Chapter 6 Using Variables

Chapter Objectives

After studying this chapter, you should be able to:

- Add new instance variables to a simple version of the `Robot` class
- Store the results of calculations in temporary variables and use those results later in the method
- Write methods using parameter variables
- Use constants to write more understandable code
- Explain the differences between instance variables, temporary variables, parameter variables, and constants
- Extend an existing class with new instance variables

Every computer program stores and updates information. When we tell a robot to move, it updates its current street and avenue information. When a robot picks up a thing from an intersection, the intersection updates its list of things, removing the thing the robot picked up. The robot also updates its list of things to include the new thing it picked up.

A variable is a place in the computer's memory where information can be stored. When stored in a variable, the information can be changed, copied, or used in an expression. Programming languages offer several kinds of variables. The best one to use depends on factors such as how long the information must be stored and the source of the first value to store.

6.1 Instance Variables in the `Robot` Class

If we could look inside the `Robot` class, what would we find? We would certainly find methods implementing the `move` and `turnLeft` services. We would find that it extends another class that provides basic functionality on which robots depend. We would also find several varieties of **variables**. Some of these variables specify the street and avenue currently occupied by the robot. A variable is like a box that can hold one piece of information. We can ask for a copy of the information in the box anytime we like. We can also replace the information in the box with new information.

KEY IDEA

Variables store information for later use.

In this chapter, we will write a simplified version of the `Robot` class, named `SimpleBot`, to see its variables in action. It will also use a simplified version of `City`, named `SimpleCity`. By the end of this chapter, you will be able to understand all of the classes used except for three that are intimately involved with displaying the robots on the screen. By the end of Chapter 13, you will be able to understand those three as well.

You are strongly encouraged to download these classes from the software downloads section of the Robots Web site (*www.learningwithrobots.com/software/downloads.html*). You will increase your understanding if you write and run the programs as we develop the `SimpleBot` class.

FIND THE CODE

cho6/simpleBots/

We will spend most of our time developing the `SimpleBot` class, but you need a brief introduction to the `SimpleCity` class for everything to make sense. The `SimpleCity` class is a container for all the things that are "in" the city and need to be displayed by the city. In our simple version, the city only contains intersections and robots.

The intersections and robots in the city are displayed by calling their `paint` method. In the `SimpleBot` class, the `paint` method paints a robot; in the `SimpleIntersection` class, the `paint` method paints a street and an avenue. We will guarantee to the `SimpleCity` object that the objects we ask it to display have a `paint` method by requiring these paintable objects to extend a class named `Paintable`. This class is very simple: It extends `Object` and has a single method that does nothing—the `paint` method. Classes that should display themselves override this method. The `Paintable` class is shown, in its entirety, in Listing 6-1.

LOOKING AHEAD

In Section 7.6, we will see how the presence of particular methods can be assured with Java interfaces.

Listing 6-1: *The complete source code for the* `Paintable` *class*

FIND THE CODE

cho6/simpleBots/Paintable.java

```
1  import java.awt.Graphics2D;
2
3  /** Subclasses of Paintable can be displayed in the city. Each subclass should
4  *     override the paint method to paint an image of itself.
5  *     @author Byron Weber Becker */
6  public class Paintable extends Object
7  {
```

Listing 6-1: *The complete source code for the* `Paintable` *class* (continued)

```
 8    public Paintable()
 9    { super();
10    }
11
12    /** Each subclass should override paint to paint an image of itself. */
13    public void paint(Graphics2D g)
14    {
15    }
16 }
```

The city displays the intersections and the robots by calling `paint` about twenty times each second, first for the intersections and then for the robots. If a robot has moved since the last time the city was displayed, painting the intersections will erase the old robot image, and painting the robot will position it in its new location.

The following sections concentrate on the `SimpleBot` class and, in particular, how it uses variables to store and manipulate the information a robot needs. The robots in this section are simple—they only move and turn left. Eventually you will be able to increase their capabilities substantially.

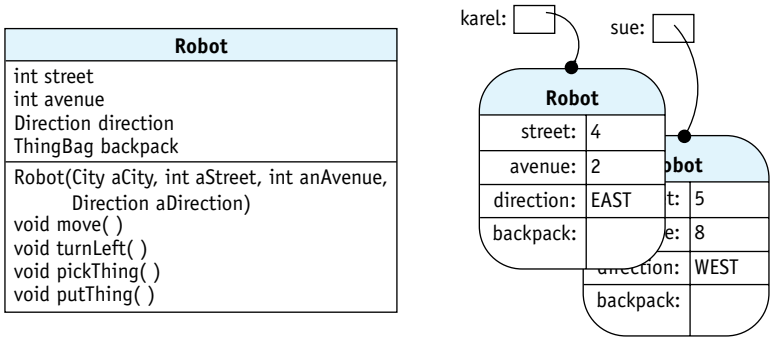
KEY IDEA

Start simply. Add functionality in small increments.

Our approach is to start simply, adding functionality in small increments. First, we'll display a round "robot" on intersection (4, 2). Then we'll make it move and turn left—except that we won't be able to tell which way it faces (because it is displayed as a circle) until it moves again. We will then improve its appearance so that it shows which direction it's facing, and enhance its functionality in other ways.

6.1.1 Implementing Attributes with Instance Variables

We know from our previous experience with robots that they have attributes that specify the street and avenue they currently occupy. In Figure 1-8, reproduced in Figure 6-1, we were introduced to a `Robot` class diagram showing these attributes. The instances of the class, as shown on the right side of Figure 6-1, have specific values for these attributes. Recall that each instance has its own copies of the attributes defined by the class. Each individual robot has its own street and avenue, for example.



(figure 6-1)

A `Robot` *class diagram, reproduced from Chapter 1, and two object diagrams corresponding to two possible instances*

When a `Robot` object paints itself on the city, it evidently looks at the street and avenue attributes to determine where to paint the image. If the attributes hold the values `4` and `2`, respectively, then the robot image is painted on the intersection of 4th Street and 2nd Avenue.

The idea of an attribute is implemented in Java with an **instance variable**. You can imagine an instance variable as a box that has a name. Inside the box can be one, and only one, value. When the name of the box is used in the code, a copy of the value currently inside the box is retrieved and used. An instance variable also allows us to change the value inside the box.

KEY IDEA

An instance variable stores a value, such as 10 or –15, for later use.

Instance variables have the following important properties:

- Each object has its own set of instance variables. Each robot, for example, has its own street and avenue variables to remember its location.
- The scope of an instance variable extends throughout the entire class. An instance variable can be used within any method.
- The **lifetime** of an instance variable—the length of time its values are retained—is the same as the lifetime of the object to which it belongs.

KEY IDEA

Each object has its own set of instance variables.

6.1.2 Declaring Instance Variables

We create and name an instance variable—a named "box" that holds a value—with a **variable declaration**. The declaration is often combined with an assignment statement to specify the variable's initial value. Instance variables are declared inside the classes' first and last braces but outside of any methods. The beginnings of the `SimpleBot` class, including two instance variables to hold the street and avenue, are shown in Listing 6-2.

KEY IDEA

A variable declaration sets aside space in memory to store a value and associates a name with that space.

Instance Variable

Listing 6-2: *The beginnings of the* `SimpleBot` *class with two instance variables declared*

```
1  public class SimpleBot extends Paintable
2  {
3    private int street = 4;   // Create space to store the robot's current street.
4    private int avenue = 2;   // Create space to store the robot's current avenue.
5
6    public SimpleBot()
7    { super();
8    }
9
10   // An incomplete class!
11 }
```

These instance variable declarations have four key parts that occur in the following order:

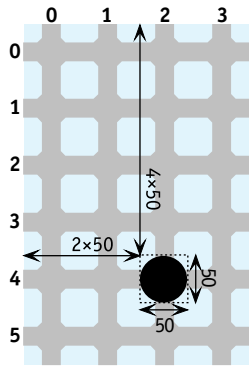
- Declarations start with an access modifier. For reasons we will explore in Chapter 11, the keyword `private` should be used almost exclusively. Like using `private` before a helper method, this keyword identifies this part of the class as "for internal use only."
- Declarations specify the type of values stored. The `int` says that these "boxes" hold integers—values such as 1, 33, or -15. Later, we'll study other possibilities, such as `double` (values such as 3.14159) and `String` (values such as `"I love Java"`).
- Declarations name the variable. In these examples, the names are `street` and `avenue`. Instance variables are generally named like methods, using one or more descriptive words, with the first letter of the entire name being lowercase and the first letter of subsequent words being uppercase. Examples include `avenue`, `direction`, and `nextLocation`.
- Declarations may include an initial value, placed after an equal sign. In these examples, `street` and `avenue` are given initial values of `4` and `2`, respectively. If the initial value is not explicitly assigned, Java will provide a default initial value appropriate to the type. The default for integers is `0` and for `boolean` variables is `false`. However, your code is more understandable if you explicitly initialize your variables.

These declarations are very similar to declaring temporary variables, as studied in Section 5.2, with two exceptions. First, instance variable declarations always occur outside of methods, whereas temporary variable declarations always occur inside of methods. Second, instance variable declarations should have an access modifier, whereas temporary variables never do.

6.1.3 Accessing Instance Variables

Two major tasks remain in writing the `SimpleBot` class. First, we need to display the robot within the city. To do this, we will access the values stored in the instance variables. Second, we need to make the robot move. We will do this by updating the values stored in the instance variables. Then, when the robot is painted again, it will appear at a different place in the city.

As we learned in the introduction to Section 6.1, objects to be displayed by a city must extend `Paintable` and override the `paint` method. Our first version of `paint` displays the robot as a black circle on the intersection of 4th Street and 2nd Avenue. Figure 6-2 shows the robot on a background of streets and avenues with annotations for drawing it. We will assume that each intersection is 50 pixels square. Therefore, 4th Street starts at pixel 200 (4×50) and 2nd Avenue starts at pixel 100 (2×50). The following code paints the robot and should be inserted between lines 10 and 11 of Listing 6-2.



(figure 6-2)

Simple robot and its location

```
1  public void paint(Graphics2D g)
2  { g.setColor(Color.BLACK);
3    g.fillOval(100, 200, 50, 50);
4  }
```

This method takes a parameter, `g`, which is an object used to paint on the screen. Line 2 says to use the color black in subsequent painting operations. Line 3 says to paint a solid oval. Recall that the first argument to `fillOval` is the *x* (horizontal) coordinate, the second argument is the *y* (vertical) coordinate, and the last two arguments are the height and width, respectively.

LOOKING AHEAD

We will discuss parameters in depth in Section 6.2.2.

There is no need for us to perform the multiplications to calculate the coordinates of the upper-left corner. Computers are very good at multiplication, and we should let them do that for us. Line 3 may be replaced by the following:

```
3  g.fillOval(2 * 50, 4 * 50, 50, 50);
```

Java uses `*` to indicate multiplication.

However, we don't *always* want to draw the robot at intersection (4, 2). We want to access the street and avenue attributes—implemented as instance variables—to determine where the robot is drawn. To do so, we use the names of the instance variables in place of the `4` and `2` in the new lines of code. That is, the following `paint` method will display the robot at the street and avenue specified in the instance variables.

