Chapter 7 More on Variables and Methods

After studying this chapter, you should be able to:

- Write queries to reflect the state of an object
- Use queries to write a test harness that tests your class
- Write classes that use types other than integer, including floating-point types, `boolean`s, characters, and strings
- Write and use an enumerated type
- Write a class modeling a simple problem
- Describe the difference between a class variable and an instance variable
- Write classes that implement an interface and can be used with provided graphical user interfaces from the `becker` library

We now have the intellectual tools to start writing object-oriented programs that have nothing to do with robots. In this chapter, we'll learn about additional kinds of information we can store (such as dollar values, individual characters, or strings of characters). We'll use that knowledge to build a class that could be used as part of a gas pump at your local gas station.

One problem, however, is that such programs are not nearly as easy to debug as robot programs because they are not as visual. We'll start by learning some techniques for testing and debugging our programs and finish by learning techniques for coupling a class with a graphical user interface.

7.1 Using Queries to Test Classes

Writing a class that functions correctly is difficult; many things can go wrong. Having a set of tests that demonstrates that a class is functioning correctly makes the job easier. Tests are also useful to students before handing in an assignment and to customers before they buy software. We'll begin by learning how to test the `SimpleBot` class used in Chapter 6. Later in this chapter, we'll apply these same techniques to a non-robot class.

KEY IDEA

A good set of tests makes a programmer's life much easier.

7.1.1 Testing a Command

It is tempting to test the `SimpleBot` class by writing and running a short program that creates a robot and moves it several times, and then looking at the screen to verify that the robot did, indeed, move correctly. The problem with this approach is that a person must remember what should happen and verify that it actually did happen. Relying on people for such tedious details is a risky proposition.

Remembering and verifying tedious details is something that computers do well, however. Our goal is to completely automate as much of the testing as possible by writing a program called a **test harness**. A test harness is used to test a class, and usually contains many individual tests.

KEY IDEA

Testing involves many tedious details—something computers are good at. Use them as much as possible in the testing process.

Writing a test involves five steps.

1. Decide which method you want to test.
2. Set up a known situation.
3. Determine the expected result of executing the method.
4. Execute the method.
5. Verify the results.

For example, we may want to test the `move` method in the `SimpleBot` class (Step 1). To set up a known situation (Step 2), we create a robot named `karel` at (4, 2) facing east in an empty city. This is shown in lines 7 and 8 of Listing 7-1. The choice of (4, 2) facing east is not critical. We could just as easily use a different intersection. However, we need to know which intersection is chosen so we can determine the expected result (Step 3). In this case, moving from (4, 2) should result in the robot being on intersection (4, 3), still facing east.

Line 11 in Listing 7-1 executes the code we want to test (Step 4).

Finally, we verify the results (Step 5) in lines 14–18. Before explaining these lines, let's take a look at the result of running the program, as shown in Figure 7-1, and note the following:

- This program prints results of the tests in the console window.
- One line is printed for each invocation of `ckEquals` in lines 15–18. It prints "`Passed`" if the last two arguments have equal values. If they do not, it prints "`***Failed`". In either case, `ckEquals` also prints the values of both arguments.
- The `ckEquals` method also prints the string given as the first argument. This serves simply to identify the test.

Listing 7-1: *A program to test the* `SimpleBot`*'s* `move` *method*

```
1  import becker.util.Test;
2
3  public class TestHarness
4  {
5    public static void main(String[] args)
6    { // Set up a known situation (an empty city; a robot on (4, 2) facing east).
7      SimpleCity c = new SimpleCity();
8      SimpleBot karel = new SimpleBot(c, 4, 2, Constants.EAST);
9
10     // Execute the move method.
11     karel.move();
12
13     // Verify the results -- robot on intersection (4, 3).
14     Test tester = new Test();      // This line isn't needed. See Section 7.5.2.
15     tester.ckEquals("new ave", 3, karel.getAvenue());
16     tester.ckEquals("same str", 4, karel.getStreet());
17     tester.ckEquals("same dir", Constants.EAST,
18                                 karel.getDirection());
19   }
20 }
```

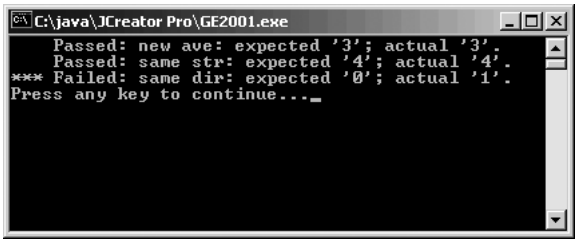
PATTERN

Test Harness

KEY IDEA

Testing a method usually requires repeating Steps 2–5 several times.

We should not be under the illusion that Listing 7-1 is sufficient to test the `move` method. At a minimum, it should test moving in each of the four directions. If programs using `SimpleBots` can include walls or similar obstructions, more tests are required to verify that `move` behaves correctly when a robot is blocked. This observation implies that Steps 2–5 for testing a method should be repeated as many times as necessary.



(figure 7-1)

Running the test in Listing 7-1, with a deliberate bug

What does `ckEquals` do? It compares the expected value (the second argument) with the actual value (the third argument) and prints an appropriate message. It is implemented approximately as shown in Listing 7-2. Overloaded versions for non-integer types have a few minor variations.

KEY IDEA

`ckEquals` *compares the expected value with the actual value and prints an appropriate message.*

Listing 7-2: *A possible implementation of the* `ckEquals` *method for integers*

```
1  public void ckEquals(String msg, int expected, int actual)
2  { String result;
3    if (expected == actual)
4    { result = " Passed: " + msg;
5    } else
6    { result = "*** Failed: " + msg;
7    }
8    result += ": expected '" + expected + "'; actual '" + actual + "'.";
9    System.out.println(result);
10 }
```

7.1.2 Testing a Query

Testing a query is actually easier than testing a command. To test a command, we need some way to verify what the command did. In the previous example, we used accessor methods to get the current values of the critical instance variables. To test a query, we only need to compare the query's actual result with the expected result.

To further illustrate testing, let's define a new `SimpleBot` query that answers the question "How far is this robot from the origin?" Remember that the origin is the intersection (0, 0). Let's assume that the distance we want is "as the robot moves" (the legs of a triangle) rather than "as the crow flies" (the hypotenuse of a triangle). If the robot is on Street 4, Avenue 2, the answer is 4 + 2 = 6.

A first attempt at our query is as follows:

```
public int distanceToOrigin()                    // Contains a bug.
{ return this.street + this.avenue;
}
```

To begin writing a test harness, we can perform the five steps mentioned previously. The code to test (Step 1) is `distanceToOrigin`. Our first known situation (Step 2) will be to create a robot at the origin facing east (testing easy cases first is a good strategy). In this situation, the distance to the origin should be 0 (Step 3). Executing the code (Step 4) and verifying the result (Step 5) is shown in the following code in lines 7 and 10, respectively:

Test Harness

```
1  public static void main(String[] args)
2  { // Create a robot in an empty city at the origin facing east.
3    SimpleCity c = new SimpleCity();
4    SimpleBot k = new SimpleBot(c, 0, 0, Constants.EAST);
5
6    // Execute the code to test.
7    int d = k.distanceToOrigin();
8
9    // Verify the result.
10   Test tester = new Test();          // This line isn't needed. See Section 7.5.2.
11   tester.ckEquals("at origin", 0, d);
12 }
```

This is a very incomplete test, however. The `distanceToOrigin` query could be written as follows and still pass this test:

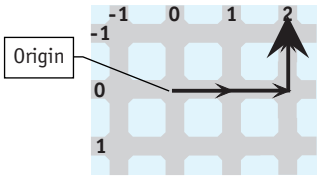
```
public int distanceToOrigin()
{ return 0;
}
```

We can add more tests to this test harness that build from the original known situation. For example, it's not hard to see that after the previous test the robot should still be at the origin. So let's add another test immediately after it that moves the robot from the origin and then checks the distance again.

```
1  public static void main(String[] args)
2  { // Create a robot in an empty city at the origin facing east.
3    SimpleCity c = new SimpleCity();
4    SimpleBot k = new SimpleBot(c, 0, 0, 0);
5
6    // Execute the code to test.
7    int d = k.distanceToOrigin();
8
9    // Verify the result.
```

```
10    Test tester = new Test();          // This line isn't needed. See Section 7.5.2.
11    tester.ckEquals("at origin", 0, d);
12
13    // Move east 2 intersections and verify.
14    k.move();
15    k.move();
16    d = k.distanceToOrigin();
17    tester.ckEquals("east 2", 2, d);
18  }
```

So far we have only tested the robot on streets and avenues that are numbered zero or larger. What if the robot turned left (facing north) and moved to Street -1, as shown in Figure 7-2? Let's test it to make sure `distanceToOrigin` works correctly.



(figure 7-2)

Robot at (-1, 2), three moves from the origin

We could add the new test to our test harness by continuing to move the robot to (-1, 2). The following code uses a simpler approach. It constructs a robot on intersection (-1, 2) and then tests the result of the `distanceToOrigin` method. In this case, moving the robot isn't necessary. This code should be added after line 17 of the test harness.

```
SimpleBot k2 = new SimpleBot(c, -1, 2, 0);
d = k2.distanceToOrigin();
tester.ckEquals("neg str", 3, d);
```

Running the test harness says that the test fails. The expected value is 3, but the actual value is 1.

Reviewing the `distanceToOrigin` method shows why: we add the current street to the current avenue. When both are positive values, that works fine. But in this situation, it gives -1 + 2, or 1—a wrong answer.

The problem is that we want to add the *distance* between the origin and the robot's street. Distances are always positive. When the street (or avenue) is negative, we need to convert it to a positive number. We can do this with the helper method `abs`, short for "absolute value."

The following implementation of `distanceToOrigin` will fix this problem.

```
 1 public int distanceToOrigin()
 2 { return this.abs(this.street) + this.abs(this.avenue);
 3 }
 4
 5 private int abs(int x)
 6 { int answer = x;
 7   if (x < 0)
 8   { answer = -x;
 9   }
10   return answer;
11 }
```

LOOKING AHEAD

In Section 7.5.2 we will learn about using a library of math functions. It already has `abs`.

With this change, all of the tests shown earlier will pass.

7.1.3 Using Multiple Main Methods

One fact that is implicit in the previous discussion is that Java allows multiple `main` methods. You can have only one `main` method in any given class, but as many classes as you want may each have their own `main` method. This is a good thing. If only one `main` method were allowed, we would need to choose between writing a test harness and writing a `main` method to run the program to perform its task.

KEY IDEA

Each class can have its own `main` *method.*

One common way to exploit the ability for each class to have a `main` method is to write one class that has nothing but `main`—the way we have been doing. This class is used to run the program to perform the desired task. However, every *other* class also has a `main` method to act as a test harness for that class. For example, the test harness shown in Listing 7-1 is in its own class. Instead, this could be written as part of the `SimpleBot` class. An outline of this approach is shown in Listing 7-3. Lines 1–14 show representative parts of the `SimpleBot` class. The test harness is in lines 16–28.

Test Harness

Listing 7-3: *An outline of how to include a test harness in the* `SimpleBot` *class*

```
 1 import java.awt.*;
 2 import becker.util.Test;
 3 ...
 4
 5 public class SimpleBot extends Paintable
 6 {
 7   private int street;
 8   private int avenue;
 9   private int direction;
10   ...
11   public SimpleBot(...) { ... }
```

Listing 7-3: *An outline of how to include a test harness in the* `SimpleBot` *class* (continued)

```
12    public void move() { ... }
13    ...
14
15    // A test harness to test a SimpleBot.
16    public static void main(String[] args)
17    { // Set up a known situation -- a robot on intersection (4, 2)
18      SimpleCity c = new SimpleCity();
19      SimpleBot karel = new SimpleBot(c, 4, 2, EAST);
20
21      // Execute the code we want to test.
22      karel.move();
23
24      // Verify the results -- robot on intersection (4, 3).
25      Test tester = new Test();     // This line isn't needed. See Section 7.5.2.
26      tester.ckEquals("new ave", 3, karel.getAvenue());
27      ...
28    }
29  }
```

One issue that may be initially confusing is that even though `main` is within the `SimpleBot` class, we don't use the keyword `this`. Inside the test harness, we construct a specific `SimpleBot` object, `karel`. Throughout the `main` method, we invoke `karel`'s methods to test what has happened to that specific object.

