Another approach is to remember the current speed. When the robot is finished turning, set the speed to the remembered value. More in Chapters 5 and 6.

```
public void turnLeft()
{ this.setSpeed(this.getSpeed() * 10);
  super.turnLeft();
  this.setSpeed(this.getSpeed() / 10);
}
```

Using queries and doing arithmetic will be discussed in much more detail in the following chapters.

2.7 GUI: Extending GUI Components

The Java package that implements user interfaces is known as the **Abstract Windowing Toolkit** or **AWT**. A newer addition to the AWT is known as **Swing**. These packages contain classes to display components such as windows, buttons, and textboxes. Other classes work to receive input from the mouse, to define colors, and so on.