Instance Variable

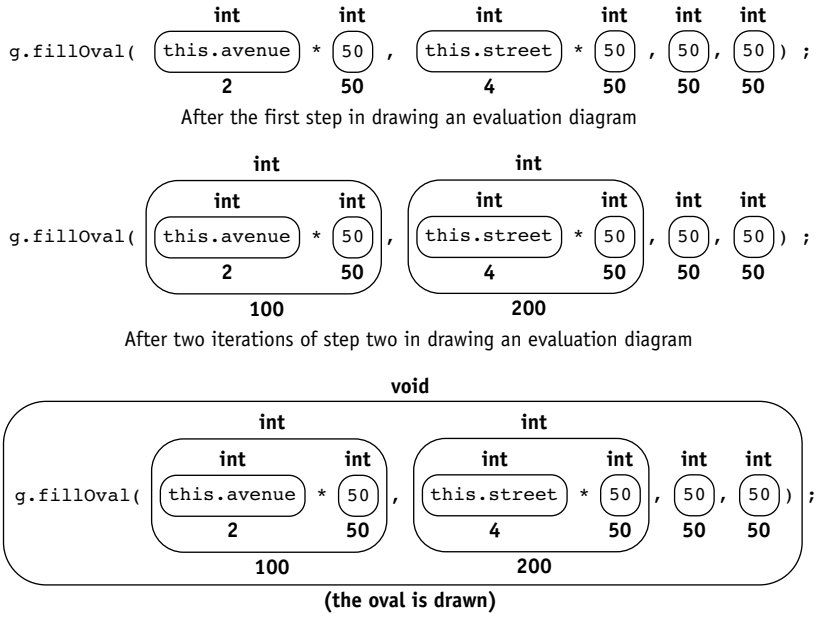
```
1 public void paint(Graphics2D g)
2 { g.setColor(Color.BLACK);
3   g.fillOval(this.avenue * 50, this.street * 50, 50, 50);
4 }
```

We will use the keyword `this` to access the instance variables in our code. Using `this` to access an instance variable is like using `this` to access a helper method: It reinforces that the variable belongs to *this* object, the one that contains the currently executing code.[1]

Line 3 of the preceding code can be better understood with the help of an evaluation diagram, like the one we used in Section 5.4. Recall that we begin by drawing ovals around literals (like `50`) and variables (like `this.avenue`), and writing their type above and their value below the ovals. We then repeatedly circle method calls and operators, together with their arguments and operands, in order of their precedence. This process is shown in Figure 6-3, where we assume that the robot is on (4, 2).

Notice that the arguments to `fillOval` are circled before `fillOval` is circled. This means that the arguments are evaluated before the method is called. Also note that the type of the oval around `fillOval` is `void` because `fillOval` has a return type of `void`. It doesn't return a value to place below the oval. Instead, the side effect of the method is written.

[1] The keyword `this` is actually optional much of the time. Students are strongly encouraged to use it, however, to reinforce that they are accessing an instance variable that belongs to a particular object. There is also the practical reason that many modern programming environments display a list of instance variables and methods when "this." is typed, reducing the burden on the programmer's memory and eliminating many spelling mistakes.



(figure 6-3)

Evaluation diagram for line 3 in `SimpleBot`*'s* `paint` *method, assuming the robot is on intersection (4, 2)*

The code for the `paint` method needs one last detail: The classes `Graphics2D` and `Color` must be imported before we can use them in lines 1 and 2, respectively. To do so, add the following two lines at the beginning of the file:

```
import java.awt.Graphics2D;
import java.awt.Color;
```

6.1.4 Modifying Instance Variables

Our robot needs a `move` method. When it is invoked, the robot should move to another intersection. From previous experience, we know that invoking `move` changed a robot's attributes. Because attributes are implemented with instance variables, we now know that `move` must change either `street` or `avenue` by 1, depending on the direction.

LOOKING BACK

Take a look at the state change diagram in Figure 1-12 to better understand the effect of `move` *on the avenue and street attributes.*

We already discussed incrementing and decrementing parameters and temporary variables in Chapters 4 and 5. Changing an instance variable by 1 is similar and is shown in the following partially implemented `move` method:

```
// Move the robot one intersection east, assuming it is facing east and nothing blocks it.
public void move()
{ this.avenue = this.avenue + 1;      // incomplete
}
```

PATTERN

Instance Variable

Because we are accessing an instance variable, a variable that belongs to an object, we use `this.` before the variable name.

Recall that an assignment statement works in two steps. First, it calculates the value of the expression to the right of the equal sign. Second, it forces the variable on the left of the equal sign to store whatever value was calculated. The variable continues to store that value until it is changed with another assignment statement. The assignment statements used with parameter and temporary variables in Chapters 4 and 5 behaved the same way.

How does this process move the robot? The entire city is repainted about 20 times per second with a loop such as the following:

```
while (true)
{  paint everything in layer 0 (the intersections)
   paint everything in layer 1 (the things)
   paint everything in layer 2 (the robots)
}
```

When `move` is called, the entire city is repainted within about 50 milliseconds. Repainting the intersections has the effect of erasing the robot's old image at its old location. Shortly thereafter, the robot's `paint` method is called. It paints the robot's image in its new location, as determined by the current values of `street` and `avenue`. The effect is that the robot appears to move on the screen.

But there is a problem. Executing the `move` method takes far less than 50 milliseconds. If several consecutive `move` instructions are executed, we won't see most of them because they occur in the time between repainting the screen. To solve this problem, we need to ensure that `move` takes at least 50 milliseconds to execute. This is done by instructing the `move` method to sleep, or do nothing, for a while. The `becker` library contains a method to sleep for a specified number of milliseconds. To use it, import `becker.util.Utilities` and add `Utilities.sleep(400)` to the `move` method. The robot will then stop for 0.400 seconds each time it moves.

The code for the `SimpleBot` class, as developed so far, is shown in Listing 6-3. Robots instantiated from this class will always start out on intersection (4, 2) and can only travel east. We will remove these restrictions soon.

FIND THE CODE

cho6/simpleBots/SimpleBot.java

Listing 6-3: *The* `SimpleBot` *class, as developed so far*

```
1  import java.awt.Graphics2D;
2  import java.awt.Color;
3  import becker.util.Utilities;
4
```

Listing 6-3: *The* `SimpleBot` *class, as developed so far* (continued)

```
5  /** A first try at the SimpleBot class. These robots are always constructed on street 4,
6  *   avenue 2. There is no way to tell which way they are facing and they can only move east.
7  *
8  *   @author Byron Weber Becker */
9  public class SimpleBot extends Paintable
10 {
11   private int street = 4;
12   private int avenue = 2;
13
14   /** Construct a new Robot at (4, 2). */
15   public SimpleBot()
16   { super();
17   }
18
19   /** Paint the robot at its current location. */
20   public void paint(Graphics2D g)
21   { g.setColor(Color.BLACK);
22     g.fillOval(this.avenue * 50, this.street * 50, 50, 50);
23   }
24
25   /** Move the robot one intersection east. */
26   public void move()
27   { this.avenue = this.avenue + 1;
28     Utilities.sleep(400);
29   }
30
31   /** Turn the robot 90 degrees to the left. */
32   public void turnLeft()
33   {
34   }
35 }
```

6.1.5 Testing the `SimpleBot` Class

The `main` method to test this class is slightly different from the ones we've written in previous chapters, in which we passed the city to the robot's constructor. The constructor then added the robot to the city. The constructor in Listing 6-3 isn't that sophisticated (yet). Therefore, we must add the robot to the city in the `main` method, specifying that it appears in layer 2 so that it is painted after the intersections (layer 0) and things (layer 1). A second change is to explicitly wait for the user to press the Start button before moving the robots. These two details are at lines 11-14 of Listing 6-4.

The `SimpleBot` classes we are discussing are *not* part of the `becker` library. Therefore, to compile the program, you will not be importing classes from the library. Instead, you need to have the source code for `SimpleBot`, `SimpleCity`, and several others in the same directory as the `TestSimpleBot` class shown in Listing 6-4. Recall that all of this source code is available from the Robots Web site (*www.learningwithrobots.com/software/downloads.html*). However, you will need to implement much of the code for the `SimpleBot` class yourself.

FIND THE CODE

cho6/simpleBots/Main.java

Listing 6-4: *A* `main` *method to test the* `SimpleBot` *class*

```
1  /** A main method to test the SimpleBot and related classes.
2  *
3  * @author Byron Weber Becker */
4  public class Main
5  {
6    public static void main(String[] args)
7    { SimpleCity newYork = new SimpleCity();
8      SimpleBot karel = new SimpleBot();
9      SimpleBot sue = new SimpleBot();
10
11     newYork.add(karel, 2);
12     newYork.add(sue, 2);
13
14     newYork.waitForStart();     // Wait for the user to press the Start button.
15
16     for(int i=0; i<4; i = i+1)
17     { karel.move();
18       karel.move();
19       karel.turnLeft();
20     }
21
22     sue.move();
23   }
24 }
```

6.1.6 Adding Another Instance Variable: `direction`

LOOKING AHEAD

In Section 7.4, we will look at a detailed example that has nothing to do with robots.

So far we've seen how to declare, initialize, access, and modify instance variables to implement the street and avenue attributes for a robot. Keep in mind that instance variables are also used to implement classes that have nothing to do with robots: bank accounts, employees, properties for a Monopoly game, and so on.

Right now, however, let's implement another attribute of robots: direction. When we're done, the robots will be able to turn left and move in the direction they are facing.

Representing Directions

Our basic plan is to use a new instance variable, `direction`, to store the direction the robot is facing. `direction` will be an integer. When it has a value of 0, the robot is facing east; 1 means the robot is facing south, 2 is west, and 3 is north. Turning left is as easy as subtracting 1 from `direction`—unless the robot is facing east (0). Then we need to wrap around and set `direction` to north (3). As with the `move` method, forcing the robot to sleep after turning allows us to see what has happened.

Listing 6-5 shows the addition of the `direction` instance variable and the `turnLeft` method in a skeleton of the `SimpleBot` class.

Listing 6-5: *Changes to the* `SimpleBot` *class to add the* `turnLeft` *service*

```
1  public class SimpleBot extends Paintable
2  { ...
3    private int direction = 0;   // Begin facing east.
4    ...
5
6    /** Turn the robot left 1/4 turn. */
7    public void turnLeft()
8    { if (this.direction == 0)          // If facing east...
9      { this.direction = 3;             // face north.
10     } else
11     { this.direction = this.direction - 1;
12     }
13     Utilities.sleep(400);
14   }
15 }
```

Using the `final` Keyword with Instance Variables

Remembering that `0` means east and `3` means north makes `turnLeft` difficult to understand. Listing 6-5 compensates with comments, but we can do better. One approach is to declare four new instance variables, as follows:

```
private int east = 0;
private int south = 1;
private int west = 2;
private int north = 3;
```

Now we can rewrite lines 8 and 9 as follows:

```
8  { if (direction == this.east)
9    { this.direction = this.north;
```

However, these "variables" seem different from instance variables such as `avenue` and `direction` because they should not change while the program executes. We should always use `0` to mean east, and it would be a programming error if `east` ever had a different value.

KEY IDEA

Use the `final` keyword when a variable should never be assigned a new value.

Java uses the keyword `final` to indicate that the first value a variable receives should also be the final value it ever receives. If we try to change the variable's value, Java will issue a compile-time error. Such variables are often called **constants**. It is traditional to use all uppercase characters to name constants to emphasize that they are unchanging, as follows:

```
private final int EAST = 0;
```

Named Constant

Another useful constant would be `INTERSECTION_SIZE`, to be used in the `paint` method in place of `50`. Notice the underscore character separating the individual words that make up the name.

In addition to making the code easier to read, constants are useful because they provide one place to change when assumptions change. For example, we assumed that intersections are 50 pixels square. If we ever need to display larger cities, we may want to change it to 40 pixels. Finding and changing one constant is much easier than finding and changing every place the value 50 is used in the program.

Using the `static` Keyword with `final` Instance Variables

LOOKING AHEAD

Using `static` with non-`final` instance variables is discussed in Section 7.5.1.

A second keyword, `static`, is often used with `final` instance variables. It allows programmers to access the variable using the class name rather than an object reference. For example, suppose `EAST` were declared as follows:

```
public static final int EAST = 0;
```

Programmers could then use it like this:

```
if (this.direction == SimpleBot.EAST)
```

Named Constant

This may not seem like much of an improvement, but if the variable is `public`, then it can be used from any class without using an object. We have, in fact, done this already in the `main` method of graphics programs when we use `JFrame.EXIT_ON_CLOSE` to set the frame's default close operation.

Sometimes constants are used in many different classes. In such cases, it can make sense to have a class named something like `Constants` that contains nothing but public constants.

Finishing the `move` Method

Now that we can easily represent directions using `final` instance variables and can change a robot's direction with the `turnLeft` method, we must reimplement the `move` method so that it actually moves in the correct direction.

Each time the robot moves, we will adjust both the street and the avenue by the values shown in Table 6-1.

