## 2.1 Understanding Programs: An Experiment

Let's try an experiment. Find a watch or a clock that can measure time in seconds. Measure the number of seconds it takes you to understand the program shown in Listing 2-1. In particular, describe the path the robot takes, its final position, and its direction.

**FIND THE CODE**

*ch02/experiment/*

**Listing 2-1:** *An experiment in understanding a longer program*

```
import becker.robots.*;

public class Longer
{
  public static void main(String[] args)
  { City austin = new City();
    Robot lisa = new Robot(austin, 3, 3, Direction.EAST);

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

Now, imagine that we had a new kind of robot with commands to turn around, turn right, and move ahead three times. Time yourself again while you try to understand the program in Listing 2-2. The robot in this program does something different. How fast can you accurately figure out what?

FIND THE CODE

*ch02/experiment/*

**Listing 2-2:** *An experiment in understanding a shorter program*

```java
import becker.robots.*;

public class Shorter
{
  public static void main(String[] args)
  { City austin = new City();
    ExperimentRobot lisa = new ExperimentRobot(
                                 austin, 3, 2, Direction.SOUTH);

    lisa.move3();
    lisa.turnRight();
    lisa.move3();
    lisa.turnAround();
    lisa.move3();
    lisa.turnLeft();
    lisa.move3();
    lisa.turnAround();
  }
}
```

You probably found the second program easier—and faster—to read and understand. The results shown in Table 2-1 are from a group of beginning Java programmers who performed the same experiment.

(table 2-1)

*Results of an experiment in understanding programs*

| Program | Minimum Time (seconds) | Average Time (seconds) | Maximum Time (seconds) |
|---|---|---|---|
| Longer | 12 | 87 | 360 |
| Shorter | 10 | 46 | 120 |

KEY IDEA

*Adapt your language to express your ideas clearly and concisely*

Why did we comprehend the second program more quickly? We raised the level of abstraction; the language we used (`turnAround`, `turnRight`, `move3`) matches our thinking more closely than the first program. Essentially, we created a more natural programming language for ourselves.

Raising the level of abstraction with language that matches our thinking has a number of benefits.

- Raising the level of abstraction makes it easier to write the program. It's easier for a programmer to think, "And then I want the robot to turn around" than to think, "The robot should turn around so I need to tell it to turn left and then turn left again." We can think of a task such as `turnAround`, deferring the definition of the task until later. Abstraction allows us to concentrate on the big picture instead of getting stuck on low-level details.
- Raising the level of abstraction allows us to understand programs better. An instruction such as `turnAround` allows the programmer to express her intent. Knowing the intent, we can better understand how this part of the program fits with the rest of the program. It's easier to be told that the programmer wants the robot to turn around than to infer it from two consecutive `turnLeft` commands.
- When we know the intent, it is easier to debug the program. Figuring out what went wrong when faced with a long sequence of service invocations is hard. When we know the intent, we can first ask if the programmer is intending to do the correct thing (**validation**), and then we can ask if the intent is implemented correctly (**verification**). This task is much easier than facing the entire program at once and trying to infer the intent from individual service invocations.

**LOOKING AHEAD**

*Quality software is easier to understand, write, debug, reuse, and modify. We will explore this further in Chapter 11.*

- We will find that extending the language makes it easier to modify the program. With the intent more clearly communicated, it is easier to find the places in the program that need modification.
- We can create commands that may be useful in other parts of the program, or even in other programs. By reusing old services and creating new services that will be easy to reuse in the future, we can save ourselves effort. We're working smarter rather than harder, as the saying goes.

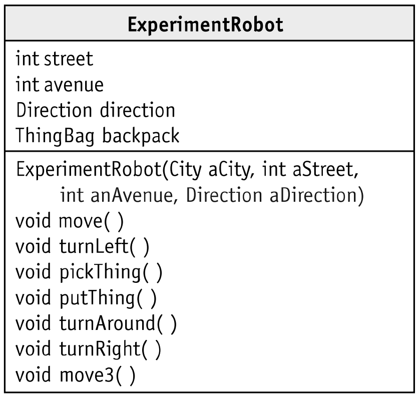
**KEY IDEA**

*Work smarter by reusing code.*