KEY IDEA

The `main` *method can't use* `this`.

One advantage of placing a `main` method inside the class it tests is that we have access to the classes' private instance variables. For example, line 26 of Listing 7-3 can be replaced with the following:

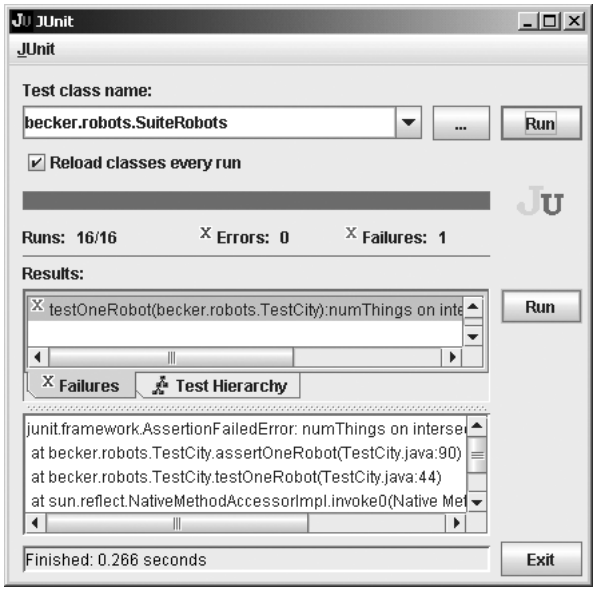
```
tester.ckEquals("new ave", 3, karel.avenue);
```

We should use an accessor method such as `getAvenue` when it is available. However, we can access the instance variables directly when their values are needed for testing but should not be provided to others via an accessor method.

Many programmers take testing even further with a tool named JUnit. It provides a graphical user interface, shown in Figure 7-3, and does a better job of isolating individual tests from each other. More information, and the tool itself, is available at *www.junit.org*.

(figure 7-3)

A popular testing tool named `JUnit`



7.2 Using Numeric Types

KEY IDEA

Java's primitive types store values such as `159` *and* `'d'`.

Not everything in Java is an object like a `Robot` or a `Thing`. Integers and the type `int` are the most prominent examples we've seen of a **primitive type**. Primitive types store values such as integers (`159`) and characters (`'d'`), and correspond to how information is represented in the computer's hardware. Primitive types can't be extended and they don't have methods that can be called. In this sense, primitive types distort the design of the language. However, the designers of Java felt it necessary to use primitive types for integers and similar values to increase the execution speed of programs.

Java includes eight primitive types. Six of these store numbers, one stores the Boolean values `true` and `false`, and the last one stores characters.

7.2.1 Integer Types

KEY IDEA

An `int` *can only store values in a certain range.*

Why would Java have six different types to store numbers? Because they differ in the size and precision of the values they store. An `int`, for example, can only store values between `-2,147,483,648` and `2,147,483,647`. This range is large enough to store the net worth of most individuals, but not that of Bill Gates. It's more than enough to store the population of any city on earth, but not the population of the earth as a whole.

To address these issues, Java offers several kinds of integers, each with a different **range**, or number of different values it can store. The ranges of the four integer types are shown in Table 7-1. Variables with a greater range require more memory to store. For programs with many small numbers to store, it makes sense to use a type with a smaller range. Because beginning programmers rarely encounter such programs, we won't need to use `byte` and `short` in this book and will use `long` only rarely.

KEY IDEA

Different types can store different ranges of values.

Type	Smallest Value	Largest Value	Precision
`byte`	`-128`	`127`	exact
`short`	`-32,768`	`32,767`	exact
`int`	`-2,147,483,648`	`2,147,483,647`	exact
`long`	`-9,223,372,036,854,775,808`	`9,223,372,036,854,775,807`	exact

(table 7-1)

Integer types and their ranges

7.2.2 Floating-Point Types

Two other primitive types, `float` and `double`, store numbers with decimal places, such as 125.25, 3.14259, or -134.0. They are called **floating-point** types because of the way they are stored in the computer hardware.

Floating-point types can be so large or small that they are sometimes written in **scientific notation**. The number 6.022E23 has two parts, the **mantissa** (6.022) and the **exponent** (23). To convert 6.022E23 to a normal number, write down the mantissa and then add enough zeros to slide the decimal point 23 places to the right. If the exponent is negative, you must add enough zeros to slide the decimal point that many places to the left. For example, 6.022E23 is the same number as 602,200,000,000,000,000,000,000, while 5.89E-4 is the same as 0.000589. Their ranges and precisions are listed in Table 7-2.

KEY IDEA

Scientific notation can be used to express very large or very small numbers.

Type	Smallest Magnitude	Largest Magnitude	Precision
`float`	`±1.40239846E-45`	`±3.40282347E+38`	About 7 significant digits
`double`	`±4.94065645841246544E-324`	`±1.79769313486231570E+308`	About 16 significant digits

(table 7-2)

The ranges and precisions of the various floating-point types

How big are these numbers? Scientists believe the diameter of the universe is about 1.0E28 centimeters, or 1.0E61 plank units—the smallest unit we can measure. The universe contains approximately 1.0E80 elementary particles such as quarks, the component parts of atoms. So the range of type `double` will certainly be sufficient for most applications.

Floating-point numbers don't behave exactly like real numbers. Consider, for a moment, 1/3 written in decimal: 0.33333.... No matter how many threes you add, 0.33333 won't be exactly equal to 1/3. The situation is similar with 1/10 in binary, the number system computers use. It's impossible to represent 1/10 exactly; the best we can do is to approximate it. The closeness of the approximation is given by the **precision**. `float`s have about 7 digits of precision, while `double`s have about 16 digits. This means, for example, that a `float` can't distinguish between `1.00000001` and `1.00000002`. As far as a `float` is concerned, both numbers are indistinguishable from `1.0`. Another effect is that assigning `0.1` to a `float` and then adding that number to itself 10 times does *not* yield `1.0` but `1.0000001`.

KEY IDEA

Comparing floating-point numbers for equality is usually a bad idea.

The fact that floating-point numbers are only approximations can cause programmers headaches if their programs require a high degree of precision. For beginning programmers, however, this is rarely a concern. One exception, however, is when comparing a `float` or a `double` for equality, the approximate nature of these types may cause an error. For example, the following code fragment appears to print a table of numbers between `0.0` and `10.0`, increasing by `0.1`, along with the squares of those numbers.

```
double d = 0.0;

while (d != 10.0)
{ System.out.println(d + " " + d*d);
  d = d + 0.1;
}
```

The first few lines of the table would be:

```
0.0 0.0
0.1 0.010000000000000002
0.2 0.04000000000000001
0.30000000000000004 0.09000000000000002
0.4 0.16000000000000003
```

Already we can see the problem: `d`, the first number on each line, is not increasing by exactly `0.1` each time as expected. In the fourth line the number printed is only approximately `0.3`.

By the time `d` gets close to `10.0`, the errors have built up. The result is that `d` skips from `9.99999999999998` to `10.09999999999998` and is never exactly equal to `10.0`, as our stopping condition requires. Consequently, the loop keeps printing for a very long time.

The correct way to code this loop is to use an inequality, as in the following code:

```
while (d <= 10.0)
{ ...
}
```

7.2.3 Converting Between Numeric Types

Sometimes we need to convert between different numeric types. In many situations, information is not lost and Java makes the conversion automatically. For example, consider the following statement:

```
double d = 159;
```

Java will implicitly convert the integer `159` to a double value (`159.0`) and then assign it to `d`.

The reverse is not true. If assigning a value to another type risks losing information, a **cast** is required. A cast is our assurance to the compiler that we either know from the nature of the problem that information will not be lost, or know that information will be lost and accept or even prefer that result.

KEY IDEA

Casting converts values from one type to another. Sometimes it loses information.

For example, consider the following statements:

```
double d = 3.999;
int i = d;
```

Java will display an error message regarding the second assignment because an integer can't store the decimal part of `3.999`, only the `3`. If we want to perform this assignment anyway and lose the `.999`, leaving only `3` in the variable `i`, we need to write it as follows:

LOOKING AHEAD

Section 7.5.2 discusses a method to round a number rather than truncate it.

```
double d = 3.999;
int i = (int)d;
```

The new part, `(int)`, is the cast. The form of a cast is the destination type placed in parentheses. It can also apply to an entire expression, as in the following statement:

```
int i = (int)(d * d / 2.5);
```

Casting has a high precedence, so you will usually need to use parentheses around expressions.

Assigning from a `double` to an `int` is not the only place information can be lost and a cast required. Information can also be lost assigning values from a `double` to a `float` or from a bigger integer type such as `long` to a smaller type such as `int`.

7.2.4 Formatting Numbers

Java automatically converts a primitive type to a string before concatenating it to another string. This capability allows us to easily print out a mixture of strings and numbers, such as `System.out.println("Age = " + age);` where `age` is an integer.

Automatic conversion to a string does not work as well for `double` values, where we often want to control how many significant digits are printed. For example, the following code might be used in calculating the price of a used car:

```
double carPrice = 12225.00;
double taxRate = 0.15;

System.out.println("Car: " + carPrice);
System.out.println("Tax: " + carPrice * taxRate);
System.out.println("Total: " + carPrice * (1.0 + taxRate));
```

This code gives the following output:

```
Car: 12225.0
Tax: 1833.75
Total: 14058.749999999998
```

These results are far from ideal. We want to see a currency symbol such as $ or £ printed. All of the amounts should have exactly two decimal places, rounding as necessary. The thousands should also be grouped with commas or spaces, depending on local conventions. It's difficult to implement all these details correctly.

Using a `NumberFormat` Object

Fortunately, Java provides a set of classes for formatting numbers, including currencies. These classes all include a method named `format` that takes a number as an argument and returns a string formatted appropriately. Listing 7-4 shows how to use a currency formatting object named `money`. These statements produce formatted output such as the following:

```
Car: $12,225.00
Tax: $1,833.75
Total: $14,058.75
```

Listing 7-4: *Using a currency formatting object*

```
1 double carPrice = 12225.00;
2 double taxRate = 0.15;
3
4 System.out.println("Car: " + money.format(carPrice));
5 System.out.println("Tax: " +
6                    money.format(carPrice * taxRate));
7 System.out.println("Total: " +
8                    money.format(carPrice * (1.0 + taxRate)));
```

FIND THE CODE

ch07/formatNumbers/

A formatting object is not normally obtained by using a constructor. Instead, a **factory method** in the `NumberFormat` class is called. A factory method returns an object reference, as a constructor does. Unlike a constructor, a factory method has the option of returning a subclass of `NumberFormat` that is specialized for a specific task. In this case, the factory method tries to determine the country where the computer is located and returns an object customized for the local currency.

LOOKING AHEAD

Implementing factory methods will be discussed in Chapter 12.

The `NumberFormat` class contains the `getCurrencyInstance`, `getNumberInstance`, and `getPercentInstance` factory methods, along with several others. The `getCurrencyInstance` factory method can be used by importing `java.text.NumberFormat` and including the following statement before line 4 in Listing 7-4.

KEY IDEA

Factory methods help you obtain an object already set up for a specific situation.

```
NumberFormat money = NumberFormat.getCurrencyInstance();
```

A formatter for general numbers can be obtained with the `getNumberInstance` factory method. It can be customized to format numbers with a certain number of decimal places and to print grouping characters. Consider the following example:

```
NumberFormat f = NumberFormat.getNumberInstance();
f.setMaximumFractionDigits(4);
f.setGroupingUsed(true);
System.out.println(f.format(3141.59265359));
```

These statements will print the value `3,141.5927`—the value rounded to four decimal places with an appropriate character (in this case, a comma) used to group the digits.

Columnar Output

Programs often produce lots of numbers that are most naturally formatted in columns. Even with the program to calculate the tax for a car purchase, aligning the labels and numbers vertically makes the information easier to read.

One of the easiest approaches uses the `printf` method in the `System.out` object. It was added in Java 1.5, and is not available in earlier versions of Java.

KEY IDEA

`printf`'s format string says how to format the other arguments.