Direction	Change `street` by	Change `avenue` by
`EAST`	0	1
`WEST`	0	-1
`NORTH`	-1	0
`SOUTH`	1	0

(table 6-1)

Adjustments to `street` *and* `avenue` *when moving in each direction*

This is a perfect job for two helper methods, `strOffset` and `aveOffset`. They use a cascading-`if` statement to set a temporary variable to the appropriate offset, based on testing the value stored in the `direction` instance variable. They then return that value using a `return` statement, just like the queries written in Section 5.2.4.

The new `direction` instance variable, the new `turnLeft` method, the modified `move` method, and the two helper methods are all shown in Listing 6-6. The class assumes that appropriate constants have been declared in a class named `Constants`. The `turnLeft` method uses two additional constants to clarify why they constitute a special case. They are declared in `Constants` as follows:

```
public static final int FIRST_DIR = EAST;
public static final int LAST_DIR = NORTH;
```

6.1.7 Providing Accessor Methods

Methods that provide access to private instance variables are called **accessor methods**. An accessor method is a query that answers the question "What value does attribute *X* currently hold?" That is, it makes the value stored in an instance variable accessible to code outside of the class.

You can use the following pattern to write an accessor method:

```
public «typeReturned» get«Name»()
{ return this.«instanceVariable»;
}
```

PATTERN

Accessor Method

«typeReturned» specifies what kind of value the method returns. It should be the same type as the instance variable itself. The variables `avenue`, `street`, and `direction` are all integers, so their accessor methods will have `int` as a return type.

«Name» is usually the name of the instance variable. It should be a name that is meaningful to users of the class.

Finally, *«instanceVariable»* is the name of the appropriate instance variable to access.

Three examples of accessor methods, one each for `street`, `avenue`, and `direction`, are shown in Listing 6-6.

Listing 6-6: *A* `SimpleBot` *class that includes the ability to turn left*

```
1  import java.awt.Graphics2D;
2  import java.awt.Color;
3  import becker.util.Utilities;
4
5
6  /** A second try at the SimpleBot class. These robots are always constructed at (4, 2) facing
7  *  east. Robots can move forward and turn left, although the user cannot determine which
8  *  way the robot is facing until it moves.
9  *
10 *  @author Byron Weber Becker */
11 public class SimpleBot extends Paintable
12 {
13   private int street = 4;
14   private int avenue = 2;
15   private int direction = Constants.EAST;
16
17   /** Construct a new robot at (4, 2) facing east. */
18   public SimpleBot()
19   { super();
20   }
21
22   /** Paint the robot at its current location. */
23   public void paint(Graphics2D g)
24   { g.setColor(Color.BLACK);
25     g.fillOval(this.avenue * Constants.INTERSECTION_SIZE,
26                this.street * Constants.INTERSECTION_SIZE,
27                Constants.INTERSECTION_SIZE,
28                Constants.INTERSECTION_SIZE);
29   }
30
```

Listing 6-6: *A* `SimpleBot` *class that includes the ability to turn left* (continued)

```
31    /** Move the robot forward 1 intersection. */
32    public void move()
33    { this.street = this.street + this.strOffset();
34      this.avenue = this.avenue + this.aveOffset();
35      Utilities.sleep(400);
36    }
37
38    /** Turn the robot left 1/4 turn. */
39    public void turnLeft()
40    { if (direction == Constants.FIRST_DIR)
41      { this.direction = Constants.LAST_DIR;
42      } else
43      { this.direction = this.direction - 1;
44      }
45      Utilities.sleep(400);
46    }
47
48    /** Get this robot's street.
49    *   @return The street this robot is currently on. */
50    public int getStreet()
51    { return this.street;
52    }
53
54    /** Get this robot's avenue.
55    *   @return The avenue this robot is currently on. */
56    public int getAvenue()
57    { return this.avenue;
58    }
59
60    /** Get this robot's direction.
61    *   @return The direction this robot is facing. */
62    public int getDirection()
63    { return this.direction;
64    }
65
66    /** Calculate how far the robot should move along the avenue.
67     *  @return {-1, 0, or 1} */
68    private int aveOffset()
69    { int offset = 0;
70      if (this.direction == Constants.EAST)
71      { offset = 1;
72      } else if (this.direction == Constants.WEST)
73      { offset = -1;
```

Listing 6-6: *A* `SimpleBot` *class that includes the ability to turn left* (continued)

```
74        }
75        return offset;
76     }
77
78     /** Calculate how far the robot should move along the street.
79      * @return {-1, 0, or 1} */
80     private int strOffset()
81     { int offset = 0;
82       if (this.direction == Constants.NORTH)
83       { offset = -1;
84       } else if (this.direction == Constants.SOUTH)
85       { offset = 1;
86       }
87       return offset;
88     }
89  }
```

6.1.8 Instance Variables versus Parameter and Temporary Variables

Like parameter and temporary variables, instance variables store a value. They are also different in important ways. We will have more to say about these similarities and differences in Section 6.5, but for now, remember the following:

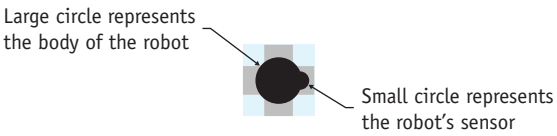
- Instance variables are declared inside a class but outside of all methods. Parameter and temporary variables are declared inside a method.
- Instance variables have a larger scope. They may be used within any of the methods in the class. Parameter and temporary variables can be used only within the method in which they are declared.
- Instance variables have a longer lifetime. They retain their value until changed by an assignment statement or until the object is no longer in use. Parameter and temporary variables disappear when their method finishes executing and are reinitialized each time the method executes again.

6.2 Temporary and Parameter Variables

Temporary and parameter variables were introduced in Chapters 4 and 5, respectively. In this section, they are used extensively to improve the `SimpleBot`. We will also use them with more complex expressions and apply the `final` keyword to them.

6.2.1 Reviewing Temporary Variables

Right now our robots are displayed with a black oval that covers the entire intersection. We can't tell which direction the robot is facing unless it moves. In this section, we will upgrade our robot to correct these problems. Our new, improved robot will appear as shown in Figure 6-4.

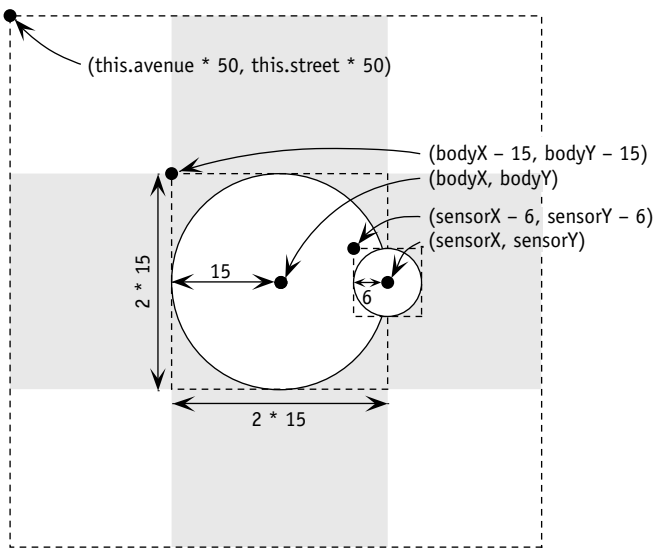


(figure 6-4)

Robot showing its direction

The large circle, representing the body of the robot, is centered on the middle of the intersection and has a radius of 15 pixels. The smaller circle, representing the robot's sensor, is centered on the perimeter of the larger circle with a radius of 6 pixels.

Because the size of the circle no longer matches the size of the intersection, more work will be required to paint the robot. Figure 6-5 shows relevant values that we will need to calculate. They depend heavily on the center of the robot's body and the center of the sensor. We will find it useful to calculate and store these values in temporary variables.



(figure 6-5)

Drawing a circle, given its center and radius

LOOKING BACK

Temporary variables were introduced in Section 5.2.

Before we proceed, let's recall what we know about temporary variables:

- They are declared inside a method.
- Declarations have a type, a name, and usually an initial value. For example, `int numThingsHere = 0`. Declarations do not include an access modifier.
- The value stored by the variable is accessed with just the variable's name; it is not prefixed with `this`.
- The scope of a temporary variable—the region in which it can be used—extends from its point of declaration to the end of the smallest enclosing block.
- Each time the variable's block is executed, the variable is created and reinitialized; each time execution exits the block, the variable disappears.

LOOKING BACK

Scope and block were defined in Section 5.2.6.

Listing 6-7 provides a skeleton for the `paint` method. It declares four temporary variables to store the center coordinates of the body and the sensor in lines 5-8. Their initialization is shown in pseudocode.

Listing 6-7: *A skeleton of the* `paint` *method*

```
1  /** Paint the robot at its current location. */
2  public void paint(Graphics2D g)
3  { g.setColor(Color.BLACK);
4
5     int bodyX = x coordinate of robot body's center
6     int bodyY = y coordinate of robot body's center
7     int sensorX = x coordinate of robot sensor's center
8     int sensorY = y coordinate of robot sensor's center
9
10    // Draw the robot's body.
11    g.fillOval(bodyX - 15, bodyY - 15, 2 * 15, 2 * 15);
12
13    // Draw the robot's sensor.
14    g.fillOval(sensorX - 6, sensorY - 6, 2 * 6, 2 * 6);
15 }
```

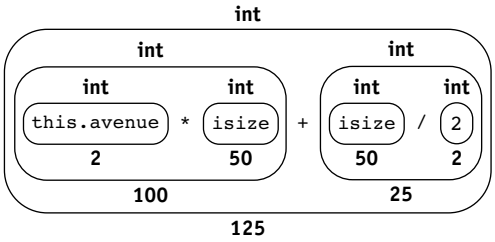
The values in these four variables are used in lines 11 and 14 to paint the two circles representing the robot's body and sensor. Recall that `fillOval`'s first two arguments represent the upper-left corner of the smallest rectangle that will include the oval, shown with dotted lines in Figure 6-5. The expression `bodyX - 15` in line 11 uses the center of the circle to calculate the left edge of the body's enclosing rectangle. `bodyY - 15` calculates the top edge of the body's enclosing rectangle.

Calculating the Body's Center

The center of the robot's body is the same as the center of the intersection. To calculate it, we can calculate the intersection's upper-left corner and then add one half of the intersection's width and height. Recall that the intersection's size is stored in `Constants.INTERSECTION_SIZE`. This name is unwieldy to use repeatedly in a method, so we first assign it to a temporary variable with a shorter name.

```
int iSize = Constants.INTERSECTION_SIZE;
int bodyX = this.avenue * iSize + iSize / 2;
int bodyY = this.street * iSize + iSize / 2;
```

We can increase our confidence that these calculations are correct by producing an evaluation diagram with some sample values for the robot's location and intersection size. For example, see Figure 6-6.



(figure 6-6)

Evaluation diagram for `bodyX` *when the robot is at (4, 2) on intersections of size 50*

One question that arises is what happens when two numbers do not divide evenly. For example, what would the preceding expression produce if the intersection size was 51 instead of 50? One might expect an answer of 125.5 because 51/2 is 25.5—but that answer is wrong.

Java performs **integer division** when both operands are integers. Integer division is like the long division you learned in grade school, but with the remainder thrown away. That is, 51 divided by 2 is 25 with a remainder of 1. The remainder is thrown away, and the answer is 25. Java has a second kind of division that preserves the decimal portion. We will study it in Section 7.2.2.

KEY IDEA

Dividing two integers results in an integer. The decimal portion, if any, is lost.

If the divisor (the second number) happens to be 0, an exception will be thrown to indicate that the division can't be performed.

A related operator is `%`, the **remainder operator**. It returns the remainder of the long division. For example, `51 % 2` returns `1` because 51 divided by 2 is 25 with a remainder of 1. If the first operand happens to be negative, the answer will be negative as well.

KEY IDEA

`n % d` *gives the remainder of dividing* `n` *by* `d`.