In the next section, we will learn how to extend an existing class such as `Robot` to add new services such as `turnAround`. We'll see that these ideas apply to all Java programs, not just those involving robots.

## 2.2 Extending the `Robot` Class

Let's see how the new kind of robot used in Listing 2-2 was created. What we want is a robot that can turn around, turn right, and move ahead three times—in addition to all the services provided by ordinary robots, such as turning left, picking things up, and putting them down. In terms of a class diagram, we want a robot class that corresponds to Figure 2-1.



**ExperimentRobot**

int street
int avenue
Direction direction
ThingBag backpack

ExperimentRobot(City aCity, int aStreet,
    int anAvenue, Direction aDirection)
void move( )
void turnLeft( )
void pickThing( )
void putThing( )
void turnAround( )
void turnRight( )
void move3( )

(figure 2-1)

*Class diagram for the new robot class,* `ExperimentRobot`

The class shown in Figure 2-1 is almost the same as `Robot`—but not quite. We would like a way to augment the existing functionality in `Robot` rather than implementing it again, similar to the camper shown in Figure 2-2.

(figure 2-2)

*Van extended to be a camper*

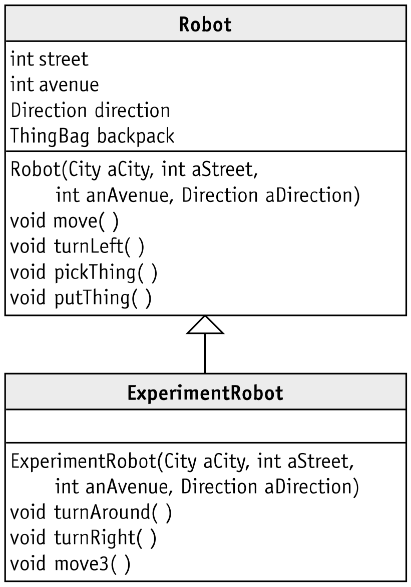
This vehicle has a number of features for people who enjoy camping: a bed in the pop-up top, a small sink and stove, a table, and so on. Did the camper's manufacturer design and build the entire vehicle just for the relatively few customers who want such a vehicle for camping? No. The manufacturer started with a simple cargo van and then added the special options for camping. The cargo van gave them the vehicle's basic frame, engine, transmission, driver's seat, instrument panel, and so on. Using all this infrastructure from an existing vehicle saved them a lot of work, so they could focus on the unique aspects required by a camper. The same cargo van, by the way, is also extended in different ways to carry more passengers.

Just as the camper manufacturer extended a van with additional features, we will extend `Robot` with additional services. Java actually uses the keyword `extends` for this purpose.

**KEY IDEA**

*Start with something that does most of what you need. Then customize it for your particular use.*

Figure 2-3 shows a class diagram in which `ExperimentRobot` extends `Robot`. Notice that the `Robot` class diagram is exactly the same as Figure 1-8. The `ExperimentRobot` class diagram shows only the attributes and services that are added to the `Robot` class. In the case of an `ExperimentRobot`, only services are added; no attributes. The hollow-tipped arrow between the two classes shows the relationship between them: `ExperimentRobot`, at the tail of the arrow, extends `Robot`, at the head of the arrow.



(figure 2-3)

*Class diagram showing* `ExperimentRobot` *extending* `Robot`

## 2.2.1 The Vocabulary of Extending Classes

When communicating about extending a class, we say that the `Robot` class is the **superclass** and the `ExperimentRobot` class is the **subclass**. Unless you're familiar with the language of mathematical sets or biology, this use of "sub" and "super" may seem backwards. In these settings, "super" means a more inclusive set or category. For example, an `ExperimentRobot` is a special kind of `Robot`. We will also define other special kinds of `Robots`. `Robot` is the more inclusive set, the superclass.

**KEY IDEA**

*The class that is extended is called the "superclass." The new class is called the "subclass."*

We might also say that `ExperimentRobot` **inherits** from `Robot` or that `ExperimentRobot` **extends** `Robot`.

If you think of a superclass as the parent of a class, that child class can have a grandparent and even a great-grandparent because the superclass may have its own superclass. It is, therefore, appropriate to talk about a class's superclasses (plural) even though it has only one direct superclass.

In a class diagram such as Figure 2-3, the superclass is generally shown above the subclass, and the arrow always points from the subclass to the superclass.

## 2.2.2 The Form of an Extended Class

**LOOKING AHEAD**

*When a pattern is used, an icon appears in the margin, as shown here for the Extended Class pattern. The patterns for this chapter are revisited in Section 2.8.*

The form of an extended class is as follows:

```
1 import «importedPackage»;
2
3 public class «className» extends «superClass»
4 {
5    «list of attributes used by this class»
6    «list of constructors for this class»
7    «list of services provided by this class»
8 }
```

**PATTERN**

*Extended Class*

The `import` statement is the same here as in the Java Program pattern. Line 3 establishes the relationship between this class and its superclass using the keyword `extends`. For an `ExperimentRobot`, for example, this line would read as follows:

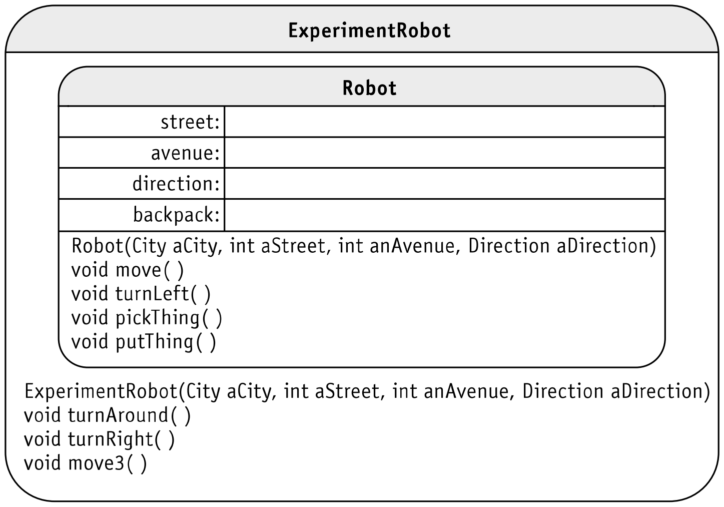
```
public class ExperimentRobot extends Robot
```

Lines 5, 6, and 7 of the code template are slots for attributes, constructors, and services. In the next section, we will implement a constructor. In the following sections, we will implement the services `turnAround`, `turnRight`, and `move3`. We will not be adding attributes to our classes until Chapter 6. Until then, we will use only the attributes inherited from the superclass.

## 2.2.3 Implementing a Constructor

The purpose of the constructor is to initialize each object that is constructed. That is, when the statement `Robot karel = new Robot(austin, 1, 1, Direction.SOUTH)` is executed, the constructor for the `Robot` class ensures that the attributes `street`, `avenue`, `direction`, and `backpack` are all given appropriate values.

We are not adding any attributes to the `ExperimentRobot`. So what is there to initialize? Is a constructor necessary? Yes. Because `ExperimentRobot` extends the `Robot` class, each `ExperimentRobot` object can be visualized as having a `Robot` object inside of it (see Figure 2-4). We need to ensure that the `Robot`-inside-the-`ExperimentRobot` object is correctly initialized with appropriate values for `street`, `avenue`, `direction`, and `backpack`.



(figure 2-4)

*Visualizing an* `ExperimentRobot` *as containing a* `Robot`

The constructor for `ExperimentRobot` is only four lines long—three if all the parameters would fit on the same line:

```
1  public ExperimentRobot(City aCity, int aStreet,
2                         int anAvenue, Direction aDirection)
3  { super(aCity, aStreet, anAvenue, aDirection);
4  }
```

Lines 1 and 2 declare the parameters required to initialize the `Robot`-inside-the-`ExperimentRobot`: a city, the initial street and avenue, and the initial direction. Each parameter is preceded by its type.

**KEY IDEA**

*The constructor must ensure the superclass is properly initialized.*

Line 3 passes on the information received from the parameters to the `Robot`-inside-the-`ExperimentRobot`. Object initialization is performed by a constructor, so you would think that line 3 would call the constructor of the superclass:

```
Robot(aCity, aStreet, anAvenue, aDirection);      // doesn't work!
```

**LOOKING AHEAD**

*Section 2.6 explains another use for the keyword* `super`.

However, the designers of Java chose to use a keyword, `super`, instead of the name of the superclass. When `super` is used as shown in line 3, Java looks for a constructor in the superclass with parameters that match the provided arguments, and calls it. The effect is the same as you would expect from using `Robot`, as shown earlier.

**LOOKING AHEAD**

*We will explore parameters further in Section 4.6 in the context of writing services with parameters.*

When an `ExperimentRobot` is constructed with the following statement, the values passed as arguments (`austin`, `3`, `2`, and `Direction.SOUTH`) are copied into the parameters (`aCity`, `aStreet`, `anAvenue`, and `aDirection`) in the `ExperimentRobot` constructor.

```
ExperimentRobot lisa = new ExperimentRobot(austin,
                                    3, 2, Direction.SOUTH);
```

Then, in line 3, those same values are passed as arguments to the parameters in the superclass's constructor.

Two other details about the constructor are that it must have the same name as the class and it does not have a return type—not even `void`. If a constructor has a name different from the class, the compiler considers it a service without a return type, and issues a compile-time error. If a constructor has a return type, the compiler considers it a service, and may not display an error until a client tries to use the constructor. Then the compiler will complain that it can't find it—because the constructor is being interpreted as a service.

Listing 2-3 contains the first steps in defining the `ExperimentRobot` class. It has a number of the template slots filled in, including imported classes, the class name, and the extended class's name. It also includes a constructor, but none of the new services. Just like the programs we wrote in Chapter 1, this class should be placed in its own file named `ExperimentRobot.java`.

**FIND THE CODE**

*cho2/experiment/*

**Listing 2-3:** *The* `ExperimentRobot` *class with a constructor but no services*

```
import becker.robots.*;

public class ExperimentRobot extends Robot
{
  public ExperimentRobot(City aCity, int aStreet,
                              int anAvenue, Direction aDirection)
  { super(aCity, aStreet, anAvenue, aDirection);
  }

  // The new services offered by an ExperimentRobot will be inserted here.
}
```

With this modest beginning, we can write a program that includes the following statement:

```
ExperimentRobot lisa = new ExperimentRobot(austin,
                                    3, 2, Direction.SOUTH);
```

The robot `lisa` can do all things any normal robot can do. `lisa` can move, turn left, pick things up, and put them down again. An `ExperimentRobot` is a kind of `Robot` object and has inherited all those services from the `Robot` class. In fact, the `Robot` in line 7 of Listing 2-1 could be replaced with an `ExperimentRobot`. Even with no other changes, the program would execute as it does with a `Robot`. However, an `ExperimentRobot` cannot yet respond to the messages `turnAround`, `turnRight`, or `move3`.

**KEY IDEA**

*A class with only a constructor, like Listing 2-3, is complete and can be used in a program. It just doesn't add any new services.*

## 2.2.4 Adding a Service

A service is an idea such as "turn around." To actually implement this idea, we add a **method** to the class, which contains code to carry out the idea. When we want a robot to turn around, we send a message to the robot naming the `turnAround` service. This message causes the code in the corresponding method to be executed.

**KEY IDEA**

*Services are ideas. Methods contain code to implement the idea. Services are invoked with a message. The object responds by executing a method.*

An analogy may help distinguish services, messages, and methods. Every child can eat. This is a service provided by the child. It is something the child can do. A message from a parent, "Come and eat your supper" causes the child to perform the service of eating. The particular method the child uses to eat, however, depends on the instructions he or she has received while growing up. The child may use chopsticks, a fork, or a fork and a knife. The idea (eating) is the service. The message ("eat your supper") causes the service to be performed. How the service is performed (chopsticks, fork, and so on) is determined by the instructions in the method.

The service `turnAround` may be added to the `ExperimentRobot` class by inserting the following method between lines 10 and 11 in Listing 2-3:

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

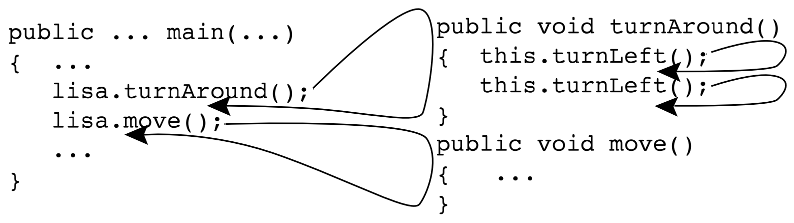
Now the robot `lisa` can respond to the `turnAround` message. When a client says `lisa.turnAround()`, the robot knows that `turnAround` is defined as turning left twice, once for each `turnLeft` command in the body of `turnAround`.

### Flow of Control

Recall the Sequential Execution pattern from Chapter 1. It says that each statement is executed, one after another. Each statement finishes before the next one in the sequence begins. When a program uses `lisa.turnAround()` we break out of the Sequential Execution pattern. The **flow of control**, or the sequence in which statements are executed, does not simply go to the next statement (yet). First it goes to the statements contained in `turnAround`, as illustrated in Figure 2-5.

(figure 2.5)

*Flow of control when executing methods*