The `printf` method is unusual in that it takes a variable number of arguments. It always takes at least one, called the **format string**, that includes embedded codes describing how the other arguments should be printed.

Here's an example where `printf` has three arguments.

FIND THE CODE

ch07/formatNumbers/

```
System.out.printf("%-10s%10s", "Car:", money.format(carPrice));
```

The first argument is the format string. It includes two **format specifiers**, each one beginning with a percent (`%`) sign and ending with a character indicating what kind of data to print. The first format specifier is for the second argument; the second specifier is for the third argument. Additional specifiers and arguments could easily be added.

In each case, the `s` indicates that the argument to print should be a string. The `10` instructs `printf` to print the string in a field that is 10 characters wide. The minus sign (`-`) in one says to print that string **left justified** (starting on the left side of the column). The specifier without the minus sign will print the string **right justified** (on the right side of the column).

This line, as specified, does not print a newline character at the end; thus, any subsequent output would be on the same line. We could call `println()` to end the line, or we could add another format specifier. The specifier `%n` is often added to the format string to begin a new line. It does *not* correspond to one of the arguments.

Table 7-3 gives several examples of the most common format specifiers and the results they produce. A `d` is used to print a decimal number, such as an `int`. An `f` is used to print a floating-point number, such as a `double`. In addition to the total field width, it specifies how many decimal places to print. More examples and a complete description are available in the online documentation for the `java.util.Formatter` class.

(table 7-3)

Examples of common format specifiers; dots signify spaces

Format Specifier and Argument	Result
`"%-10s", "Car:"`	`Car:·····`
`"%10s", "Car:"`	`·····Car:`
`"%10d", 314`	`·······314`
`"%10.4f", 3.1415926`	`3.1416····`
`"%-10.4f", 3.1415926`	`····3.1416`

The `printf` method has many other options that are documented in the `Formatter` class. Discussing them further, however, is beyond the scope of this book.

7.2.5 Taking Advantage of Shortcuts

KEY IDEA

`i++` is a shortcut for `i = i + 1`.

Java includes a number of shortcuts for some of the most common operations performed with numeric types. For example, one of the most common is to add 1 to a variable. Rather than writing `ave = ave + 1`, Java permits the shortcut of writing `ave++`. A similar shortcut is writing `ave--` in place of `ave = ave - 1`.

It is also common to add the result of an expression to a variable. For example, the following is `SimpleBot`'s `move` method as written in Listing 6-6:

```
public void move()
{ this.street = this.street + this.strOffset();
  this.avenue = this.avenue + this.aveOffset();
  Utilities.sleep(400);
}
```

Instead of repeating the variable on the right side of the equal sign, we can use the `+=` operator, which means to add the right side to the value of the variable on the left, and then store the result in the variable on the left. More precisely, *`«var»` `+=` `«expression»`* means *`«var»` `=` `«var»` `+` `(«expression»)`*. The parentheses are important in determining what happens if *`«expression»`* contains more than a single value. The following example is equivalent to the previous code:

```
public void move()
{ this.street += this.strOffset();
  this.avenue += this.aveOffset();
  Utilities.sleep(400);
}
```

There are also `-=`, `*=`, and `/=` operators. They are used much less frequently but behave the same as `+=` except for the change in numeric operation.

7.3 Using Non-Numeric Types

Variables can also store information that is not numeric, using the types `boolean`, `char`, and `String`.

7.3.1 The `boolean` Type

The `boolean` type is used for `true` and `false` values. We have already seen Boolean expressions used to control `if` and `while` statements, and as a temporary variable and the return type in predicates (see, for example, Listing 5-3). We have also explored using `boolean` values in expressions (see Section 5.4).

Instance variables, named constants, and parameter variables can also be of type `boolean`. For example, a Boolean instance variable can store information about whether a robot is broken. The robot might consult that variable each time it is asked to move, and only move if it has not been previously broken.

LOOKING BACK

A Boolean temporary variable is used in `rightIsBlocked`, *section 5.2.5.*

```
public class SimpleBot extends Paintable
{ private int avenue;
  private int street;
  private boolean isBroken = false;
  ...

  public void breakRobot()
  { this.isBroken = true;
  }

  public void move()
  { if (!this.isBroken)
    { this.avenue = ...
      this.street = ...
    }
  }
  ...
}
```

7.3.2 The Character Type

A single character such as `a`, `Z`, `?`, or `5` can be stored in a variable of type `char`. These include the characters you type at the keyboard—and many more that you can't type directly. Like the other primitive types, the `char` type may be used for instance variables, temporary variables, parameter variables, and named constants, and may be returned from queries.

One use for characters is to control a robot from the keyboard. `Sim`, a superclass of `Robot`, has a protected method named `keyTyped` that is called each time a key is typed, yet it does nothing. The method has a `char` parameter containing the character that was typed. By overriding the method, we can tell a robot to move when '`m`' is typed, turn right when '`r`' is typed, and so on. The `KeyBot` class in Listing 7-5 defines such a robot. The same technique can be used in subclasses of `Intersection` and `Thing` because they all descend from `Sim`—the class implementing `keyTyped`. (When running a program using this feature, you must click on the image of the city before it will accept keystrokes. The image will have a black outline when it is ready to accept keystrokes.)

Listing 7-5: *A robot that responds to keystrokes*

```
1  import becker.robots.*;
2
3  public class KeyBot extends RobotSE
4  {
5    public KeyBot(City c, int str, int ave, Direction dir)
6    { super(c, str, ave, dir);
7    }
8
9    protected void keyTyped(char key)
10   { if (key == 'm' || key == 'M')
11     { this.move();
12     } else if (key == 'r' || key == 'R')
13     { this.turnRight();
14     } else if (key == 'l' || key == 'L')
15     { this.turnLeft();     // Watch out. The above test uses
16                            // a lowercase 'L', not a "one".
17   }
18 }
```

FIND THE CODE

ch07/keyBot/

The parameter, `key`, is compared to the letters 'm', 'r', and 'l' in lines 10, 12, and 14. In each case, if the comparison is true (that is, the parameter contains an 'm', 'r', or 'l'), an action is taken. If a different key is pressed, the robot does nothing. A slightly enhanced version of this method is implemented in the `RobotRC` class. You can extend `RobotRC` anytime you want to use the keyboard as a remote control (RC) for a robot.

KEY IDEA

Override `keyTyped` *to make a robot that can be controlled from the keyboard.*

The `'m'`, `'r'`, and `'l'` are character literals. To write a specific character value, place the character between two single quotes. What if you want to compare a value to a single quote? Placing it between two other single quotes (`''`) confuses the compiler, causing an error message. The solution is to use an **escape sequence**. An escape sequence is an alternative way to write characters that are used in the code for other purposes. The escape sequence for a single quote is `\'` (a backslash followed by a single quote). All escape sequences begin with a backslash. The escape sequence is placed in single quotes, just like any other character literal. Table 7-4 shows some common escape sequences, many of which have their origins in controlling printers.

KEY IDEA

Some characters have special meaning to Java. They have to be written with an escape sequence.

The last escape sequence, `\u`*dddd*, is used for representing characters from a wide range of languages, and includes everything from accented characters to Bengali characters to Chinese ideograms. You can find more information online at *www.unicode.org*. Unfortunately, actually using these characters requires corresponding fonts on your computer.

(table 7-4)

Character escape sequences

Sequence	Meaning
`\'`	Single quote
`\"`	Double quote
`\\`	Backslash
`\n`	Newline—used to start a new line of text when printing at the console
`\t`	Tab—inserts space so that the next character is placed at the next **tab stop**. Each tab stop is a predefined distance from the previous tab stop.
`\b`	Backspace—moves the cursor backwards over the previously printed character
`\r`	Return—moves the cursor to the beginning of the current line
`\f`	Form feed—moves the cursor to the top of the next page in a printer
`\u`*dddd*	A Unicode character, each *d* being a hexadecimal digit (0–9, a–f, A–F)

7.3.3 Using Strings

Strings of characters such as "Hello, karel!" are used frequently in Java programs. Strings are stored, appropriately, in variables of type `String`. A string can hold thousands of characters or no characters at all (the empty string). These characters can be the familiar ones found on the keyboard or those specified with escape characters, as shown in Table 7-4.

`String` is *not* a primitive type. In fact, it is a class just as `Robot` is a class. On the other hand, strings are used so often that Java's designers included special support for them that other classes do not have—so much special support that it sometimes feels like strings are primitive types.

Special Java Support for Strings

KEY IDEA

Java provides special support for the `String` *class.*

The special support the `String` class enjoys from the Java compiler falls into three categories:

- Java will automatically construct a `String` object for each sequence of characters between double quotes; that is, Java has literal values for strings just like it has literal values for integers (`5`, `-259`), doubles (`3.14159`), and Booleans (`true`).
- Java will "add" two strings together with the plus operator to create a new string consisting of one string followed by the other. This is called **concatenation**.
- Java will automatically convert primitive values and objects to strings before concatenating them with a string.

Listing 7-6 shows several examples of this special support. The program uses `System.out.println` to print the strings, as we did in Section 6.6.1. The difference here is the manipulations of the strings before they are printed.

Listing 7-6: *A simple program demonstrating built-in Java support for the* `String` *class*

FIND THE CODE

ch07/stringDemo/

```
1  import becker.robots.*;
2
3  public class Main
4  {
5    public static void main(String[] args)
6    { String greeting = "Hello";
7      String name = "karel";
8
9
10     System.out.println(greeting + "," + name + "!");
11
12
13
14     System.out.println("Did you know that 2*PI = " + 2*Math.PI + "?");
15
16     City c = new City();
17     Robot karel = new Robot(c, 1, 2, Direction.SOUTH);
18     System.out.println("c=" + c);
19   }
20 }
```

A `String` object is created automatically from the string literal "Hello"

Four strings are concatenated using the "+" operator to produce a single string, "Hello, karel!"

The primitive value resulting from this expression is automatically converted to a string and concatenated using the plus operator

The object referenced by `c` is automatically converted to a string by calling its `toString` method

Program output:

```
Hello, karel!
Did you know that 2*PI = 6.283185307179586?
c=becker.robots.City[SimBag[robots=[becker.robots.Robot
[street=1, avenue=2, direction=SOUTH, isBroken=false, numThings
InBackpack=0]], things=[]]]
```

In lines 6 and 7, two `String` objects are created using the special support the Java language provides for strings. These lines would look more familiar if they used a normal constructor, which works as expected:

```
String greeting = new String("Hello");
String name = new String("karel");
```

Line 14 contains an expression that is evaluated before it is passed as an argument to `println`. The normal rules of evaluation are used: multiplication has a higher precedence than addition, so `2*Math.PI` is evaluated first. Then, two string additions, or concatenations, are performed left to right. Because the left and right sides of the first

addition operator do not have the same type, the less general one (the result of `2*Math.PI`) is converted to a string before being "added" to the other operand.

Finally, when Java converts an object to a string, as it does in line 18, it calls the method named `toString`, which every class inherits from `Object`.

Overriding `toString`

KEY IDEA

Every class should override `toString` *to provide meaningful information.*

Java depends on the fact that every object has a `toString` method that can be called to provide a representation of the object as a string. The default implementation, inherited from the `Object` class, only prints the name of the class and a number identifying the particular object. To be useful, the method should be overridden in classes you write. The information it presents is often oriented to debugging, but it doesn't have to be.

The standard format for such information is the name of the object's class followed by an open bracket, "`[`". Information relevant to the object follows, and then ends with a closing bracket, "`]`". This format allows objects to be nested. For example, when the `City` object is printed, we see that it prints the `Robot` and `Thing` objects it references. Each of these, in turn, print relevant information about themselves, such as their location.

Listing 7-7 shows a `toString` method that could be added to the `SimpleBot` class shown in Listing 6-6.

toString

Listing 7-7: *A sample* `toString` *method*

```
1  public class SimpleBot extends Paintable
2  { private int street;
3    private int avenue;
4    private int direction;
5
6    // Constructor and methods are omitted.
7
8   /** Represent a SimpleBot as a string. */
9   public String toString()
10  { return "SimpleBot" +
11       " [street=" + this.street +
12       ", avenue=" + this.avenue +
13       ", direction=" + this.direction +
14       "]";
15  }
16 }
```

Querying a String

The `String` class provides many methods to query a `String` object. These include finding out how long a string is, whether two strings start the same way, the first location of a particular character, and so on. The most important of these queries are shown in Table 7-5.