The remainder operator has four common uses in programming:

- The remainder operator can be used to determine if a number is even or odd, as in the following example:

```
if (n % 2 == 0)
{ // n is even...
```

- The remainder operator can be used to process every n^{th} item. For example, consider a robot traveling east until it finds a wall. The following code will place a `Thing` on every 5^{th} intersection.

```
while (karel.frontIsClear())
{ if (this.getAvenue() % 5 == 0)
  { this.putThing();
  }
}
```

- The remainder operator can be used together with the `/` operator to find the individual digits of a number. For example, `123 % 10` gives the right-most digit, `3`. Dividing by 10 gives the number without the right-most digit. For example, `123 / 10` gives `12`. Taking the remainder of this number gives the next digit, `2`, and so on.
- The remainder operator can be used to perform "wrap around" or "clock" arithmetic. We've already seen an example of this kind of arithmetic when we implemented `turnLeft`. We subtracted one from `direction`, unless the direction was `EAST` (`0`); then we wrapped around back to `NORTH` (`3`). The more common case is incrementing by one until an upper limit is reached, then starting over at 0. This calculation can be implemented as follows:

```
var = (var + 1) % upperLimit;
```

Calculating the Sensor's Center

We now turn to calculating the sensor's center, as required by lines 7 and 8 in Listing 6-7. Once again, we turn to Figure 6-5 for guidance. It appears that `sensorY` is the same as `bodyY` and that `sensorX` is the same as `bodyX + 15`, the body's radius.

Unfortunately, it isn't that simple. These calculations only work if the robot is facing east. Figure 6-7 shows the robot in all four positions with the associated calculations.

(figure 6-7)

Drawing the robot in each of the four directions

We could solve this problem with a cascading-`if` statement, but there is an easier way. This situation is similar to moving the robot. There, we wanted to add `-1`, `0`, or `1` to the street or avenue, depending on the direction the robot is facing. Here we want to add `-15`, `0`, or `15`. For the `move` method, we used two helper methods—`strOffset` and `aveOffset`. For this problem, we just need to multiply their results by `15`.

Lines 7 and 8 in Listing 6-7 can be replaced by the following two lines:

```
int sensorX = bodyX + this.aveOffset() * 15;
int sensorY = bodyY + this.strOffset() * 15;
```

Using the `final` Keyword with Temporary Variables

The `final` keyword can be used with any kind of variable, not just instance variables. It always means that the first value assigned to the variable should also be the final value assigned.

In the `paint` method, we assigned the constant `INTERSECTION_SIZE` to a temporary variable, `iSize`, for convenience. However, it would be a bug if `iSize` were mistakenly changed. For this reason, using `final` would be an excellent idea, as follows:

```
final int iSize = Constants.INTERSECTION_SIZE;
```

It's also worth noting that none of the temporary variables change after they are initialized. (They may, however, have a different value the next time the `paint` method is called and the variables are initialized again.) It wouldn't hurt to make the fact that they don't change while `paint` is executing explicit by using `final` for all of the temporary variables.

Delaying Initialization

It is possible to separate a temporary variable's declaration and initialization. This is useful, for example, if we use a cascading-`if` statement to calculate `sensorX`, as follows:

```
int sensorX;
if (this.direction == Constants.EAST)
{ sensorX = bodyX + 15;
} else if (this.direction == Constants.NORTH)
{ sensorX = bodyX;
...
```

When initialization is delayed, the temporary variable holds an unknown value between the time it is declared and when it is initialized. It would be an error to try to use it. Fortunately, the Java compiler actively tries to prevent this error. For example,

KEY IDEA

An uninitialized temporary variable holds an unknown value.

the following program fragment produces an error message saying "variable bodyX may not have been initialized."

```
int bodyX;
int sensorX = bodyX + 15;
```

Occasionally, the compiler will issue this error even though the variable is initialized in an `if` statement. In that case, simply initialize the variable when it is declared even though you know it will have a new value assigned before it is used.

Temporary Variable Summary

This `paint` method could have been written without temporary variables (see Listing 6-8). However, temporary variables allow us to break the calculation into smaller pieces. The pieces can be individually named and documented, making them easier to understand than one large calculation.

Another use of temporary variables is to reuse a calculation in several places in the same method. By performing the calculation once and storing the result, we save time and effort in programming and debugging.

Listing 6-8: *The* `paint` *method without temporary variables*

```
1  /** Paint the robot at its current location. */
2  public void paint(Graphics2D g)
3  { g.setColor(Color.BLACK);
4
5    // Draw the robot's body.
6    g.fillOval(this.avenue * 50 + 50/2 - 15,
7               this.street * 50 + 50/2 - 15,
8               2*15, 2*15);
9
10   // Draw the robot's sensor.
11   g.fillOval(
12         this.avenue * 50 + 50/2 + this.aveOffset() * 15 - 5,
13         this.street * 50 + 50/2 + this.strOffset() * 15 - 5,
15         2*5, 2*5);
16 }
```

KEY IDEA

Use temporary variables when you can; instance variables only if you must.

Temporary variables and instance variables are similar in that they both store a value that can be used later. Their major differences are in how long the value is stored and in where the value can be used. Because instance variables have a longer lifetime and a larger scope, they can often be used in place of temporary variables. This can lead to mistakes. The shorter lifetimes of temporary variables and their much smaller scope (a

method rather than the entire class) result in a much smaller opportunity for misuse. Instance variables are vitally important in object-oriented programming, but should only be used when other kinds of variables cannot be used.

6.2.2 Reviewing Parameter Variables

We were introduced to parameter variables in Section 4.6, where we wrote a method that took an argument specifying how far the robot should move. In this section, we will show how parameter variables are closely related to temporary variables, explore using parameters with constructors, and discuss overloading.

Parameter Variables versus Temporary Variables

Consider the following modification of the `move` method in the `SimpleBot` class. It causes the robot to move two intersections in the direction it is currently facing.

```
1 public void moveFar()
2 { int howFar = 2;
3   this.street = this.street + this.strOffset() * howFar;
4   this.avenue = this.avenue + this.aveOffset() * howFar;
5   Utilities.sleep(400);
6 }
```

A method to move the robot three intersections can be developed by copying `moveFar` to a new method, `moveReallyFar`, and changing the `2` in line 2 to `3`. Another method, `moveReallyReallyFar`, could be identical to `moveFar` except for setting `howFar` to `4`.

The methods are all identical except for that one number. This seems silly, for a number of reasons:

- What if we discover a bug in the first one—for example, if line 3 used `aveOffset()` instead of `strOffset()`? Chances are good that the same bug has been cut and pasted into the other methods.
- What if we want to move 7 intersections? We must define a new method, with a new name—and that still wouldn't help us move 25 intersections in another part of the program.
- What if we want to calculate the distance to move, storing it in a variable? We need to resort to a messy cascading-`if` or `switch` statement to choose the specific method to execute.

Instead of initializing `howFar` when we *write* the method, we want to initialize it when we *call* the method. Using parameter variables, we can accomplish this goal. Parameters allow us to replace `karel.moveFar()` with `karel.move(2)` and to replace `karel.moveReallyReallyFar()` with `karel.move(4)`. The argument—the number in the parentheses—specifies how far we want the robot to move. If we want the robot to move five intersections, we can write `karel.move(5)`.

KEY IDEA

The argument, provided when the method is called, is used to initialize the parameter variable.

The argument is used to initialize a parameter variable defined inside the `move` method. The parameter variable is similar to a temporary variable except that it is declared differently and is initialized by the argument.

Consider the temporary variable `howFar` from the `moveFar` method:

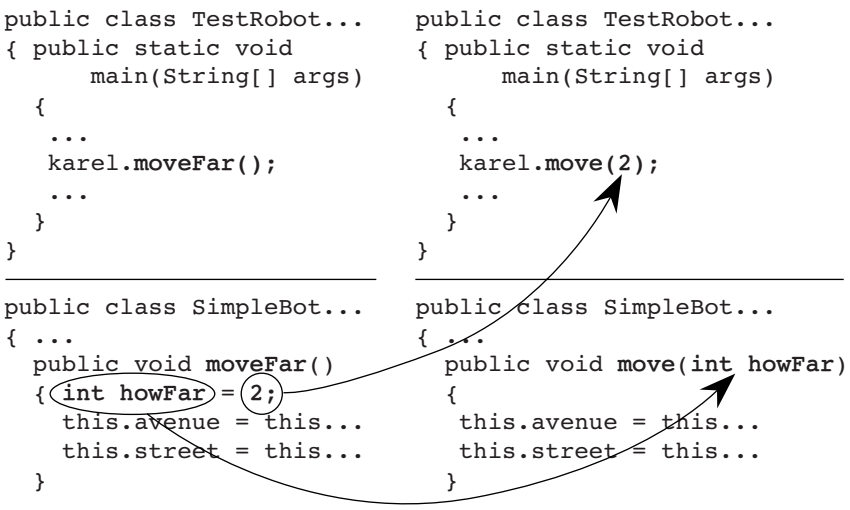
```
int howFar = 2;
```

To transform it into a parameter, think of its two distinct parts: the declaration and the initialization. The declaration, `int howFar`, stays inside the method, where it becomes the parameter variable. The value it is initialized with, `2`, becomes the argument. It is provided when the method is called. (The equal sign is discarded in the process.)

The left side of Figure 6-8 shows relevant portions of a program that uses `moveFar`. On the top is a `main` method that calls `moveFar`, and on the bottom is the definition of `moveFar`. The right side of the figure shows the program after transforming it to use a parameter variable.

(figure 6-8)

Transforming a temporary variable into a parameter variable



Inside the method, the parameter variable behaves like any other temporary variable. It can be used in expressions, passed as an argument to another method, and assigned a new value. Its scope is the entire method. Like a temporary variable, it has a short lifetime, disappearing when the method finishes executing. It is re-created and reinitialized each time the method is executed. The difference is in how it is initialized.

As we've seen in previous chapters, a method may have more than one parameter. For example, the following method is called with two arguments, `karel.move(5, Constants.EAST)`. It turns `karel` to face the specified direction and then move the specified distance. Each pair of declarations is separated with a comma.

```
public void move(int howFar, int aDir)
{ this.face(aDir);
  this.move(howFar);
}
```

Overloading Methods

We now have three methods named `move`, the usual one without a parameter, one with a single parameter, and one with two parameters. Fortunately, this does not usually cause a problem as long as every method in the class has a different **signature**. A method's signature is its name together with an ordered list of its parameter types.

The signature of the usual `move` method is simply `move()`. It has no parameters and hence its ordered list of parameter types is empty. The signature of the `move` method shown in the right side of Figure 6-8 is `move(int)`. Notice that the parameter name is not included in the signature. The last version of `move` has the signature `move(int, int)`.

Assuming `karel` is an instance of a `SimpleBot` class that has these three methods defined, `karel.move()`, `karel.move(3)`, and `karel.move(3, Constants.NORTH)` are all legal method calls. In each case, Java executes the method with the matching signature.

Methods and constructors that have the same name but different signatures are said to be **overloaded**. Note that we now have two terms incorporating the word "over":

- **Overload**—A method overloads another method in either a superclass or the same class when they have the same name but different signatures. Any of the methods may be executed, depending on the arguments provided when it is called.
- **Override**—A method in a subclass overrides a method in a superclass if they have the same signatures. The overriding method is executed and the overridden method is not (unless it is called by the overriding method).

Constructors may also be overloaded. The same principles apply to them.

Using Parameters to Initialize Instance Variables

Parameters are also useful for writing constructors. Our current implementation of `SimpleBot` always begins on 4th Street and 2nd Avenue facing east. We can use parameters in the constructor to make it more flexible.

Listing 6-9 shows a constructor with four parameters to construct a robot in a specified city at a specified location. Three of the parameters are used to provide the initial values to the instance variables `street`, `avenue`, and `direction`. Because the initial

KEY IDEA

Initialize each instance variable either where it is declared or in the constructor.

values are provided in the constructor, initial values are no longer needed on lines 2–4 where the variables are declared; there is no need to initialize them in both places.