```
public ... main(...)          public void turnAround()
{  ...                        {  this.turnLeft();
   lisa.turnAround();            this.turnLeft();
   lisa.move();               }
   ...                        public void move()
}                             {   ...
                              }
```

When main sends the message `lisa.turnAround()`, Java finds the definition of `turnAround` and executes each of the statements it contains. It then returns to the statement following `lisa.turnAround()`. This is an example of a much more general pattern that occurs each time a message is sent to an object:

**KEY IDEA**

*Calling a method temporarily interrupts the Sequential Execution pattern.*

- The method implementing the service named in the message is found.
- The statements contained in the method are executed. Unless told otherwise, the statements are executed sequentially according to the Sequential Execution pattern.
- Flow of control returns to the statement following the statement that sent the message.

Look again at Figure 2-5. This same pattern is followed when `lisa` is sent the `move` message, although we don't know what the statements in the `move` method are, so they are not shown in the figure. Similarly, when `turnAround` is executed, each `turnLeft` message it sends follows the same pattern: Java finds the method implementing `turnLeft`, executes the statements it contains, and then it returns, ready to execute the next statement in `turnAround`.

When we are considering only `main`, the Sequential Execution pattern still holds. When we look only at `turnAround`, the Sequential Execution pattern holds there, too. But when we look at a method together with the methods it invokes, we see that the Sequential Execution pattern does *not* hold. The flow of control jumps from one place in the program to another—but always in an orderly and predictable manner.

### The Implicit Parameter: this

In previous discussions, we have said that parameters provide information necessary for a method or constructor to do its job. Because `turnAround` has no parameters, we might conclude that it doesn't need any information to do its job. That conclusion is not correct.

One vital piece of information `turnAround` needs is which robot it should turn around. When a client says `lisa.turnAround()`, the method must turn `lisa` around, and if a client says `karel.turnAround()`, the method must turn `karel` around. Clearly the method must know which object it is to act upon.

This piece of information is needed so often and is so vital that the designers of Java made accessing it extremely easy for programmers. Whenever a method is invoked with the pattern *«objectReference».«methodName»(…)*, *«objectReference»* becomes an **implicit parameter** to *«methodName»*. The implicit parameter is the object receiving the message. In the case of `lisa.turnAround()`, the implicit parameter is `lisa`, and for `karel.turnAround()`, the implicit parameter is `karel`.

The implicit parameter is accessed within a method with the keyword `this`. The statement `this.turnLeft()` means that the same robot that called `turnAround` will be instructed to turn left. If the client said `lisa.turnAround()`, then `lisa` will be the implicit parameter and `this.turnLeft()` will instruct `lisa` to turn left.

Sometimes when a person learns a new activity with many steps they will mutter instructions to themselves: "First, *I* turn left. Then *I* turn left again." Executing a method definition is like that, except that "I" is replaced by "this robot." You can think of the `ExperimentRobot` as muttering instructions to itself: "First, *this* robot turns left. Then *this* robot turns left again."

#### `public` and `void` Keywords

The two remaining keywords in the method definition are `public` and `void`. The keyword `public` says that this method is available for any client to use. In Section 3.6, we will learn about situations for which we might want to prevent some clients from using certain methods. In those situations, we will use a different keyword.

The keyword `void` distinguishes a command from a query. Its presence tells us that `turnAround` does not return any information to the client.

### 2.2.5 Implementing `move3`

Implementing the `move3` method is similar to `turnAround`, except that we want the robot to move forward three times. The complete method follows. Like `turnAround`, it is placed inside the class, but outside of any constructor or method.

**LOOKING AHEAD**

*Eventually we will learn how to write a method with a parameter so we can say* `lisa.move(50)`*—or any other distance.*

```
public void move3()
{ this.move();
  this.move();
  this.move();
}
```

As with `turnAround`, we want the same robot that is executing `move3` to do the moving. We therefore use the keyword `this` to specify which object receives the `move` messages.

### 2.2.6 Implementing `turnRight`

To tell a robot to turn right, we could say “turn left, turn left, turn left.” We could also say “turn around, then turn left.” Both work. The first approach results in the following method:

PATTERN

*Parameterless Command*

```
public void turnRight()
{ this.turnLeft();
  this.turnLeft();
  this.turnLeft();
}
```

KEY IDEA

*An object can send messages to itself, invoking its own methods.*

The second approach is more interesting, resulting in this method:

```
public void turnRight()
{ this.turnAround();
  this.turnLeft();
}
```

LOOKING AHEAD

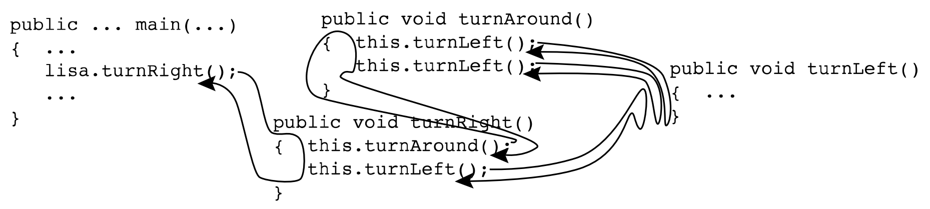
*Methods calling other methods is a core idea of Stepwise Refinement, the topic of Chapter 3.*

The second version works by asking the `ExperimentRobot` object to execute one of its own methods, `turnAround`. The robot finds the definition of `turnAround` and executes it (that is, it turns left twice as the definition of `turnAround` says it should). When it has finished executing `turnAround`, it is told to `turnLeft` one more time. The robot has then turned left three times, as desired.

This flow of control is illustrated in Figure 2-6. Execution begins with `lisa.turnRight()` in the `main` method. It proceeds as shown by the arrows. Each method executes the methods it contains and then returns to its client, continuing with the statement after the method call. Before a method is finished, each of the methods it calls must also be finished.

(figure 2-6)

*Flow of control when one method calls another*