(table 7-5)

Methods that query a string

Method	Description
`char charAt(int index)`	Returns the character at the location specified by the **index**. The index is the position of the character—an integer between 0 (the first character) and one less than the length of the string (the last character).
`int compareTo(String aString)`	Compares this string to `aString`, returning a negative integer if this string is lexicographically smaller than `aString`, 0 if the two strings are equal, and a positive integer if this string is lexicographically greater than `aString`.
`boolean equals(Object anObject)`	Compares this string to another object (usually a string). Returns `true` if `anObject` is a string containing exactly the same characters in the same order as this string.
`int indexOf(char ch)`	Returns the index within this string of the first occurrence of the specified character. If the character is not contained within the string, -1 is returned.
`int indexOf(char ch, int fromIndex)`	Returns the index within this string of the first occurrence of the specified character, starting the search at `fromIndex`. If no such character exists, -1 is returned.
`int indexOf(String substring)`	Returns the index of the first character of the first occurrence of the given substring within this string. If the given substring is not contained within this string, -1 is returned.
`int lastIndexOf(char ch)`	Returns the index of the last occurrence of the given character within this string. If the given character is not contained within this string, -1 is returned.
`int length()`	Returns the number of characters contained in this string.
`boolean startsWith(String prefix)`	Returns `true` if this string starts with the specified prefix.

KEY IDEA

Characters in strings are numbered starting at position `0`.

The `charAt` and `indexOf` methods in Table 7-5 refer to a character's index, or position within the string. In the string "Hello", 'H' is at index 0, 'e' is at index 1, and 'o' is at index 4. For example, if the variable `greeting` refers to "Hello", then `greeting.charAt(1)` returns the character 'e'.

It may seem strange for strings to begin indexing at zero, but this is common in computer science. We have already seen it in the robot cities, where streets and avenues begin with zero. We'll see it again in upcoming chapters, where collections of values are indexed beginning with zero.

KEY IDEA

>, >=, <, and <= don't work for strings.

When `a` and `b` are primitive types, we can compare them with operators such as `a == b`, `a < b`, and `a >= b`. For reference types such as `String`, only the == and `!=` operators work—and they do something different than you might expect.

KEY IDEA

Use the `equals` *method to compare strings for equality.*

Instead of ==, compare two strings for equality with the `equals` method. It returns true if every position in both strings has exactly the same character.

```
if (oneString.equals(anotherString))
{ System.out.println("The strings are equal.");
}
```

Instead of `!=`, use a Boolean expression, as follows:

```
!oneString.equals(anotherString)
```

KEY IDEA

Use `compareTo` *to compare strings for order.*

The string equivalent to less than and greater than is the `compareTo` method. It can be used as shown in the following code fragment:

```
String a = ...
String b = ...
if (a.compareTo(b) < 0)
{  // a comes before b in the dictionary
} else if (a.compareTo(b) > 0)
{  // a comes after b in the dictionary
} else  // if (a.compareTo(b) == 0)
{  // a and b are equal
}
```

The `compareTo` method determines the **lexicographic order** of two strings—essentially, the order they would have in the dictionary. To determine which of two strings comes first, compare the characters in each string, character by character, from left to right. Stop when you reach the end of one string or a pair of characters that differ. If you stop because one string is shorter than the other, as is the case with "`hope`" and "`hopeful`" in Figure 7-4, the shorter string precedes the longer string. If you stop because characters do not match, as is the case with "`f`" and "`l`" in "`hopeful`" and "`hopeless`", then compare the mismatched characters. In this case "`f`" comes before "`l`", and so "`hopeful`" precedes "`hopeless`" in lexicographic order.

```
hope
hopeful
hopeless
```

(figure 7-4)

Lexicographic ordering

If the strings have non-alphabetic characters, you may consult Appendix D to determine their ordering. For example, a fragment of the ordering is as follows:

```
! " # … 0 1 2 … 9 : ; < … A B C … Z [ \ … a b c … z { | }
```

This implies that "`hope!`" comes before "`hopeful`" because `!` appears before `f` in the previous ordering. Similarly, "`Hope`" comes before "`hope`".

Transforming Strings

Other methods in the `String` class do not answer questions about a given string, but rather return a copy of the string that has been transformed in some way. For example, the following code fragment prints "`Warning WARNING`"; `message2` is a copy of `message1` that has been transformed by replacing all of the lowercase characters with uppercase characters.

```
String message1 = "Warning";
String message2 = message1.toUpperCase();
System.out.println(message1 + " " + message2);
```

The designers of the `String` class had two options for the `toUppercase` method. They could have provided a command that changes all of the characters in the given string to their uppercase equivalents. The alternative is a method that makes a copy of the string, changing each lowercase letter in the original string to an uppercase letter in the copy.

KEY IDEA

An immutable class is one that does not provide methods to change its instance variables.

The designers of the `String` class consistently chose the second option. This makes the `String` class immutable. After a string is created, it cannot be changed. The methods given in Table 7-6, however, make it easy to create copies of a string with specific transformations. The `StringBuffer` class is similar to `String`, but includes methods that allow you to modify the string instead of creating a new one.

The `substring` method is slightly different. Its transformation is to extract a piece of the string, returning it as a new string. For example, if `name` refers to the string "`Karel`", then `name.substring(1,4)` returns "`are`". Recall that strings are indexed beginning with 0, so the character at index 1 is `a`. The second index to `substring`, 4 in this example, is the index of the first character *not* included in the substring.

(table 7-6)

Methods that return a transformed copy of a string

Method	Description
`String replace(char oldChar, char newChar)`	Returns a copy of this string that has all occurrences of `oldChar` replaced with `newChar`.
`String substring(int beginIndex, int endIndex)`	Returns a new string containing all the characters between `beginIndex` and `endIndex-1`, inclusive. The character at `endIndex` is the first character not included in the new string.
`String toLowerCase()`	Returns a copy of this string that has all the uppercase characters replaced with their lowercase equivalents.
`String toUpperCase()`	Returns a copy of this string that has all the lowercase characters replaced with their uppercase equivalents.
`String trim()`	Returns a copy of this string that has all white space (such as space, tab, and newline characters) removed from the beginning and end of the string.

Example: Counting Vowels

As an illustration of what you can do with strings, let's write a program that counts the number of vowels (a, e, i, o, u) in a string. As a test, we'll use the famous quotation from Hamlet, "To be, or not to be: that is the question." The expected answer is 13.

To begin solving this problem, let's think about how to solve it without a computer. One straightforward method is to look at each letter, proceeding from left to right. If the letter is a vowel, we can put a tick mark on a piece of paper. When we get to the end, the number of ticks corresponds to the number of vowels. Our program can adopt a similar strategy by using a variable to record the number of vowels.

This illustration will require us to examine individual letters in a string and compare letters to other letters. We don't have experience solving these kinds of problems, so let's proceed by solving a series of simpler problems. First, let's print the individual letters in the quotation. This shows that we can process the letters one at a time. After mastering that, let's count the number of times a single vowel, such as 'o', occurs. Finally, after solving these subproblems, we'll count all the vowels.

To print all the letters in the quotation, we must access each individual letter. According to Table 7-5, the `charAt` method will return an individual character from a string. However, it needs an index, a number between 0 and one less than the length of the string. Evidently, the `length` method will also be useful. To obtain the numbers between 0 and the length, we could use a `for` loop. We'll start a variable named `index` at 0 and increment it by 1 each time the loop is executed until `index` is just less than

the length. These ideas are included in the following Java program that prints each character in the quotation, one character per line.

```
 1 public static void main(String[] args)
 2 { String quotation = "To be, or not to be: that is the question.";
 3
 4   // Loop over each letter in the quotation.
 5   for (int index = 0; index < quotation.length(); index++)
 6   { // Examine one letter in the quotation.
 7     char ch = quotation.charAt(index);
 8     System.out.println(ch);
 9   }
10 }
```

Notice that the `for` loop starts the index at `0`, the first position in the string. The loop continues executing as long as the index is *less* than the length of the string. As soon as it equals the length of the string, it's time to stop. For example, Figure 7-5 illustrates a string of length 5, but its largest index is only 4. Therefore, the appropriate test to include in the `for` loop is `index < quotation.length()`.

KEY IDEA

A string of length 5 *has indices numbered* 0 *to* 4.

Index:	0	1	2	3	4
Characters:	T	o		b	e

(figure 7-5)

A string with its index positions marked

To modify this program to count the number of times 'o' appears, we can replace the `println` with an `if` statement and add a counter. The call to `println` in line 14 concatenates the value of our `counter` variable with two strings to make a complete sentence reporting the results. The modifications are shown in bold in the following code:

```
 1 public static void main(String[] args)
 2 { String quotation = "To be, or not to be: that is the question.";
 3
 4   int counter = 0;        // Count number of os.
 5   // Loop over each letter in the quotation.
 6   for (int index = 0; index < quotation.length(); index++)
 7   { // Examine one letter in the quotation.
 8     char ch = quotation.charAt(index);
 9     if (ch == 'o')
10     { counter += 1;
11     }
12   }
13
14   System.out.println("There are " + counter + " occurrences of 'o'.");
15 }
```

The last step is to count *all* the vowels instead of only the *o*s. A straightforward approach is to add four more `if` statements, all similar to the one in lines 9–11. However, when we consider that other quotations might include uppercase vowels (totaling 10 `if` statements), looking for an alternative becomes attractive.

We can reduce the number of tests if we first transform the quote using `toLowerCase`, as shown in line 3 of Listing 7-8. This assures us that all vowels will be lowercase.

KEY IDEA

`indexOf` searches a string for a particular character.

The `indexOf` method shown in Table 7-5 offers an interesting possibility. It will search a string and return the index of the first occurrence of a given character. If the character isn't there, `indexOf` returns -1. Suppose we take a letter from our quotation and search for it in a string that has only vowels. If the letter from the quotation is a vowel, it will be found and `indexOf` will return a 0 or larger. If it's not there, `indexOf` will return -1. This idea is implemented in Listing 7-8. The changes from the previous version are again shown in bold.

FIND THE CODE

ch07/countVowels/

Listing 7-8: *Searching a string to count the number of vowels*

```
1  public static void main(String[] args)
2  { String quotation = "To be, or not to be: that is the question";
3    String lowerQuote = quotation.toLowerCase();
4    String vowels = "aeiou";
5
6    int counter = 0;       // Count the number of vowels.
7    // Loop over each letter in the quotation.
8    for (int index = 0; index < lowerQuote.length(); index++)
9    { // Examine one letter in the quotation.
10     char ch = lowerQuote.charAt(index);
11     if (vowels.indexOf(ch) >= 0)
12     { counter += 1;
13     }
14   }
15
16   System.out.println("There are" + counter + "vowels.");
17 }
```

7.3.4 Understanding Enumerations

KEY IDEA

Java version 1.5 or higher is required to use this feature.

Programmers often need a variable that holds a limited set of values. For example, we may you need to store a person's gender—either male or female. For this we need only two values.

We could define some constants, using `m` for male and `f` for female, as follows:

```
public class Person extends Object
{ public static final char MALE = 'm';
  public static final char FEMALE = 'f';

  private String name;
  private char gender;

  public Person(String aName, char aGender)
  { super();
    this.name = aName;
    this.gender = aGender;
  }
  ...
}
```

But still, someone could create a `Person` object like this, either by mistake or maliciously:

```
Person juan = new Person("Juan", 'z');
```

Is Juan male or female? Neither. This mistake might create a severe problem later in the program. It might crash, or it could just cause embarrassment if Juan happened to be male and was assigned to a sports team with the following `if` statement:

```
if (juan.getGender() == MALE)
{ add to the boy's team
} else
{ add to the girl's team
}
```

A better solution is to define an **enumeration**, also called an **enumerated type**. An enumeration lists all of the possible values for that type. Those values can be used as literals in the program, and the compiler will allow only those literals to be used. This makes it impossible to assign Juan the gender of 'z'. `Direction`, used extensively in robot programs, is an example of an enumeration.

KEY IDEA

An enumeration has an explicitly listed set of values.