Listing 6-9: *A version of the* `SimpleBot` *class that uses parameters to initialize its location*

```
1  public class SimpleBot extends Paintable
2  { private int street;
3    private int avenue;
4    private int direction;
5
6    /** Construct a new robot in the given city at the given location.
7     *   @param aCity                The city in which this robot appears.
8     *   @param aStreet              This robot's initial street.
9     *   @param anAvenue             This robot's initial avenue.
10    *   @param aDirection           This robot's initial direction. */
11   public SimpleBot(SimpleCity aCity,
12                    int aStreet, int anAvenue, int aDirection)
13   { super();
14     this.street = aStreet;
15     this.avenue = anAvenue;
16     this.direction = aDirection;
17     aCity.add(this, 2);     // Add this robot to the given city in the top level.
18   }
19   // Remainder of the class omitted.
20 }
```

One of the constructor's parameters—the city—is *not* used to initialize an instance variable. Recall that the robot must be added to the city, which keeps a list of all the objects to be painted. So far the robot has been added to the city in the `main` method. The following lines show how it was done in Listing 6-4.

```
 7  SimpleCity newYork = new SimpleCity();
 8  SimpleBot karel = new SimpleBot();
    ...
11  newYork.add(karel, 2);
```

With this new constructor, line 8 is changed as follows to place the robot in the city named `newYork` on 4th Street and 2nd Avenue, facing east:

```
 8  SimpleBot karel = new SimpleBot(newYork, 4, 2, Constants.EAST);
```

Line 11 is omitted from the `main` method because that task is now performed in the `SimpleBot` constructor. When the constructor is called as shown in the preceding code, the value stored in `newYork` is assigned to the parameter variable `aCity`. The reference to the newly created object is assigned to the implicit parameter variable

`this`. Both variables are used in line 17 of Listing 6-9 to add *this* robot to the city known within the constructor as `aCity`. The effect is the same as our previous approach, `newYork.add(karel, 2)`.

KEY IDEA

A reference to this object can be passed as a parameter using `this`.

Name Conflicts

It is often the case that the natural name for a parameter is the same as the name of an instance variable. For example, some people find the parameter names in lines 11 and 12 of Listing 6-9 awkward and would rather use names like `street` and `avenue`. In fact, the names of the parameters can be the same as the names of the instance variables. Using `this` removes the ambiguity that would otherwise be present. For example, lines 11–14 could be reimplemented as follows:

```
11 public SimpleBot(City city,
12                  int street, int avenue, Direction direction)
13 { super();
14   this.street = street;
...
```

A temporary variable may also have the same name as an instance variable, but temporary and parameter variables within the same method must have unique names.

There is, however, a danger in using the same names. As noted briefly earlier, `this` is actually optional in most circumstances, and many programmers, unfortunately, habitually omit it. Omitting `this` when the parameter name and instance variable name are different poses no danger. But suppose `this` was omitted from line 14 of the preceding code, as follows:

```
14 street = street;
```

The compiler would interpret this as assigning the value in the parameter to itself—a useless but perfectly valid action. The instance variable would remain uninitialized.

Using the `final` Keyword with Parameter Variables

Like other kinds of variables, parameter variables can use the keyword `final`. As elsewhere, it means that the variable's value may not be changed. As with other kinds of parameters, use `final` to emphasize and enforce that intention.

6.3 Extending a Class with Variables

In Section 6.1, we saw how instance variables can be used inside a class such as `SimpleBot`. It is also possible to extend an existing class with new instance variables,

KEY IDEA

Instance variables in a subclass add to the information maintained by the superclass.

just as we extended an existing class with new methods in Chapter 2. In defining the new class, we will specify only the new instance variables. The Java compiler will automatically include them with the instance variables already defined in the superclass.

In the following example, we will extend `Robot` (*not* the `SimpleBot` class used earlier in this chapter) to create a new class, `LimitedBot`. Our goal is to create a kind of robot that can carry only a limited number of things; if it attempts to carry more, it will break. Each of these limited robots will need to know two pieces of information: How many things it can hold before breaking, and how many things it is currently holding. We'll call one `maxHold` (the maximum the robot can hold at one time) and call the other `numHeld` (the number held right now).

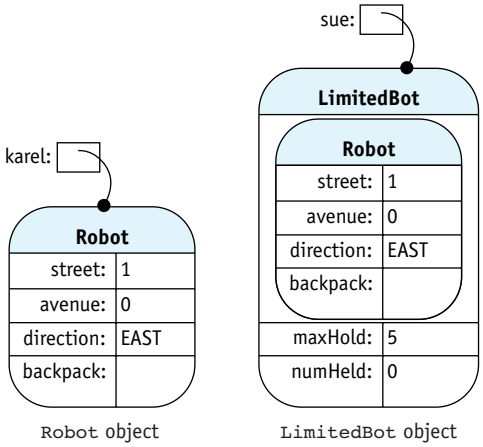
These two pieces of information will be stored as instance variables. Why use instance variables and not some other kind of variable? A temporary variable won't work because the robot needs to remember this information even when a method is not being executed. A parameter variable isn't what we need because we don't want to rely on the client to tell the robot how much it can carry every time a method is called.

In Chapter 1, we illustrated the attributes of a robot with an object diagram similar to the one shown on the left side of Figure 6-9. It represents a robot on the corner of (1, 0) facing east.

We can imagine an instance of `LimitedBot` as having a `Robot` object inside itself, along with the new instance variables it defines. This is illustrated on the right side of Figure 6-9. In this case, the robot is limited to holding five things at a time; it is currently holding none.

(figure 6-9)

Visualizing instance variables in a `Robot` *object and a* `LimitedBot` *object*



6.3.1 Declaring and Initializing the Variables

Listing 6-10 shows the beginning of our new kind of robot, `LimitedBot`. It includes the two new instance variables and the constructor, but nothing else. `LimitedBot` objects are identical to normal `Robot` objects except for the (currently unused) instance variables.

Listing 6-10: *A* `LimitedBot` *is like a normal Robot, but has two additional (yet to be used) instance variables*

```
1  import becker.robots.*;
2
3  /** A LimitedBot can carry or hold only a limited number of things. The
4  *  actual limit set when the robot is constructed.
5  *
6  *  @author Byron Weber Becker */
7  public class LimitedBot extends Robot
8  {
9     private int maxHold;          // Maximum # of things this robot can hold.
10    private int numHeld = 0;      // Number of things currently held by this robot.
11
12    /** Construct a new LimitedBot.
13    *  @param aCity          This robot's city.
14    *  @param aStr           This robot's initial street.
15    *  @param anAve          This robot's initial avenue.
16    *  @param aDir           This robot's initial direction.
17    *  @param maxCanHold     The maximum number of things this robot can carry/hold. */
18    public LimitedBot(City aCity, int aStr, int anAve,
19                      Direction aDir, int maxCanHold)
20    { super(aCity, aStr, anAve, aDir);
21      this.maxHold = maxCanHold;
22    }
23 }
```

The number of things held by the robot will always be zero when the robot is constructed, and so the `numHeld` instance variable is initialized to `0` when it is declared in line 10. The initial value of `maxHold`, however, isn't known when the class is written. It is initialized in the constructor with the value passed to the `maxCanHold` parameter, allowing its initial value to be determined when the `LimitedBot` is constructed.

Invoking `super` in line 20 calls a constructor in the superclass. Parameters such as `aStr` and `anAve` are passed as arguments to `super`, where they are likely used to initialize instance variables in the superclass.

This example illustrates two guidelines that are seldom broken:

- Every instance variable is initialized either where it is declared or in the constructor, with information passed via a parameter.
- Parameters to a constructor are used to initialize an instance variable in the same class via an assignment statement or an instance variable in a superclass via the call to `super`.

6.3.2 Maintaining and Using Instance Variables

Having the `maxHold` and `numHeld` instance variables is not enough. We need to maintain and use the information they store.

First, we need to monitor how many things are currently held by the robot, and call `breakRobot` if this number exceeds the number stored in `maxHold`. The number of things held by the robot changes whenever it picks a thing up or puts a thing down. Thus, we will need to override the definitions of `pickThing` and `putThing`.

Let's focus on `pickThing` first. In pseudocode, we want it to perform the following tasks:

```
if (already holding the maximum number of things)
{ break the robot
} else
{ pick up a thing
  increment the count of the number of things being held
}
```

The pseudocode for putting a thing down is similar except that there is no need to check if the maximum has been exceeded:

```
put down a thing
decrement the count of the number of things being held
```

These two methods are shown in lines 24–33 and 35–39 of Listing 6-11.

FIND THE CODE

ch06/limitedBot/

Listing 6-11: *Source code for a kind of robot that can pick up only a limited number of things*

```
1 import becker.robots.*;
2
3 /** A LimitedBot can carry or hold only a limited number of things. The
4 *  actual limit set when the robot is constructed.
5 *
6 *  @author Byron Weber Becker */
7 public class LimitedBot extends Robot
```

Listing 6-11: *Source code for a kind of robot that can pick up only a limited number of things* (continued)

```
 8  {
 9    private int maxHold;          // Maximum # of things this robot can hold.
10    private int numHeld = 0;      // Number of things currently held by this robot.
11
12    /** Construct a new LimitedBot.
13    *  @param aCity              This robot's city
14    *  @param aStr               This robot's initial street.
15    *  @param anAve              This robot's initial avenue.
16    *  @param aDir               This robot's initial direction.
17    *  @param maxCanHold         The maximum number of things this robot can carry/hold. */
18    public LimitedBot(City aCity, int aStr, int anAve,
19                      Direction aDir, int maxCanHold)
20    { super(aCity, aStr, anAve, aDir);
21      this.maxHold = maxCanHold;
22    }
23
24    /** Pick up a thing. If the robot is already holding the maximum number
25    *  of things, it breaks. */
26    public void pickThing()
27    { if (this.numHeld == this.maxHold)
28      { this.breakRobot("Tried to pick up too many things.");
29      } else
30      { super.pickThing();
31        this.numHeld = this.numHeld + 1;
32      }
33    }
34
35    /** Put down one thing. */
36    public void putThing()
37    { super.putThing();
38      this.numHeld = this.numHeld - 1;
39    }
40  }
```

In `pickThing`, we call `super.pickThing()` at line 30. This statement calls the unmodified version of `pickThing` provided by the `LimitedBot`'s superclass. The code surrounding this call details the additional steps that should be taken when a `LimitedBot`'s version of `pickThing` is called. `super.putThing()` is called at line 37 for similar reasons.

LOOKING BACK

Overriding methods was discussed in Section 2.6.1.

6.3.3 Blank Final Instance Variables

In Listing 6-11, `maxHold` is given a value when the object is initialized, but thereafter the value is unchanged. This suggests that `maxHold` is really a kind of constant even though we don't know its value until the object is instantiated.

The constructor may assign a value to a final instance variable as long as the variable hasn't been used already. This suggests that line 9 of Listing 6-11 should be rewritten as follows:

```
9  private final int MAX_HOLD;     // Maximum # of things this robot can hold.
```

Appropriate changes in the variable name should also be made in lines 21 and 27. A final variable that is not initialized until later is called a **blank final**. The compiler must be able to verify that a blank final is not used before it is assigned a value.

6.4 Modifying vs. Extending Classes

We now have two distinct approaches to modifying a class to do something new:

- Extending the class with additional methods and instance variables.
- Adding additional functionality within the class itself. In fact, the problem set for this chapter asks for many modifications to `SimpleBot`.

So, which is preferable: to extend a class with new functionality or modify the class itself?

As usual, the answer depends on the context. If the source code is not available (as is the case with `Robot`), you can't modify the class directly. The question becomes more interesting when the source code is available. The decision is usually made based on two criteria:

- How extensively has the class already been used, including subclasses? Modifying a class that is extensively used carries a significant risk of breaking code that already works. It also carries the burden of significant testing. In these cases, extending the class is usually the better idea.
- Are the proposed changes useful in many circumstances? If they are, modifying the class may be a good idea. However, if the changes are of limited use, the class becomes cluttered with features that are not typically used. Extending the class is often the wiser course in this situation as well.

These observations are represented in Table 6-2.

	Modifications are useful almost everywhere.	Modifications are useful in many settings.	Modifications are useful in only a few settings.
Class is already used extensively.	Modify the class.	Extend the class.	Extend the class.
Class is not used extensively.	Modify the class.	Modify the class.	Extend the class.