```
public ... main(...)
{  ...
   lisa.turnRight();
   ...
}
```

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

```
public void turnRight()
{  this.turnAround();
   this.turnLeft();
}
```

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

This discussion completes the `ExperimentRobot` class. The entire program is shown in Listing 2-4.

**Listing 2-4:** *The complete listing for* `ExperimentRobot`

FIND THE CODE

*ch02/experiment/*

```
import becker.robots.*;

// A new kind of robot that can turn around, turn right, and move forward
// three intersections at a time.
// author: Byron Weber Becker
public class ExperimentRobot extends Robot
{
  // Construct a new ExperimentRobot.
  public ExperimentRobot(City aCity, int aStreet,
                   int anAvenue, Direction aDirection)
  { super(aCity, aStreet, anAvenue, aDirection);
  }

  // Turn this robot around so it faces the opposite direction.
  public void turnAround()
  { this.turnLeft();
    this.turnLeft();
  }

  // Move this robot forward three times.
  public void move3()
  { this.move();
    this.move();
    this.move();
  }

  // Turn this robot 90 degrees to the right by turning around and then left by 90 degrees.
  public void turnRight()
  { this.turnAround();
    this.turnLeft();
  }
}
```

### 2.2.7 RobotSE

You can probably imagine other programs requiring robots that can turn around and turn right. `DeliverParcel` (Listing 1-1) could have used `turnRight` in one place, while `GoAroundRoadBlock` (Listing 1-2) could have used it twice. Several of the programming projects at the end of Chapter 1 could have used either `turnAround` or `turnRight` or both.

**LOOKING AHEAD**

*You learn how to build your own package of useful classes in Chapter 9.*

When we write methods that are applicable to more than one problem, it is a good idea to add that method to a class where it can be easily reused. The `becker` library has a class containing commonly used extensions to `Robot`, including `turnRight` and `turnAround`. It's called `RobotSE`, short for "Robot Special Edition." In the future, you may want to extend `RobotSE` instead of `Robot` so that you can easily use these additional methods.

### 2.2.8 Extension vs. Modification

**LOOKING AHEAD**

*The `turnRight` method in `RobotSE` actually turns right instead of turning left three times. Section 3.6 explains how.*

Another approach to making a robot that can turn around and turn right would be to modify the existing class, `Robot`. Modifying an existing class is not always possible, and this is one of those times. The `Robot` class is provided in a library, without source code. Without the source code, we have nothing to modify. We say that the `Robot` class is **closed for modification**.

There are other reasons to consider a class closed for modification, even when the source code is available. In a complex class, a company may not want to risk introducing errors through modification. Or the class may be used in many different programs, with only a few benefiting from the proposed modifications.

As we've seen, however, the `Robot` class is **open for extension**. It is programmed in such a way that those who want to modify its operation can do so via Java's extension mechanism. When a class is open for extension it can be modified via subclasses without fear of introducing bugs into the original class or introducing features that aren't generally needed.

## 2.3 Extending the `Thing` Class

**KEY IDEA**

*Classes other than `Robot` can also be extended.*

`Robot` is not the only class that can be extended. For example, `City` is extended by `MazeCity`. Instances of `MazeCity` contain a maze for the robots to navigate. At the end of this chapter you will find that graphical user interface components can be extended to do new things as well. In fact, every class can be extended unless its programmer has taken specific steps to prevent extension.

To demonstrate extending a class other than `Robot`, let's extend `Thing` to create a `Lamp` class. Each `Lamp` object will have two services, one to turn it "on" and another to turn it "off." When a lamp is "on" it displays itself with a soft yellow circle and when it is "off" it displays itself with a black circle. Because `Lamp` extends `Thing`, lamps behave like things—robots can pick them up, move them, and put them down again.