An enumeration for gender can be defined as shown in Listing 7-9. Like a class, the code is placed in a file matching the type name, `Gender.java`.

Listing 7-9: *Defining an enumeration*

```
1 /** An enumeration of the genders used in the Person class.
2 *
3 *  @author Byron Weber Becker */
4 public enum Gender
5 { MALE, FEMALE
6 }
```

FIND THE CODE

ch07/enums/

PATTERN

Enumeration

This is similar to a class definition except that the keyword `class` replaces the keyword `enum` and the enumeration does not include a clause to extend another class. Inside the braces, we list the different values for variables of type `Gender`, separating each with a comma.

KEY IDEA

The compiler guarantees that only valid values are assigned to enumerations.

The `Person` class shown earlier can be rewritten using this enumeration, as shown in Listing 7-10. Notice that `Gender` is used as a type, just like `int` or `Robot`, when the instance variable `gender` is declared in line 9. Similarly, it's used to declare a parameter variable in line 12 and a return type in line 19. In each of these cases, the Java compiler will guarantee that the value is `Gender.MALE`, `Gender.FEMALE`, or `null`—and nothing else. `null` is a special value that means "no value"; we will learn more about `null` in Section 8.1.2.

The `main` method in lines 24–29 uses the value `Gender.MALE` twice, once to construct a new `Person` object and once to test that the `getGender` method returns the expected value.

FIND THE CODE

ch07/enums/

PATTERN

Enumeration Test Harness

Listing 7-10: *Using the* `Gender` *enumerated type*

```
1  import becker.util.Test;
2
3  /** Represent a person.
4  *
5  *  @author Byron Weber Becker */
6  public class Person extends Object
7  {
8    private String name;
9    private Gender gender;
10
11   /** Construct a person. */
12   public Person(String aName, Gender aGender)
13   { super();
14     this.name = aName;
15     this.gender = aGender;
16   }
17
18   /** Get this person's gender. */
19   public Gender getGender()
20   { return this.gender;
21   }
22
23   // Test the Person class
24   public static void main(String[] args)
25   { Person juan = new Person("Juan", Gender.MALE);
26     Test tester = new Test();     // This line isn't needed. See Section 7.5.2.
```

Listing 7-10: *Using the* `Gender` *enumerated type* (continued)

```
27      tester.ckEquals("gender", Gender.MALE, juan.getGender());
28    }
29  }
```

7.4 Example: Writing a Gas Pump Class

We now have all the pieces needed to write a class that has nothing to do with robots. It could be a part of a drawing program, a payroll package, or a word processor.

LOOKING AHEAD

In this chapter's GUI section, we'll learn how to add a prepared user interface to this class.

For our first example, we'll start small and write a class that could be used as part of a gas pump. Every gas pump must have a meter to measure the gas that is delivered to the customer. Of course, measuring the gas is not enough. We must also be able to get the measurement so that we can display it on the gas pump. Our meter can also provide the following information:

- The price of one unit of gas (the price per liter or price per gallon)
- The octane level of the gas (a performance measure, typically between 87 and 93)
- A marketing name for that kind of gas (for example, "silver" or "ultra")
- The total cost of the gas delivered to the customer

In addition, when one customer is finished and another one arrives, we must be able to reset the measurements.

We'll call this class a `Meter`. To develop it, we'll build on the testing strategies outlined in Section 7.1.1 and include a `main` method for testing. An initial skeleton is shown in Listing 7-11. It extends `Object` because a meter doesn't seem to be based on any of the classes we've seen so far. It includes a constructor with no parameters (so far), and a `main` method for testing purposes. The `main` method creates a new `Meter` object, but there is nothing to test (yet).

Listing 7-11: *Beginning the* `Meter` *class*

```
1  public class Meter extends Object
2  {
3    public Meter()
4    { super();
5    }
6
7    // Test the class.
8    public static void main(String[] args)
```

Listing 7-11: *Beginning the* `Meter` *class* (continued)

```
 9    { Meter m = new Meter();
10    }
11 }
```

KEY IDEA

Write tests as you write the class.

We'll proceed by repeating the following steps until we think we're finished with the class:

- Choose one part of the description that we don't have working and decide what method is required to implement it.
- Understand what the method is to do and give it a name.
- Write one or more tests to determine if the method is working correctly.
- Write code so that the method passes the test(s).

7.4.1 Implementing Accessor Methods

Once again, we'll use a question-and-answer style to get started.

Expert What is one part of the description that isn't working?

Novice Well, nothing is working, so we're really just looking for something to implement. I think the first bullet in the description, to provide the price of one unit of gas, would be a good place to start.

Expert What is this method supposed to do?

Novice Return the price of one unit of gas. I guess that would be, for example, \$1.109/liter or \$2.85/gallon. I think a good name for the method would be `getUnitCost`.

Expert How can we test if `getUnitCost` is working correctly?

Novice We can add a statement in `main` that calls `ckEquals`:

```
tester.ckEquals("unit cost", 1.109, m.getUnitCost());
```

Expert So, you're assuming that gas costs \$1.109 per liter?

Novice Yes.

Expert How would you implement `getUnitCost` so that it passes this test?

Novice It's easy. Just return the value. I'll even throw in the documentation:

```
/** Get the cost per unit of fuel.
*   @return cost per unit of fuel */
public double getUnitCost()
{ return 1.109;
}
```

A number with a decimal point like `1.109` can be stored in a variable of type `double`, so that will be the return type of the method.

Expert Aren't you assuming that gas is always $1.109 per liter? What if the price goes up or down? Or what if the gas pump can deliver three different grades of gasoline? Surely they wouldn't all have the same price.

Novice I see your point. Somehow each `Meter` object should have its own price for the gas it measures, just like each `Robot` object must have its own street and avenue. To test that, we want to have two `Meter` objects, each with a different price:[1]

```
tester.ckEquals("Cost 1", 1.109, m1.getUnitCost());
tester.ckEquals("Cost 2", 1.159, m2.getUnitCost());
```

It sounds like we need to have an instance variable to store the unit price.

Expert Suppose you had an instance variable. How would you initialize it?

Novice Well, it couldn't be where the instance variable is declared because then we're right back where we started—each `Meter` object would always have the same price for its gas. I guess we'll have to initialize it in the constructor. I think that means the constructor requires a parameter so that the price can be specified when the `Meter` object is created.

Putting these observations together results in the class shown in Listing 7-12. It adds an instance variable, `unitCost`, at line 5 to remember the unit cost of the gas for each `Meter` object. The instance variable is initialized at line 11 using the parameter variable declared in line 9. In line 22, the value `1.109` is passed to the `Meter` constructor. This value is copied into the parameter variable `unitCost` declared in line 9. The value in `unitCost` is then copied into the instance variable in line 11. The value is stored in `unitCost` for as long as the object exists (or it is changed with an assignment statement).

Finally, the contents of `unitCost` are returned at line 17 each time `getUnitCost` is called.

[1] `ckEquals` verifies that two `double` values differ by less than 0.000001.

Listing 7-12: *A partially completed* `Meter` *class*

```
1  import becker.util.Test;
2
3  public class Meter extends Object
4  {
5    private double unitCost;
6
7    /** Construct a new Meter object.
8    *  @param unitCost The cost for one unit (liter or gallon) of gas */
9    public Meter(double unitCost)
10   { super();
11     this.unitCost = unitCost;
12   }
13
14   /** Get the cost per unit of fuel.
15   *  @return cost per unit of fuel */
16   public double getUnitCost()
17   { return this.unitCost;
18   }
19
20   // Test the class.
21   public static void main(String[] args)
22   { Meter m1 = new Meter(1.109);
23     Test tester = new Test(); // This line isn't needed. See Section 7.5.2.
24     tester.ckEquals("unit cost", 1.109, m1.getUnitCost());
25     Meter m2 = new Meter(1.149);
26     tester.ckEquals("unit cost", 1.149, m2.getUnitCost());
27   }
28 }
```

Two other parts of the requirements—getting the octane level and getting the marketing name—follow a similar strategy. The difference is that they will use an integer and a `String`, respectively. See Listing 7-13 for their implementations.

7.4.2 Implementing a Command/Query Pair

Expert So, how are you going to implement the actual measurement of the gas? Wasn't the point of the `Meter` class to measure how much gas is delivered to a customer?

Novice Yes. Somehow, it seems we need to find out when the pump is actually pumping gas—and how much. You know, when the handle is only squeezed a little way, only a little gas flows from the pump into the car. But when you squeeze the handle all the way, a lot of gas flows.

Expert It sounds like the pump—the code that is going to be using your `Meter` class—needs to call a method every time a little bit of gas is pumped. Does it get called repeatedly?

Novice Yes, and it needs to tell how much gas was pumped in that time. The job of the `Meter` object is to keep track of the units of gas that are pumped.

Expert I'm getting confused. Can you explain it another way?

Novice Sure. Think of a real pump. It has a motor to pump the gas. Every time the motor goes around, some gas is pumped. How much depends on the speed of the motor.

In our system, it's as if the motor called a method in the `Meter` class every time it turns. Furthermore, it will tell that method how much gas it pumped. If the motor is turning slowly, it pumps only a small amount of gas; but if the motor is turning fast, it pumps more. We'll add up all the units of gas that are pumped to calculate the total amount delivered to the customer.

Expert What do you want to call this method that is called by the motor?

Novice How about calling it `pump`? It will need a parameter, so the full signature will be

```
public void pump(double howMuch)
```

It's a command, not a query, so the return type is `void`.

Expert How would you test this method? How will you know if it's working correctly?

Novice It's like the `move` method in the `Robot` class. To test it, we had to have some queries: `getAvenue` and `getStreet`. For the `Meter` class, we'll need a query—something like `getVolumeSold`.

Expert How will that help you?

Novice First, we'll call `pump` to "pump" some gas. Maybe we'll call it several times, just like the real pump would. Then we'll call `getVolumeSold` and make sure that the value it returns matches the amount we "pumped." We could put the following in the test harness:

```
Meter m = new Meter(1.109);
tester.ckEquals("vol. sold", 0.0, m.getVolumeSold());
m.pump(0.02);
m.pump(0.03);
m.pump(0.01);
tester.ckEquals("vol. sold", 0.06, m.getVolumeSold());
```

Expert How will you implement these methods?

Novice Well, somehow we need to add up all the units of gas that get passed as an argument to the `pump` command. I'm thinking of using a temporary variable inside the `pump` command.

Expert Are you sure about that? Doesn't a temporary variable disappear each time the method is finished, only to be re-created the next time the method is called? Besides, how would `getVolumeSold` get access to a temporary variable?

Novice You're right. We should use an instance variable instead. It maintains a value even when a method is not being executed—and every method, including `getVolumeSold`—can access an instance variable.

Expert Please recap the plan for me.

Novice We'll have an instance variable called `volumeSold`. It will be initialized to `0.0` when the `Meter` object is created. Every time `pump` is called, it will add the value passed in the parameter variable to `volumeSold`. Each time `getVolumeSold` is called, we'll just return the current contents of the `volumeSold` instance variable.

Expert Sounds good. What about resetting when a new customer comes? That was another one of the requirements. I think we're also supposed to return the cost of the gas sold.

Novice We'll create a `reset` method that will assign `0.0` to the `volumeSold` instance variable. A method named `calcTotalCost` can simply return the volume sold times the cost per unit. Both of those values will be stored in instance variables.

Expert And your plan for testing?

Novice Much like the others. We'll set up a `Meter` object with a known unit price for the gas. We'll "pump" some gas and then call `getVolumeSold` and `calcTotalCost`. Then we can reset the pump and verify that the volume sold is back to 0.

This plan is a good one and is implemented in Listing 7-13.