(table 6-2)

Factors in deciding whether to modify or extend a class

A third option is to create a new class that makes substantial use of an existing class to do its job. That's the topic of Chapter 8.

6.5 Comparing Kinds of Variables

We have examined three kinds of variables: instance variables, temporary variables, and parameter variables. How do you choose which kind of variable to use? This section compares and contrasts them, and provides some guidelines on selecting an appropriate kind of variable.

6.5.1 Similarities and Differences

Table 6-3 compares and contrasts the different kinds of variables.

	Instance Variables...	Temporary Variables...	Parameter Variables...
are declared...	inside a class but outside of the methods.	inside a method.	inside a method's parameter list.
are declared...	with an access modifier; beginning programmers should always use `private`.	without an access modifier.	without an access modifier.
use the `final` keyword when...	the value stored should not be changed.	the value stored should not be changed.	the value stored should not be changed.
are named (by convention)...	like methods: the first "word" is lowercase; subsequent "words" have an initial capital. If the `final` keyword is used, names should be all uppercase.	like methods: the first "word" is lowercase; subsequent "words" have an initial capital.	like methods: the first "word" is lowercase; subsequent "words" have an initial capital.
can be used...	in any method in the class.	in the smallest block enclosing the declaration.	in the method where they are declared.

(table 6-3)

Comparing the different kinds of variables

(table 6-3) *continued*

Comparing the different kinds of variables

	Instance Variables...	Temporary Variables...	Parameter Variables...
are initialized...	where they are declared or in the constructors.	where they are declared.	where the method is called.
store their value until...	it is changed or the object is no longer used.	it is changed or the smallest enclosing block has finished executing.	it is changed or the method has finished executing.
are referenced...	with the keyword `this`, a dot, and the variable's name; may be accessed with the class name when modifiers permit and they have the `static` keyword.	with only the variable's name.	with only the variable's name.

6.5.2 Rules of Thumb for Selecting a Variable

Table 6-4 can help you decide when each kind of variable is an appropriate choice based on your program's needs. The solutions are ordered from the most preferred to the least preferred; therefore, read the table from the top and use the first solution that meets your needs.

(table 6-4)

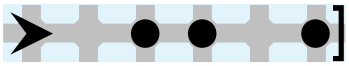
Rules of thumb for choosing which kind of variable to use

If you...	Then...
need a value that never changes while the program is running	use a `final` instance variable (constant). Valid exceptions are for the values 0, 1, and -1, unless the special value could be something else but just happens to be one of these.
need to store a value that will be used in a calculation later in the same method but then discarded	use a temporary variable.
have a method that could do things slightly differently based on a value known by the client	use a parameter.
find yourself writing almost identical code several times	look for a way to put the code in a method, accounting for the differences with parameters.
need a value in many methods within a class	consider using an instance variable.
need to implement an attribute of an object	use an instance variable or calculate the value based on existing instance variables.
have an object that must store a value even when none of its services are being used	use an instance variable.

6.5.3 Temporary versus Instance Variables

One of the hardest choices for many beginning programmers is deciding whether to use an instance variable or a temporary variable. This choice is difficult because nearly anything that can be done with a temporary variable can also be done with an instance variable. This situation often leads beginning programmers to overuse instance variables and underuse temporary variables.

Suppose that you need a query, `numIntersectionsWithThings`, that counts the number of intersections containing `Things` between the robot's current location and a wall that is somewhere in front of it. Invoking `numIntersectionsWithThings` on the robot shown in Figure 6-10 would move the robot to just before the wall and return the value 3.



(figure 6-10)

Initial situation for counting the number of things before a wall

We could solve this problem using an instance variable, as shown in Listing 6-12. This approach is not appropriate for an instance variable, however, because it stores temporary information, not an attribute of the robot.

Listing 6-12: *An inappropriate use of an instance variable*

```
1  import becker.robots.*;
2
3  public class CounterBot1 extends RobotSE
4  { private int intersections = 0;
5
6    public CounterBot1(City c, int str, int ave, Direction d)
7    { super(c, str, ave, d);
8    }
9
10   public int numIntersectionsWithThings()
11   {  while(true)
12      {  if (this.canPickThing())
13         {  this.intersections = this.intersections + 1;
14         }
15         if (!this.frontIsClear())   {  break; }
16         this.move();
17       }
18       return this.intersections;
19    }
20 }
```

FIND THE CODE

cho6/counter/

LOOKING AHEAD

This code can give an incorrect answer. See Written Exercise 6.3.

A better solution is to use a temporary variable. Rewriting the class in Listing 6-12 to use a temporary variable results in the class shown in Listing 6-13. The differences are shown in bold in both listings.

FIND THE CODE

cho6/counter/

Listing 6-13: *A robot using a temporary variable in* `numIntersectionsWithThings`

```
1  import becker.robots.*;
2
3  public class CounterBot2 extends RobotSE
4  {
5    public CounterBot2(City c, int str, int ave, Direction d)
6    { super(c, str, ave, d);
7    }
8
9    public int numIntersectionsWithThings()
10   { int intersections = 0;
11     while(true)
12     { if (this.canPickThing())
13       { intersections = intersections + 1;
14       }
15       if (!this.frontIsClear())   {  break; }
16       this.move();
17     }
18     return intersections;
19   }
20 }
```

Does it matter whether you choose an instance variable or a temporary variable? Yes, for the following reasons:

- Reading a program is easiest if variables are declared close to their use. Temporary variables keep declarations as close to their use as possible. That way the reader doesn't have to remember as many details for as long a time.
- The class as a whole is easier to understand if it isn't cluttered by extraneous instance variables. Readers assume that each instance variable has a meaning to the class as a whole and to several methods. If that's not true, it can take longer to understand the class.
- The longer lifetimes and larger scope of instance variables give programmers more opportunity to misuse them. Don't provide such opportunities unless you must.
- Extra instance variables increase the amount of memory required to run the program. For large programs, this can become an issue because it may limit the amount of data it can handle.

Temporary variables should be used when the value is not an attribute of the object and is primarily local to a method, or when storing a temporary value. Prime candidates include loop counters, a temporary variable to store an intermediate calculation, an accumulator such as `intersections` in Listing 6-13, or the temporary storage of the answer to a query before it's used in further calculations.

KEY IDEA

Use parameter and temporary variables when you can; instance variables only when you must.

For each instance variable, you should think carefully about whether it must be an instance variable. Is the data relevant to more than one public method? Does the data represent an attribute of the class? If so, make it an instance variable. If not, consider other options.

6.6 Printing Expressions

When debugging programs that use variables and expressions, it is often useful to print their values as the program is running. There are two approaches: inserting temporary code in the class to print the values out, and using a tool called a debugger.

6.6.1 Using `System.out`

`System.out` is an object that is automatically made available throughout every Java program. It has two methods, `print` and `println`, that are used to print values in the **console** window. The console is usually a separate window used specifically for default textual input and output. This is also where Java prints its error messages.

For example, the `pickThing` method in `LimitedBot` (see Listing 6-11) can be modified to print out useful debugging information by adding lines 2 and 3 in the following code:

```
 1 public void pickThing()
 2 { System.out.print("PickThing: numHeld=");          // debug
 3   System.out.println(this.numHeld);                 // debug
 4   if (this.numHeld == this.maxHold)
 5   { this.breakRobot("Tried to pick up too many things.");
 6   } else
 7   { super.pickThing();
 8     this.numHeld = this.numHeld + 1;
 9   }
10 }
```

The result of picking up three things using the modified class is shown in Figure 6-11. The black window in front of the usual robot window is the console.

(figure 6-11)

Information printed in the console window using `System.out`

The `print` and `println` methods are overloaded to take all of Java's types as arguments. In line 2, the `print` method is used to print the given string literal. In the next line, the `println` method is used to print the value stored in an integer variable.

Most programmers would combine lines 2 and 3 as follows:

```
System.out.println("PickThing: numHeld=" + this.numHeld);
```

When the plus operator (+) is used with a string, the result is a single string composed of the first operand textually followed by the second operand. The resulting string is then printed.

The difference between `print` and `println` is in where text will go the *next* time one of these methods is called. Using `print` causes subsequent text to be printed on the same line; using `println` causes subsequent text to be printed on the next line. The "ln" in `println` stands for "line."

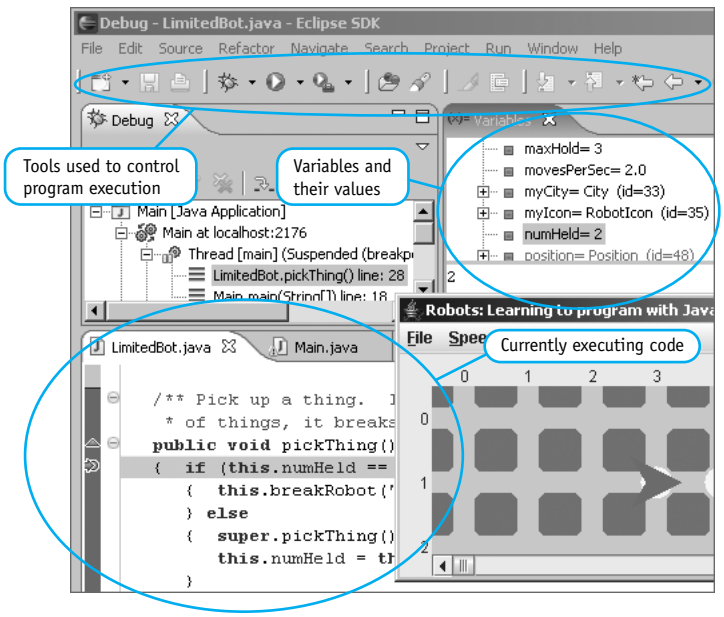
6.6.2 Using a Debugger

A **debugger** is a tool that can be used to view values while the program is running. It does not require you to add temporary code to your program. An example of one debugger is shown in Figure 6-12. It is part of the Eclipse project, a freely available integrated development environment. Three areas of the debugger are shown under the robot's window:

- The source code that is currently being executed is shown in the bottom left of the figure. It helps remind the programmer which variables are currently relevant. It is possible to set **breakpoints** before running the program. A breakpoint is associated with a program statement and causes the debugger to stop executing the program each time the statement is reached, giving the user an opportunity to examine the values held by variables.
- The variables that are currently in scope are shown in the upper-right corner of Figure 6-12. In this example, all the variables happen to be instance variables, but parameter and temporary variables can also appear in this area. The current value held by each variable is also shown. If the variable happens to refer to an object, a plus sign appears to the left, allowing its instance variables

to be shown as well. The debugger even shows private instance variables in `LimitedBot`'s superclasses.

- After a program stops at a breakpoint, the toolbar shown in the upper-left corner of Figure 6-12 is used to continue execution. For example, the arrow on the far left continues execution until the next breakpoint is reached. Some of the other tools allow the programmer to step to the next statement. One tool treats a method call as one statement to execute while another steps into a method to execute the next statement.



(figure 6-12)

Eclipse debugger in use

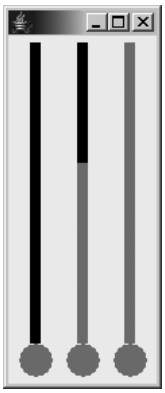
Debuggers are powerful tools that are worth learning. However, they are also complex and may distract beginning programmers from more important learning tasks.

6.7 GUI: Repainting

In this section, we'll create a new kind of graphical user interface component, a `Thermometer`. The major difference between a `Thermometer` and the `StickFigure` component created in Chapter 2 is that the `Thermometer` has an instance variable that controls its appearance. As Figure 6-13 shows, each thermometer can be set to show a different temperature.

(figure 6-13)

Three `Thermometer` *components, each with a different temperature setting*



The program in Listing 6-14 was used to create this image and may be used as a test harness during the development process. It creates three instances of the `Thermometer` class, displays them, and sets each to show a different temperature.