Listing 7-13: *The completed code for the* `Meter3` *class*

FIND THE CODE

ch07/gasPump/Meter3.java

```
1  import becker.util.Test;
2
3  /** Measure the volume of fuel sold and calculate the amount owed by the
4  *  customer, given the current fuel cost.
5  *
6  *  @author Byron Weber Becker */
7  public class Meter3 extends Object
8  {
9    private double unitCost;            // unit cost
10   private double volumeSold = 0.0;    // volume sold
11   private int octane;                 // octane rating
12   private String label;               // marketing label
13
14   /** Construct a new Meter object.
15   *  @param unitCost The cost for one unit (liter or gallon) of gas
16   *  @param octaneRating An integer related to the "performance" of
17   *  the fuel;  usually between 87 and 93.
18   *  @param theLabel A label for the fuel, such as "Gold" or "Ultra". */
19   public Meter3(double unitCost, int octaneRating,
20                        String theLabel)
21   { super();
22     this.unitCost = unitCost;
23     this.octane = octaneRating;
24     this.label = theLabel;
25   }
26
27   /** Get the cost per unit of fuel.
28   *  @return cost per unit of fuel */
29   public double getUnitCost()
30   { return this.unitCost;
31   }
32
33   /** Get the octane rating of the fuel.
34   *  @return octane rating (typically between 87 and 93) */
35   public int getOctane()
36   { return this.octane;
```

Listing 7-13: *The completed code for the* `Meter3` *class* (continued)

```
37    }
38
39    /** Get the label for this meter's fuel.  For example, "Gold" or "Ultra".
40    *  @return this meter's fuel label */
41    public String getLabel()
42    { return this.label;
43    }
44
45    /** Pump some fuel into a tank.  This method is called
46    *  repeatedly while the "handle" on the pump is pressed.
47    *  @param howMuch How much fuel was pumped since the last time
48    *  this method was called. */
49    public void pump(double howMuch)
50    { this.volumeSold = this.volumeSold + howMuch;
51    }
52
53    /** Get the volume of fuel sold to this customer.
54    *  @return volume of fuel sold */
55    public double getVolumeSold()
56    { return this.volumeSold;
57    }
58
59    /** Calculate the total cost of fuel sold to this customer.
60    *  @return price/unit * number of units sold */
61    public double calcTotalCost()
62    { double tCost = this.unitCost * this.volumeSold;
63      return tCost;
64    }
65
66    /** Reset the meter for a new customer. */
67    public void reset()
68    { this.volumeSold = 0.0;
69    }
70
71    // Test the class.
72    public static void main(String[] args)
73    { Test tester = new Test();
74      Meter3 m1 = new Meter3(1.109, 87, "Regular");
75      tester.ckEquals("unit cost", 1.109, m1.getUnitCost());
76      tester.ckEquals("octane", 87, m1.getOctane());
77      tester.ckEquals("label", "Regular", m1.getLabel());
78
79      Meter3 m2 = new Meter3(1.149, 89, "Ultra");
```

Test Harness

Listing 7-13: *The completed code for the* `Meter3` *class* (continued)

```
80        tester.ckEquals("unit cost", 1.149, m2.getUnitCost());
81        tester.ckEquals("octane", 89, m2.getOctane());
82        tester.ckEquals("label", "Ultra", m2.getLabel());
83
84        tester.ckEquals("volSold", 0.0, m2.getVolumeSold());
85        m2.pump(0.02);
86        m2.pump(0.03);
87        m2.pump(0.01);
88        tester.ckEquals("volSold", 0.06, m2.getVolumeSold());
89        tester.ckEquals("totCost", 0.06*1.149, m2.calcTotalCost());
90        m2.reset();
91        tester.ckEquals("after reset", 0.0, m2.getVolumeSold());
92        tester.ckEquals("after reset", 0.0, m2.calcTotalCost());
93    }
94 }
```

7.5 Understanding Class Variables and Methods

KEY IDEA

Variables and methods declared with `static` *apply to the entire class.*

So far we have studied instance variables, temporary variables, parameter variables, constants, and methods. We now need to look at variables and methods that use the `static` keyword. Such variables and methods apply to the entire class rather than to a single object.

7.5.1 Using Class Variables

Instance variables are always associated with a specific object. Each `Robot` object knows which avenue and street it is on. Each `Meter` object knows the price of the gas it is measuring.

A **class variable** (also called a **static variable**) relates to the class as a whole rather than to an individual object. A class variable is declared using the `static` keyword and is used to store information common to *all* the instances of the class.

Consider an analogy: suppose that `people` are objects and that all `people` live in the same town. Some information is specific to each individual—their name, age, birth date, and so on. This information is stored in instance variables. But other information is known by everyone—the current year, the name of the town, the name of the mayor, whether the sun is up, and so on. In this situation, it doesn't make sense for each `person` object to have its own instance variable to store the year. Using a class variable,

the year is stored only once but is still accessible to each `person` object. Using an instance variable, there are as many copies of the year as there are `person` objects.

A class variable is declared like an instance variable but includes the `static` keyword:

```
public class Person extends Object
{ ...
  private int birthYear;
  private static int year;          // a class variable
  ...
}
```

Inside the class, a class variable can be accessed using the name of the class, the name only, or `this`. For example, here are three different implementations of the method `getAge`:

```
public int getAge()
{ return Person.year - this.birthYear;
}

public int getAge()
{ return year - this.birthYear;
}

public int getAge()
{ return this.year - this.birthYear;
}
```

KEY IDEA

Access a class variable with the name of the class containing it.

Of these three, the first is preferred because it is clear that `year` is a class variable. The second example is probably the most common because it saves a few keystrokes (`this` could also be omitted for `birthYear`). Accessing the year with `this.year` strongly implies that `year` is an instance variable and is discouraged.

A method may also change a class variable. For example, the following method could be used on January 1:

```
public void incrementYear()
{ Person.year = Person.year + 1;
}
```

The effect of this is to change the year for every `Person` object—and it's accomplished with only *one* method call.

Class variables are created and initialized before a class is first used. They are set up even before the first object is created for that class.

Assigning Unique ID Numbers

One use of class variables is to assign individual objects identification numbers. For example, suppose that we want each person to have a unique identification number. Obviously, if each person has a unique number, we need to store it in an instance variable. We can use a class variable to make it unique, as follows:

- Declare a class variable to store the ID number to assign to the next `Person` object created.
- In the `Person` constructor:
  - Assign the ID number using the class variable.
  - Increment the class variable in preparation for assigning the next number.

```
public class Person extends Object
{ ...
  privatefinal int id;
  private static int nextID = 1000000;     // first id is 1000000
  ...

  public Person(String name)
  { super();
    this.id = Person.nextID;
    Person.nextID++;
    ...
  }
}
```

PATTERN

Assign a Unique ID

With this scheme, every time a `Person` object is created, it is assigned an ID number. Because `nextID` is a class variable and is incremented as soon as it has been assigned, the next `Person` object constructed will receive the next higher number.

A Guideline for Class Variables

Class variables are quite rare in object-oriented code. If you find yourself declaring a class variable, you should be able to clearly explain why *every* instance of the class should access the same value or why there won't be any instances of the class at all.

KEY IDEA

Class variables are relatively rare in object-oriented code.

7.5.2 Using Class Methods

The `static` keyword can also be applied to methods; doing so, however, involves a trade-off. On the one hand, such a method cannot access any instance variables and are limited to calling methods that are also declared `static`. On the other hand, class methods can be called using only the name of the class. Because no object is needed, this makes them easier to use in some circumstances.

KEY IDEA

Class methods cannot use instance variables or nonstatic methods.

We can use two methods in the previous section as examples. The method `getAge` *cannot* be a class method because it accesses an instance variable. However, `incrementYear` is a perfect candidate because it accesses only a class variable. To make it into a class method, add the `static` keyword as shown in the following code fragment:

```
public static void incrementYear()
{ Person.year = Person.year + 1;
}
```

KEY IDEA

Class methods can be called without using an object.

With this change, the year can be incremented as follows:

```
Person.incrementYear();
```

This works even if no `Person` objects have been created yet. Using a specific object such as `john.incrementYear()` also works but using the class name is preferred because it tells the reader that `incrementYear` applies to the entire class.

Class Methods in the `Math` Class

One of Java's provided classes, `java.lang.Math`, contains *only* class methods. For example, consider a method to calculate the maximum of two numbers:

```
public static int max(int a, int b)
{ int answer = a;
  if (b > a)
  { answer = b;
  }
  return answer;
}
```

This method does not use any instance variables. In fact, all of the methods in the `Math` class are like this. Because the `Math` class does not have any instance variables, all of the methods are static. Thus, all of the methods are called using the class name, `Math`, as a prefix, as shown in the following example:

```
int m = Math.max(0, this.getStreet());
```

Most of the functions in the `Math` class are listed in Table 7-7. Some of them are overloaded with different numeric types for their parameters.

Method	Returned Value
`int abs(int x)` `double abs(double x)`	absolute value of *x*; also overloaded for `long` and `float`
`double acos(double x)`	arccosine of *x*, $0.0 \le x \le \pi$
`double asin(double x)`	arcsine of *x*, $-\pi/2 \le x \le \pi/2$
`double atan(double x)`	arctangent of *x*, $-\pi/2 \le x \le \pi/2$
`double cos(double x)`	cosine of the angle *x*, where *x* is in radians
`double exp(double x)`	*e*, the base of natural logarithms, raised to the power of *x*
`double log(double x)`	natural logarithm (base *e*) of *x*
`int max(int x, int y)` `double max(double x, double y)`	larger of *x* and *y*; also overloaded for `long` and `float`
`int min(int x, int y)` `double min(double x, double y)`	the smaller of *x* and *y*; also overloaded for `long` and `float`
`double pow(double x, double y)`	*x* raised to the power of *y*
`double random()`	random number greater than or equal to 0.0 and less than 1.0
`long round(double x)`	integer nearest *x*
`double sin(double x)`	sine of the angle *x*, where *x* is in radians
`double sqrt(double x)`	square root of *x*
`double tan(double x)`	tangent of the angle *x*, where *x* is in radians
`double toDegrees(double x)`	converts an angle, *x*, measured in radians to degrees
`double toRadians(double x)`	converts an angle, *x*, measured in degrees to radians

(table 7-7)

Many of the mathematical functions included in `java.lang.Math`

In addition to these functions, `java.lang.Math` also includes two public constants: `PI` (3.14159...) and `E` (2.71828...).

In Section 7.1.2, we wrote our own version of the absolute value function to use in the `distanceToOrigin` query. We now know that we could have used the `Math` class, as follows:

```
public int distanceToOrigin()
{ return Math.abs(this.street) + Math.abs(this.avenue);
}
```

The absolute value function is overloaded for both `int` and `double`. Because `street` and `avenue` are integers, Java selects the method with `int` parameters (which happens to have an `int` return type).

Our version of `distanceToOrigin` was the "as the robot moves" interpretation. If we wanted the "as the crow flies" interpretation, we could use the Pythagorean theorem ($a^2 + b^2 = c^2$) and the square root function, as follows:

```
public double distanceToOrigin()
{ double a2 = this.street * this.street;   // one way to square a #
  double b2 = Math.pow(this.avenue, 2.0); // another way to square a #
  return Math.sqrt(a2 + b2);
}
```

In Section 7.2.3 we discussed casting. For example, when the variable `d` holds `3.999`, the statement `int i = (int)d` assigns the value `3` to the variable `i`. In many cases, however, we want the nearest integer, not just the integer portion. For example, we want to round `3.999` to `4`.

The `Math` class has a `round` method that will do just that. However, when the method is passed a `double` as an argument it returns a `long` integer. This implies that we often cast the result when working with integers. For example,

```
int i = (int)Math.round(d);
```

KEY IDEA

`random` returns a pseudorandom number, x, such that $0 \le x < 1$.

One of the most fun methods in the `Math` class is `random`. Each time it is called, it returns a number greater than or equal to 0 and less than 1. When called repeatedly, the sequence of numbers appears to be random.[2] The first 10 numbers returned in one experiment are shown in Figure 7-6. The first number in the sequence depends on the date and time the program begins running.

(figure 7-6)

Sequence of 10 pseudorandom numbers

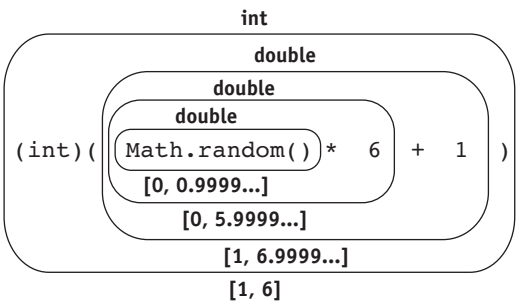
```
0.425585145743809
0.49629326982879207
0.4467070769009338
0.23377387885697887
0.33762066427975934
0.25442482711460535
0.9986103921074468
0.9822012645708958
0.420499613228824
0.22309030308848088
```

[2] These numbers appear to be random but are not. If the numbers were really random, the next number could not be predicted. Because the next number in these sequences can be predicted, they are called "pseudorandom."