FIND THE CODE

ch06/thermometer/

Listing 6-14: *A test harness for the* `Thermometer` *class*

```
1  import javax.swing.*;
2
3  /** Test a thermometer component.
4  *
5  *  @author Byron Weber Becker */
6  public class Main extends Object
7  {
8    public static void main(String[] args)
9    { // Create three thermometer components.
10     Thermometer t0 = new Thermometer();
11     Thermometer t1 = new Thermometer();
12     Thermometer t2 = new Thermometer();
13
14     // Create a panel to hold the thermometers.
15     JPanel contents = new JPanel();
16     contents.add(t0);
17     contents.add(t1);
18     contents.add(t2);
19
20     // Set up the frame.
21     JFrame f = new JFrame();
22     f.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
23     f.setContentPane(contents);
24     f.pack();
25     f.setVisible(true);
26
```

Listing 6-14: *A test harness for the* `Thermometer` *class* (continued)

```
27        // Set the temperature of each thermometer.
28        t0.setTemperature(0);
29        t1.setTemperature(30);
30        t2.setTemperature(50);
31     }
32  }
```

6.7.1 Instance Variables in Components

In Section 2.7.3, we learned that the `paintComponent` method can be called by the Java system at any time. The user can resize a frame or expose a previously hidden frame. In either case, `paintComponent` will be called to repaint the contents of the frame. Therefore, the `paintComponent` method must be able to determine what the component should look like. For a `Thermometer`, this includes determining how high the alcohol (the modern replacement for mercury) should be drawn. It does so by consulting an instance variable. A client can set the instance variable to a given temperature with a small method called `setTemperature`.

Listing 6-15 shows the beginnings of the `Thermometer` class, complete with the instance variable used to store the current temperature. Most of the code in `paintComponent` must still be developed.

Listing 6-15: *The beginnings of the* `Thermometer` *class*

FIND THE CODE

cho6/thermometer/

```
1  import javax.swing.*;
2  import java.awt.*;
3
4  /** A thermometer component to use in graphical user interfaces. It can
5  *  display temperatures from MIN_TEMP to MAX_TEMP, inclusive.
6  *
7  *  @author Byron Weber Becker */
8  public class Thermometer extends JComponent
9  {
10    public final int MIN_TEMP = 0;
11    public final int MAX_TEMP = 50;
12    private int temp = MIN_TEMP;
13
```

Listing 6-15: *The beginnings of the* `Thermometer` *class* (continued)

```
14    /** Construct a new thermometer. */
15    public Thermometer()
16    { super();
17      this.setPreferredSize(new Dimension(50, 250));
18    }
19
20    /** Paint the thermometer to show the current temperature. */
21    public void paintComponent(Graphics g)
22    { super.paintComponent(g);
23
24      // paint the thermometer
25    }
26
27     /** Set the thermometer's temperature.
28     *  @param newTemp the new temperature. */
29     public void setTemperature(int newTemp)
30     { this.temp = newTemp;
31     }
32 }
```

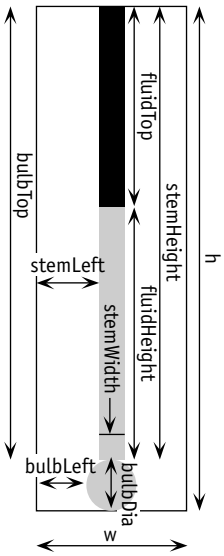
Recall that the preferred size, set in line 17, is used by the frame to determine how large the thermometer should be. Forgetting to set the preferred size will make the component so small that it is almost invisible.

LOOKING AHEAD

Programming Project 6.15 asks you to improve upon the hard-coded minimum and maximum.

This version of the class fixes the minimum and maximum temperature the thermometer can display with two named constants.

Working out the actual code for `paintComponent` is somewhat tedious. It helps to declare temporary variables initialized with significant values. The diagram in Figure 6-14 illustrates the meaning of those used in Listing 6-16.



(figure 6-14)

`Thermometer` *calculations*

The height and width of the component are found first in lines 5 and 6 and stored in variables to make using them more convenient. All the calculations should ultimately depend on the height and width so that the thermometer is drawn appropriately as the component is resized.

The variables with names ending in `Left` and `Top` hold values specifying the location of a shape. Variables with names ending in `Height`, `Width`, and `Dia` (short for "diameter") hold values specifying the size of a shape.

Listing 6-16: *The finished implementation of* `paintComponent`

```
1  /** Paint the thermometer to show the current temperature. */
2  public void paintComponent(Graphics g)
3  { super.paintComponent(g);
4
5    final int w = this.getWidth();
6    final int h = this.getHeight();
7
8    final int bulbDia = h/10;
9    final int bulbLeft = w/2 - bulbDia/2;
10   final int bulbTop = h - bulbDia;
11
12   final int stemWidth = bulbDia/3;
13   final int stemLeft = w/2 - stemWidth/2;
```

FIND THE CODE

cho6/thermometer/

Listing 6-16: *The finished implementation of* `paintComponent` (continued)

```
14    final int stemHeight = h - bulbDia;
15
16    final int fluidHeight = stemHeight *
17          (this.temp - MIN_TEMP) / (MAX_TEMP - MIN_TEMP);
18    final int fluidTop = stemHeight - fluidHeight;
19
20    // paint the fluid
21    g.setColor(Color.RED);
22    g.fillOval(bulbLeft, bulbTop, bulbDia, bulbDia);
23    g.fillRect(stemLeft, fluidTop, stemWidth, fluidHeight);
24
25    // paint the stem above the fluid
26    g.setColor(Color.BLACK);
27    g.fillRect(stemLeft, 0, stemWidth, fluidTop);
28 }
```

6.7.2 Triggering a Repaint

If you run the test harness with the current version of `Thermometer`, you will notice that the thermometers are painted as though the temperature is 0 rather than the temperatures set in the test harness. However, if you resize the frame, forcing the thermometers to be repainted, then they will be drawn with the correct temperatures. In other words, the thermometers display a temperature change only when they are repainted.

KEY IDEA

Call `repaint` *when instance variables affecting the image change.*

Somehow we need to be able to trigger the repainting of the component whenever the temperature changes. We do so with an inherited method, `repaint`. Calling `repaint` after we have reset the instance variable informs the Java system that it should call `paintComponent` as soon as possible. The revised version of `setTemperature` is:

```
public void setTemperature(int newTemp)
{ this.temp = newTemp;
  this.repaint();
}
```

6.7.3 Animating the `Thermometer`

Adding the following code to the end of the test harness will cause the thermometer to show a steadily increasing temperature—just like the temperature climbing on a hot summer's morning.

```
for(int temp = t0.MIN_TEMP; temp <= t0.MAX_TEMP; temp = temp
+ 1)
{  t0.setTemperature(temp);
   Utilities.sleep(50);
}
```

The call to `Utilities.sleep` causes the current thread to pause for 50 milliseconds, or 0.050 seconds, to give the Java system a chance to repaint the screen—and so you have time to see the change in the thermometer.

The `sleep` method should *not* be called inside the `paintComponent` method. `paintComponent` is called by the Java system; it has many important things to do and should not be forced to wait for anything.

6.8 Patterns

Every time you are writing an expression, you need values. These values could come from any of the constructs discussed in this chapter. In almost every situation, one of the constructs is a better choice than the others. Carefully consider which of the following patterns best describes your situation and is best suited to solve your problem.

6.8.1 The Named Constant Pattern

Name: Named Constant

Context: You have a literal value used one or more times in your program. The value is known when you write the program and does not change while the program is running.

Solution: Use a named constant, as suggested by the following examples:

```
private static final int DAYS_IN_WEEK = 7;
private static final int COST_PER_MOVE = 25;
```

In general, a named constant has the following form:

```
«accessModifier» static final «type» «name» = «value»;
```

where *«accessModifier»* is `public`, `protected`, or `private`. Use `private` if the value is used only within the class where it is defined. Use `public` if other classes might need it—for example, as an actual parameter to a method defined within the class.

«type» is the type of the value stored in the constant. So far, we have discussed only integers, but any type (including a class name) is possible. *«name»* is the name of the variable, and *«value»* is the first (and last) value assigned to it.

Graphics programs often use many constants in the course of drawing a picture. (See `paintComponent` in Section 2.7.3 for an example.) Having a named constant for each

can become tedious, and it is common practice to use literals instead. An excellent middle ground is to look for relationships between the numbers. It is often possible to define a few well-chosen constants that can be used in expressions to calculate the remaining values.

Consequences: Programs become more self-documenting when special values are given meaningful names. Reading, debugging, and maintaining a program become easier and faster when the program uses meaningful names.

Related Patterns:

- This pattern is a specialization of the Instance Variable pattern.
- When constants are used to distinguish a set of values, such as the four directions or `MALE` and `FEMALE`, the Enumeration pattern (see Section 7.7.3) is often a better choice.

6.8.2 The Instance Variable Pattern

Name: Instance Variable

Context: An object needs to maintain a value. The value must be remembered for longer than one method call (when a temporary variable would be appropriate). The value is usually needed in more than one method.

Solution: Use an instance variable. Instance variables are declared within the class but outside of all the methods. Following are examples of instance variables:

```
private int numMoves = 0;
private int currentAve;
```

An instance variable is declared with one of two general forms:

```
«accessModifier» «type» «name» = «initialValue»;
«accessModifier» «type» «name»;
```

where *«accessModifier»* is usually `private` and *«type»* is the type of the variable. Examples include `int`, `double`, `boolean`, and names of classes such as `Robot`. *«name»* is the name used to refer to the value stored. The variable's initial value should either be established in the declaration, as shown in the first form, or assigned in the constructor. Assign the initial value in the declaration if all instances of the class start with the same value. Assign it in the constructor if each instance will have its initial value supplied by parameters to a constructor.

An instance variable may be accessed within methods or constructors with the implicit parameter, `this`, followed by a dot and the name of the variable. It may also be accessed by giving the name of the variable if the name is not the same as a parameter or temporary variable.

An instance variable that is not explicitly initialized will be given a default value appropriate for its type, such as `0` for integer types and `false` for `boolean`.

Consequences: An instance variable stores a value for the lifetime of the object. It can be explicitly changed by an assignment statement.

Related Patterns:

- The Instance Variable pattern is inappropriate for storing values used within a single method for intermediate calculations, counting events, or loop indices. Use the Temporary Variable pattern instead.
- The Instance Variable pattern is inappropriate for communicating a value from client code to a method. Use the Parameterized Method pattern instead.
- The Instance Variable pattern always occurs within an instance of the Class pattern.

6.8.3 The Accessor Method Pattern

Name: Accessor Method

Context: You have a class with instance variables that are private to prevent misuse by clients. However, clients have a legitimate need to know the values of the instance variables even though they should not be allowed to directly change them.

Solution: Provide public accessor methods using the following template:

```
public «typeReturned» get«Name»()
{ return this.«instanceVariable»;
}
```

An example is an accessor for the `street` in the `SimpleBot` class, as follows:

```
public class SimpleBot
{ private int street;
  ...
  public int getStreet()
  { return this.street;
  }
}
```

Consequences: Restricted access is provided to an instance variable.

Related Patterns: The Accessor Method pattern is a specialization of the Query pattern.