A computer implementation of a game with dice will often use `random` to simulate the dice. In this case, we need to map a `double` between 0 and 1 to an integer between 1 and 6. The following method will do so:

```
public int rollDie()
{ return (int)(Math.random() * 6 + 1);
}
```

We can understand how this works with a slight variation of an evaluation diagram, as shown in Figure 7-7. The fact that the `random` method returns a value greater than or equal to 0 and less than 1 is reflected below its oval with the notation [0, 0.9999...]. After the multiplication by 6, the expression has a value in the range [0, 5.9999...], a number greater than or equal to 0 and less than 6. After the other operations are carried out, we see that the result is an integer between one and six—exactly what is needed to simulate rolling a die.



(figure 7-7)

Evaluating an expression used in simulating dice

Class Methods in the `Character` Class

The `Character` class is automatically imported into every Java class and includes a number of methods for classifying characters. A selection of these methods is shown in Table 7-8. They are all declared `static` and can be called using the `Character` class name, as shown in the following example:

```
if (Character.isDigit(ch))...
```

(table 7-8)

A selection of useful methods in the `Character` *class*

Method	Returned value
`boolean isDigit(char ch)`	`true` if the specified character is a digit; `false` otherwise
`boolean isLetter(char ch)`	`true` if the specified character is a letter; `false` otherwise
`boolean isLowerCase(char ch)`	`true` if the specified character is a lowercase character; `false` otherwise
`boolean isUpperCase(char ch)`	`true` if the specified character is an uppercase character; `false` otherwise
`boolean isWhitespace(char ch)`	`true` if the specified character is white space (space, tab, new line, etc.) ; `false` otherwise

Class Methods in the `Test` Class

The methods in the `Test` class are other examples of class methods. The code for `ckEquals`, as shown in Listing 7-2, does not use any instance variables or nonstatic methods and can therefore be declared with the `static` keyword. Indeed, all the methods in the class are like this.

Because the methods are static, a typical test harness will not instantiate a `Test` object. For example, the test harness in Listing 7-1 could be rewritten by removing line 14 and replacing `tester` with `test` in lines 15–17, as follows:

```
 1 import becker.util.Test;
 2
 3 public class TestHarness extends Object
 4 {
 5   public static void main(String[] args)
 6   { // Set up a known situation (an empty city; a robot on (4, 2) facing east).
 7     SimpleCity c = new SimpleCity();
 8     SimpleBot karel = new SimpleBot(c, 4, 2, Constants.EAST);
 9
10     // Execute the move method.
11     karel.move();
12
13     // Verify the results -- robot on intersection (4, 3).
14     ~~Test tester = new Test();~~
15     ~~tester~~Test.ckEquals("new ave", 3, karel.getAvenue());
16     ~~tester~~Test.ckEquals("same str", 4, karel.getStreet());
17     ~~tester~~Test.ckEquals("same dir", Constants.EAST,
18                                        karel.getDirection());
19   }
20 }
```

The `City` class in the `becker` library automatically displays the city which is usually not desirable in a test harness. This behavior can be controlled with another class method, `showFrame`. The following code fragment shows how to use this method to avoid having the city show.

```
public static void main(String[] args)
{ City.showFrame(false);
  City c = new City();
  ...
}
```

The `main` Method

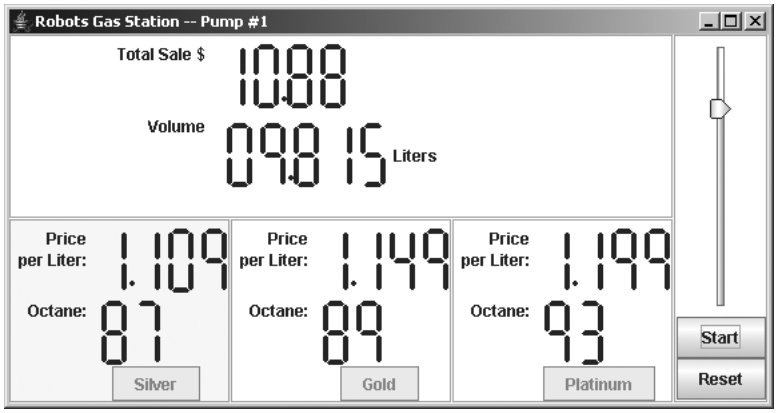
We also write a class method for every program—`main`. Java requires `main` to be static so that the Java system can call it using only the name of the class containing it. The name of the class is passed to the Java system when the program is run.

7.6 GUI: Using Java Interfaces

The `becker.xtras` package contains a number of graphical user interfaces that can be used to make programs that look and feel more professional than we can write with the skills learned so far. The `GasPumpGUI` class in the `becker.xtras.gasPump` package is an example; it can be used with the `Meter` class we developed earlier in this chapter to create a program with the graphical user interface shown in Figure 7-8.

LOOKING AHEAD

The skills to write a graphical user interface like `GasPumpGUI` *are covered in Chapter 13.*



(figure 7-8)

Image of the graphical user interface provided by the `gasPump` *package*

The problem set refers to several such GUIs in the `becker.xtras` package. A problem will often begin by directing you to explore the documentation for a particular package. You may want to do that now for the `gasPump` package. Go to *www.learningwithrobots.com*.

Navigate to "Software" and then "Documentation." In the large panel on the right, click `becker.xtras.gasPump`. You'll see a brief description of each of the classes included in the package. Scroll down and you'll find an image of the graphical user interface and a sample `main` method that you can use to run the program (see Listing 7-14).

This gas pump user interface is set up for a program that uses three instances of the `Meter` class—one for each of three different octane levels. Of course, each octane level has its own price.

FIND THE CODE

ch07/gasPump/

Listing 7-14: *A sample* `main` *method to run our class (*`Meter`*) with the provided graphical user interface*

```
1  import becker.xtras.gasPump.*;
2
3  /** Run a gas pump with a graphical user interface.
4  *
5  *  @author Byron Weber Becker */
6  public class Main extends Object
7  {
8    public static void main(String[] args)
9    { // Create three meters for the pump.
10     Meter silver = new Meter(1.109, 87, "Silver");
11     Meter gold = new Meter(1.149, 89, "Gold");
12     Meter platinum = new Meter(1.199, 93, "Platinum");
13
14     // Create the graphical user interface.
15     GasPumpGUI gui = new GasPumpGUI(
16                                  silver, gold, platinum, "Liter");
17   }
18 }
```

7.6.1 Specifying Methods with Interfaces

The graphical user interface class `GasPumpGUI` will not work with just any class. It must somehow be assured that the `Meter` objects passed in lines 15 and 16 have methods to get the price of the gasoline, the octane level, how much gas has been pumped to the current customer, and how much that gasoline is worth so that it can display this information to the user. Furthermore, just having methods that perform these functions is not enough. The methods must be named exactly as `GasPumpGUI` expects, return the expected types of values, and take the expected arguments; otherwise, it won't be able to call them.

This is a common problem: Two classes need to work together, but they are written by different people at different times and places. This problem was also faced by the programmers who wrote the classes used in graphical user interfaces, such as `JComponent`, `JButton`, and `JFrame`. To fully exploit their functionality, classes written several years ago must be assured that objects we give them possess methods with specified signatures.

Fortunately, Java provides a solution. The person who writes the first class also provides a list of the methods it requires to be in the second class. The list written by the author of `GasPumpGUI` includes the following methods:

```
public double getUnitCost();
public double getVolumeSold();
public int getOctane();
public String getLabel();
public void reset();
public void pump(double howMuch);
public double calcTotalCost();
```

This list, together with documentation, is put into a Java **interface**. Unfortunately, the word *interface* has two meanings in this section. One meaning is "graphical user interface," like the one shown in Figure 7-8. The other meaning—the one intended here—is a Java file used to guarantee that a class contains a specified set of methods.

KEY IDEA

A Java interface is used to guarantee the presence of specified methods.

Listing 7-15 shows a complete interface except for the documentation, and is similar to a class. It has a name (`IMeter`) and must be in a file with the same name as the interface (`IMeter.java`). The list of methods is enclosed in curly braces. Interfaces may also have constants, defined as they would be defined in a class. An interface should be documented like a class.

The differences between an interface and a class are as follows:

- An interface uses the keyword `interface` instead of `class`.
- An interface cannot extend a class.[3]
- Method bodies are omitted. Each method lists its return type and signature. If an access modifier is present, it must be `public` (all methods in an interface are assumed to be `public`).

[3] It is possible, however, for an interface to extend another interface. In fact, it can extend several interfaces, but that's beyond the scope of this textbook.

Listing 7-15: *An interface for* `Imeter` *(documentation is omitted to better show the essential structure)*

```
1  public interface IMeter
2  {
3    public double getUnitCost();
4    public int getOctane();
5    public String getLabel();
6    public double getVolumeSold();
7    public void reset();
8    public void pump(double howMuch);
9    public double calcTotalCost();
10 }
```

7.6.2 Implementing an Interface

KEY IDEA

An interface name can be used as the type in variable declarations.

So, how is the `IMeter` interface used? The author of `GasPumpGUI` used it in at least one place—defining the type of object required by the constructor. From the online documentation, we know that the constructor's signature is as follows:

```
public GasPumpGUI(IMeter lowOctane, IMeter medOctane,
                  IMeter highOctane, String volumeUnit)
```

KEY IDEA

An interface name can be used to declare the type of a variable.

As this example shows, an interface can be used as the type in a variable declaration, including parameter variables, temporary variables, and instance variables.

The way we use `IMeter` is in the line that begins the definition of the `Meter` class:

```
public class Meter extends Object implements IMeter
```

KEY IDEA

An interface name is used in a class declaration.

It's the last part—`implements IMeter`—that tells the Java compiler that our class must be sure to implement each method listed in `IMeter` and that the signatures must match exactly. This phrase is also the part that allows a `Meter` object to be passed to a `GasPumpGUI` constructor even though the constructor's signature says the argument should be an `IMeter` object.

There is no required relationship between the names `IMeter` and `Meter`, although they are often similar. Both names are chosen by programmers, but should follow conventions. In the `becker` library, the convention is for interface names to begin with `I`. What follows the `I` should give an indication of the interface's purpose. The person implementing the `Meter` class can choose any name he wants, but should of course follow the usual conventions for naming a class.

A class can implement as many interfaces as required, although implementing only one is the usual case. To implement more than one, list all the interfaces after the `implements` keyword, separating each one from the next with a comma.

What happens if the `Meter` class omits one of the methods specified by `IMeter`, say `calcTotalCost`? The Java compiler will print an error message and refuse to compile the class. The error message might refer to a missing method or might say that the class "does not override the abstract method `calcTotalCost`."

7.6.3 Developing Classes to a Specified Interface

In a sense, we developed the `Meter` class in a backwards fashion. We first wrote the class and then found out that it just happened to match the `IMeter` interface. A more usual situation is one where we know, at the beginning, that we will be implementing a particular interface. Suppose, for example, that our instructions were to develop a class to use with the graphical user interface in the `becker.xtras.gasPump` package. As soon as we investigated the package, we would know that we need to write a class implementing the `IMeter` interface.

How should we proceed?

Begin by creating a class with your chosen name and a `main` method to be used for testing. Add the `implements` keyword followed by the name of the interface. Add the methods specified by the interface and a constructor that is implied by the `main` method shown in the documentation. For each method with a non-`void` return type, add a `return` statement—it doesn't matter what value is returned—so that the method will compile. Such a method is called a stub. Following these clues results in the skeleton shown in Listing 7-16. Some development environments will do this much for you almost automatically.

LOOKING BACK

A stub is a method with just enough code to compile. Stubs were first discussed in Section 3.2.2.

Finally, write tests and develop the methods, as we did earlier in this chapter.