6.9 Summary and Concept Map

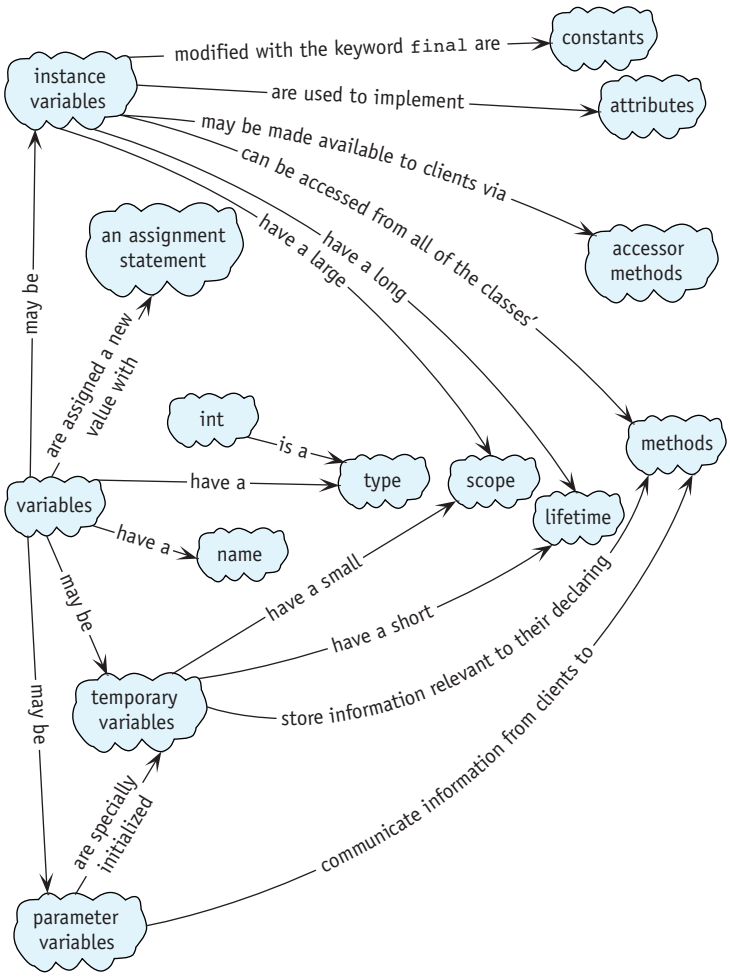
Variables are used to store information that will be useful at a later point in the program. The three fundamental kinds of variables are instance variables, temporary variables, and parameter variables. They differ in their scope, lifetime, and initialization.

Instance variables belong to an object. Each instance of a class has its own set of instance variables that implement that object's attributes. The lifetime is the same as the lifetime of the object. The scope is the entire class.

Temporary variables belong to the method or the block within a method in which they are declared, which also limits their scope. Of the three kinds of variables, temporary variables have the most limited scope. They are used for tasks such as storing intermediate calculations and counting events, such as loop iterations, within the method.

Parameter variables are temporary variables that are initialized when the method is called. Their scope is the entire method where they are declared, and their lifetime is for as long as the method executes. Both temporary and parameter variables disappear when the method in which they are declared finishes execution. If the method is executed again, space for the variable is reallocated and the variable is reinitialized.

Classes may be extended with additional instance variables, much as they can be extended with additional methods.



6.10 Problem Set

Written Exercises

6.1 What kind of variable does not occur in a class diagram?

6.2 The `Account` class models a bank account. Identify which type of variable (temporary, parameter, or instance) should be used in each of the following values. Justify your answers using Table 6-4.

a. The bank account's balance.

b. The amount to deposit in the account.

c. The account's current interest rate.

d. The amount of interest earned in the last month.

6.3 Listing 6-12 and Listing 6-13 contain code for `CounterBot1` and `CounterBot2`, both of which purport to count the number of intersections with things between the robot's current location and a wall. Consider executing the following `main` method in the initial situation shown in Figure 6-15. Execute it again, but using `CounterBot2` in line 3. The two solutions display different values for `side1` and `side2`.

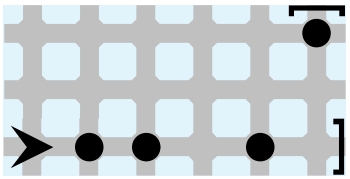
a. What are the four values printed (two for `CounterBot1` and two for `CounterBot2`)?

b. Explain why they differ.

```
1  public static void main(String[] args)
2  { City testCity = new City("testCity.txt");
3    CounterBot1 karel = new CounterBot1(...);
4
5    int side1 = karel.numIntersectionsWithThings();
6    karel.turnLeft();
7    int side2 = karel.numIntersectionsWithThings();
8
9    System.out.println("side1 =" + side1);
10   System.out.println("side2 =" + side2);
11 }
```

(figure 6-15)

Count the number of things



6.4 Draw an evaluation diagram for the expression assigned to `fluidHeight` in lines 16 and 17 of Listing 6-16. Assume the value of `stemHeight` is `225`, `this.temp` is `35`, `MIN_TEMP` is `-30`, and `MAX_TEMP` is `110`.

6.5 Section 6.2.1 noted that the remainder operator can be used to implement "wrap around" arithmetic and gave the example of `var = (var + 1) % upperLimit`.

a. Assume `upperLimit` has a value of `4`. Calculate the new value for `var` assuming that `var` is `0`, `1`, `2`, ..., `9`.

b. Implementing `turnLeft` with the following expression seems like it should work, but it doesn't. Explain why.

```
this.direction = (this.direction - 1) % 4
```

6.6 In Section 6.2.1 we saw the following code to place a `Thing` on every 5th intersection:

```
while (...)
{ if (this.getAvenue() % 5 == 0)
  { this.putThing();
  ...
```

What difference would it make if the `if` statement's Boolean expression was changed to `this.getAvenue() % 5 == 2`?

Programming Exercises

6.7 Finish the following code to sum, and print the individual digits stored in `digits`. For example, the sum, of the digits of `312` is `6` because 3 + 1 + 2 = 6. (*Hint*: Review the integer division and remainder operations and apply the four-step process to construct a loop.)

```
public static void main(String[] args)
{ int digits = 312;
  int sum ...

  System.out.println(sum);
}
```

6.8 Write a class named `FixedDistanceBot` that can only travel a specified number of intersections. The exact limit should be specified when the robot is constructed. If the limit is reached, the robot should break. Write a `main` method to test your class.

6.9 Extend the harvester robot from Section 3.2.7 to pick up all the things on each intersection (there may be 0, 1, or many), and count the total number of things it collects. Make the total available to the robot's client with a query. The robot is not guaranteed to start with an empty backpack.

6.10 Write a class named `DistanceBot` that extends `Robot`. It will have a query named `totalDistance` that returns the total distance traveled by the robot so far. A second query, `tripDistance`, returns the distance traveled since the "trip" was started by a call to `resetTrip`.

6.11 Create a component similar to the stick figure shown in Figure 2-15. Add methods named `setShirtColor` and `setPantsColor` that each take a single parameter of type `Color`. Invoking these methods should change the color of the corresponding article of clothing. (*Hint*: You will need variables of type `Color`; import `java.awt.Color`.)

6.12 Create a subclass of `JFrame` named `JClosableFrame`. Its constructor takes a `JPanel` as a parameter and does everything necessary to display it. Rewrite the `main` method in Listing 6-14 to test your class. (*Hint*: Your class will have a constructor but no methods of its own.)

6.13 Modify the `Thermometer` class as follows:

a. Allow different minimum and maximum temperatures for each instance of `Thermometer`. For example, a candy thermometer might show 100 to 400 degrees Fahrenheit, whereas a fever thermometer might show 37 to 42 degrees Celsius. Don't forget to test the class with negative numbers. The Fahrenheit or Celsius isn't relevant, only the numeric range.

b. Modify the `Thermometer` class so that it prints the minimum and maximum temperatures beside the fluid to form a scale. Also, print the current temperature beside the top of the fluid.

Programming Projects

Robot problems in this section use the simplified `Robot` classes. Get them from your instructor or download them from the Examples section of *www.learningwithrobots.com/software/downloads.html*.

6.14 Download the `SimpleBot` classes from the Robots Web site. Make the following enhancements to the `SimpleBot` class. In all cases, write a `main` method to test your work.

a. Complete the `SimpleBot` class as described in this chapter, including the `move`, `turnLeft`, and `paint` methods as well as the `SimpleBot` constructor.

b. Add a `turnRight` method.

c. Add a method named `goToOrigin`. The effect of calling `karel.goToOrigin()` is to have the robot named `karel` appear at the origin, facing east, the next time its `paint` method is called.

d. Add a method named `teleport`. The effect of calling `karel.teleport (5, 3)` is to have `karel` appear on the intersection of 5th Street and 3rd Avenue the next time `paint` is called. The direction it faces should not change. Of course, your method should work with values other than 5 and 3.

e. Implement a suite of three methods in the `SimpleBot` class that modify the robot's speed. `ben.goFaster()` causes the robot named `ben` to move 10% faster. `ben.goSlower()` causes `ben` to move 10% slower. Finally, `ben.setMoveTime(400)` causes `ben` to wait 400 milliseconds each time it moves. Also accommodate values other than 400.

f. Modify the `SimpleBot` class so that its color can be specified. This change will require a new instance variable, a change to the `paint` method, and a new method named `setColor` that takes a parameter variable of type `Color`.

g. Modify the `SimpleBot` class so that the size of each robot can be specified. `puffer.setSize(30)` causes the robot named `puffer` to have a body with a radius of 30 pixels. Other features, such as the sensor, should change size accordingly. Note that the size of the intersection should *not* change and that your method should work with many different values, not just 30.

h. Rewrite the `paint` method in `SimpleBot` so that robots have two "eyes" set on short antennae, as shown in Figure 6-16. Choose different colors for the eyes and the body.

(figure 6-16)

Robot with two eyes

i. A color can be created with three integers that specify the red, green, and blue components of the color. There is a constructor for the `Color` class that takes these three values as parameters. Each color component must be in the range of 0 to 255.

   Modify the `SimpleBot` class so that the robot will change color slightly every time it is painted. (*Hint*: Use the remainder operator (`%`).)

j. Modify the `SimpleBot` class so that the robot will move in four steps from one intersection to the next—that is, instead of moving instantly to the next intersection, move one step, wait a moment, move another step, wait a moment, step again, wait, and then complete the move and wait again.

   (*Hint*: This requires changes to both the `move` and the `paint` methods. One approach is to add a new instance variable that represents which step the robot is taking. This instance variable is set in `move` and used in `paint`.)

6.15 Write a class named `HomingBot`. A `HomingBot`'s home is the intersection where it is constructed. Add a method named `goHome` that moves the robot to its home facing east. Assume there are no obstacles. Write a `main` method to test your class.

6.16 Write a class named `FuelBot`. A `FuelBot` has a "fuel tank" that can hold "fuel." The maximum number of units of fuel it can hold is specified when the robot is created. Each move consumes one unit of fuel. If there is no fuel, the robot won't move. Each time the robot encounters an intersection with a `Thing` on it, the fuel tank is refilled.

Extend the `RobotRC` (Remote Controlled) class and read the documentation to learn to direct the robot's actions from the keyboard. Set up a game to see if you can choose a path to move between two points—with appropriate refueling stops—without running out of fuel.

6.17 Write a new class named `RobotME` (My Edition) that extends `Robot` and includes a method named `clearArea`. This method takes four parameters. The first two are an avenue and street that specify the upper-left corner of a rectangular area. The third and fourth specify the width and height of the area. Calling `clearArea` causes the robot to pick up everything in the given rectangular area and then move to the area's upper-left corner and face east.

The robot may start anywhere in the city. Once it has reached the area, it should not leave it.

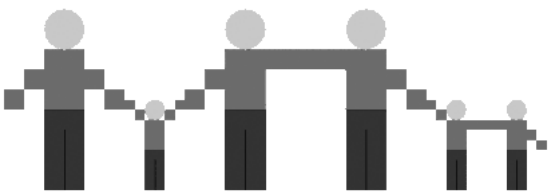
6.18 Create a component similar to the stick figure shown in Section 4.7.

a. Modify the stick figure component so that a stick figure may be constructed as either a child or an adult. An adult's preferred size is 180 by 270 pixels. A child has a preferred size that is half as large. Modify the test harness shown in Listing 6-14 to show two child stick figures and 1 adult.

(*Hints*: First, each stick figure will be similar to the `Thermometer` class. Use a test harness similar to Listing 6-14 to test your class. Second, define two constants, `CHILD` and `ADULT`. Pass one of them as a parameter to the stick figure's constructor. Third, assuming the `JPanel` containing the stick figures is named `contents`, include the statement `contents.setLayout (new RowLayout())` in your `main` method; it will align the stick figures appropriately. You will need to import `becker.gui.RowLayout`.)

b. Modify the stick figure constructor so that it takes three parameters. One, as in Part a, specifies whether the stick figure is an adult or a child. The other two parameters specify whether the left and right arms should be up, down, or straight out. Modify the test harness to construct six stick figures that are holding hands, as shown in Figure 6-17.

c. Modify the stick figure from Part b to add methods allowing the client to specify while the program is running whether an arm is up, down, or straight out. Modify the test harness to make the stick figures at each end of the line wave their free arm.



(figure 6-17)

Stick figures holding hands