Listing 7-16: *Beginning the* `Meter` *class with methods required to implement* `IMeter`

```
1  import becker.xtras.gasPump.IMeter;
2  import becker.util.Test;
3
4  public class Meter extends Object implements IMeter
5  {
6    public Meter(double unitCost, int octaneRating,
7                 String theLabel)
8    { super();
9    }
10
```

FIND THE CODE

ch07/gasPump/

Listing 7-16: *Beginning the* `Meter` *class with methods required to implement* `IMeter` (continued)

```
11    public double getUnitCost()
12    { return 0.0;
13    }
14
15    public int getOctane()
16    { return 0;
17    }
18
19    public String getLabel()
20    { return "dummy";
21    }
22
23    public void pump(double howMuch)
24    {
25    }
26
        ...
38
39    /** To use for testing. */
40    public static void main(String[] args)
41    { Meter m = new Meter(1.109, 87, "Regular");
42    }
43 }
```

7.6.4 Informing the User Interface of Changes

Graphical user interfaces often use a pattern known as Model-View-Controller. We will study this pattern in depth in Chapter 13, which is devoted to writing graphical user interfaces.

KEY IDEA

The model must inform the user interface when changes have been made.

The `Meter` class is the "model" part of this pattern. It keeps track of the information that the user interface—the "view" and "controller" parts—displays. The model must inform the user interface each time information on the screen needs updating. In practice, this means calling a method named `updateAllViews` at the end of each method in `Meter` that changes an instance variable. This can always be done in the same way, as shown in Listing 7-17. The changes from the previous version of `Meter` (Listing 7-13) are shown in bold.

Listing 7-17: *Code required in the* `Meter` *class to inform the view of changes*

```
1  import becker.gasPump.IMeter;
2  import becker.util.*;
3
4  public class Meter extends Object implements IMeter
5  { // Instance variables...
6    private ViewList views = new ViewList();
7
8    // Methods that do not change instance variables...
9
10   // Methods that do change instance variables...
11   public void reset()
12   { this.volumeSold = 0;
13     this.views.updateAllViews();
14   }
15
16   public void pump(double howMuch)
17   { this.volumeSold += howMuch;
18     this.views.updateAllViews();
19   }
20
21   public void addView(IView aView)
22   { this.views.addView(aView);
23   }
24 }
```

FIND THE CODE

ch07/gasPump/

`views`, declared in line 6, is an object that maintains a list of graphical user interface parts (the views) that need to be updated when this model changes. The class, `ViewList`, is imported from the package `becker.util` in line 2.

The graphical user interface adds views to this list by calling the method `addView`, which is declared in lines 21–23. It receives a parameter that implements the `IView` interface. By using an interface, we don't need to know exactly what kind of object is passed as an argument—only that it includes the methods named in `IView`. The `addView` method doesn't actually do anything with the view except tell the list of views to add it.

With this infrastructure in place, the last step is to call the `updateAllViews` method in the `views` object at the appropriate times. It should be called at the end of each method in the model that changes an instance variable. What happens if you forget to call `updateAllViews`? The user interface will not change when you expect it to.

Finally, `addView` is in the `IMeter` interface even though it was omitted from Listing 7-15.

7.7 Patterns

7.7.1 The Test Harness Pattern

Name: Test Harness

Context: You want to increase the reliability of your code and make the development process easier with testing.

Solution: Write a `main` method in each class. The following template applies:

```
import becker.util.Test;
public class «className» ...
{  // Instance variables and methods
   ...
   public static void main(String[] args)
   {  // Create a known situation.
      «className» «instance» = new «className»(...);

      // Execute the code being tested.
      «instance».«methodToTest»(...);

      // Verify the results.
      Test.ckEquals(«idString», «expectedValue»,
                                «actualValue»);
      ...
   }
}
```

Verifying the results will often require multiple lines of code. A typical test harness will include many tests, all of which set up a known situation, execute some code, and then verify the results.

Consequences: Writing tests before writing code helps you focus on the code you need to write. Being able to test as you write usually speeds up the development process and results in higher quality code.

Related Pattern: The Test Harness pattern is a specialization of the Java Program pattern.

7.7.2 The `toString` Pattern

Name: `toString`

Context: You would like to be able to easily print information about an object, usually for debugging purposes.

Solution: Override the `toString` method in the `Object` class. The usual format lists the class name with values of relevant instance variables between brackets, as shown in the following template:

```
public String toString()
{  return "«className»[" +
          "«instanceVarName1»=" + this.«instanceVarName1» +
          ", «instanceVarName2»=" + this.«instanceVarName2»
+
          ...
          ", «instanceVarNameN»=" + this.«instanceVarNameN»
+
          "]";
}
```

Consequences: The `toString` method is called automatically when an object is concatenated with a string or passed to the `print` or `println` method in `System.out`, making it easy to use a textual representation of the object.

Related Patterns: This pattern is a specialization of the Query pattern.

7.7.3 The Enumeration Pattern

Name: Enumeration

Context: You would like to have a variable with a specific set of values such as `MALE` and `FEMALE` or the four compass directions.

Solution: Define an enumeration type listing the desired values. A template follows:

```
public enum «typeName»
{ «valueName1», «valueName2», ..., «valueNameN»
}
```

For example, a set of values identifying styles of jeans for a clothing store inventory system could be defined as follows:

```
public enum JeanStyle
{  CLASSIC, RELAXED, BOOT_CUT, LOW_RISE, STRAIGHT
}
```

Use the name of the enumeration to declare variables and return types, such as the following:

```
public class DenimJeans
{ private JeanStyle style;

  public DenimJeans(JeanStyle aStyle)
  { this.style = aStyle;
  }

  public JeanStyle getStyle()
  { return this.style;
  }
}
```

Consequences: Variables using an enumeration type are prevented from having any value other than those defined by the enumeration and the special value `null`, helping to avoid programming errors. Well-chosen names help make programs more understandable. Enumerations cannot be used with versions of Java prior to 1.5.

Related Pattern: Prior to Java 1.5, programmers often used the Named Constant pattern to define a set of values. The Enumeration pattern is a better choice.

7.7.4 The Assign a Unique ID Pattern

Name: Assign a Unique ID

Context: Each instance of a specified class needs a unique identifier. The class should not depend on something external to itself to establish and maintain the uniqueness of the identifiers.

Solution: Store the unique identifier in an instance variable. Use a class variable to maintain the next unique identifier to assign. For example:

```
public class «className»
{ privatefinal int «uniqueID»;
  private static int «nextUniqueID» = «firstID»;

  public «className»()
  { super();
    this.«uniqueID» = «className».«nextUniqueID»;
    «className».«nextUniqueID»++;
  }
}
```

Consequences: Unique identifiers are assigned to each instance of the class for each execution of the program. If objects are stored in a file and then read again (see Section 9.4), care must be taken to save and restore *«nextUniqueID»* appropriately.

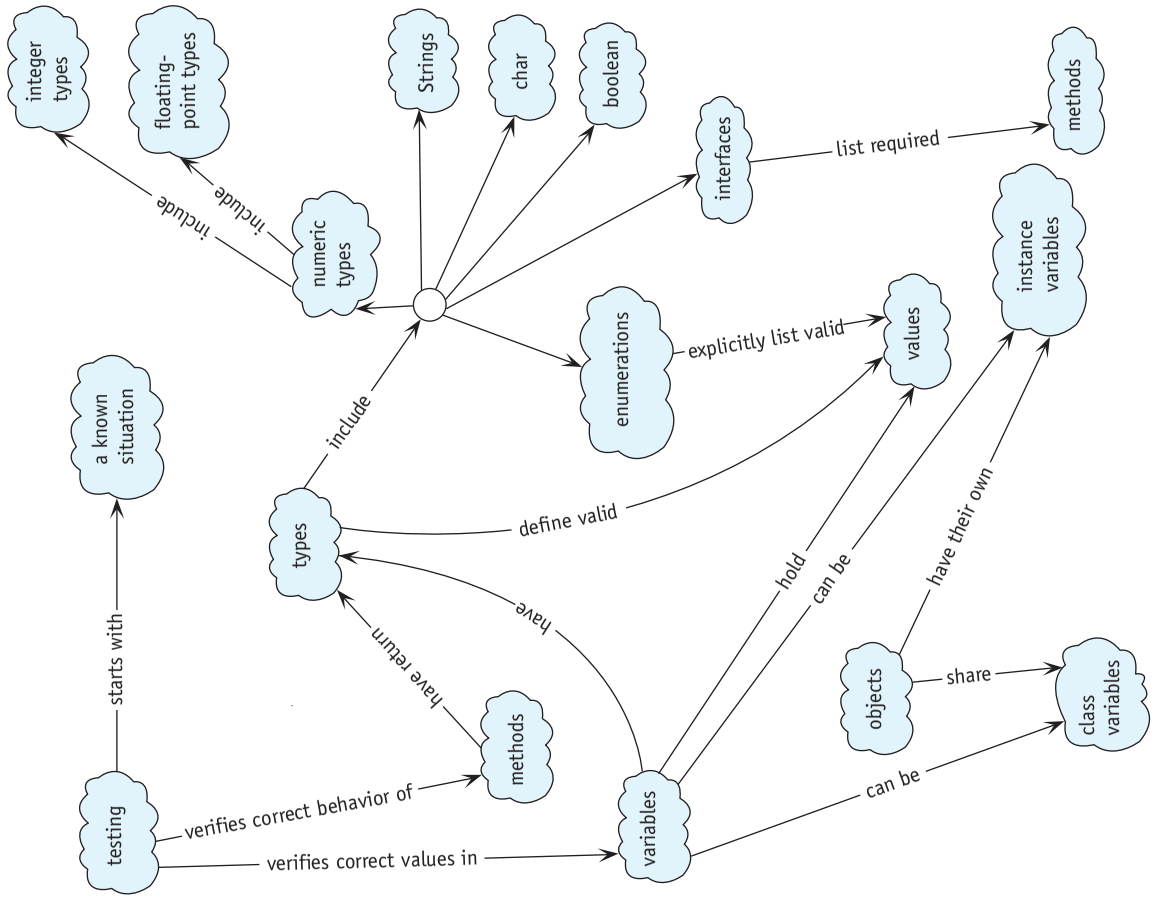
Related Patterns: This pattern makes use of the Instance Variable pattern.

7.8 Summary and Concept Map

Variables can have one of many types, such as `int`, `double`, `boolean`, and `String`. Incrementally testing variables and the methods that use them is a vital part of demonstrating that a program is correct. Developing tests before writing the code is a sound development practice that can help programmers develop correct code faster. Class variables and methods don't depend on an instance of a class for their operation.

Methods and variables provide the essential tools needed to write classes that address many different kinds of problems; the gas pump meter class is just one example.

Interfaces include a list of methods that implementing classes are required to define. Interfaces allow classes to work together in situations where it isn't possible or desirable to specify a class name. One example of this is when two programmers work on a project, one writing the graphical user interface and the other writing the model.


integer types
floating-point types
Strings
char
boolean
methods
include
include
numeric types
interfaces
list required
instance variables
enumerations
explicitly list valid
values
a known situation
include
define valid
types
have
have return
hold
can be
have their own
starts with
objects
share
class variables
methods
testing
verifies correct behavior of
verifies correct values in
variables
can be

7.9 Problem Set

Written Exercises

7.1 For each of the following situations, what would be the best choice(s) for the variable's type? Answer with one or more of `int`, `double`, `char`, `boolean`, `String`, or an enumeration defined by a programmer.

a. Store the current temperature.

b. Store the most recent key typed on the keyboard.

c. Store a compass heading such as "north" or "southeast."

d. Store the height of your best friend.

e. Pass the Dewey decimal number of a book to a method.

f. Store whether a recording is on cassette, CD, or a vinyl record.

g. Return the name of a company from a method.

h. Store the month of the year.

i. Store the number of books in your school's library.

j. Store the area of your room.

k. Store the title of your favorite novel.

l. Pass a person's admission category for the local museum (one of "Child," "Adult," or "Senior") to a method.

7.2 Place the following strings in increasing lexicographic order: `grated`, `grate!`, `grate`, `grateful`, `grate99`, `grace`, `grate[]`, `gratitude`, `grate(grace)`.

7.3 Recall that a `SimpleBot` must extend `Paintable` to guarantee to the compiler that it has a `paint` method. Could `Paintable` be an interface instead of a class? If so, explain what changes to `Paintable` and `SimpleBot` are required. If not, explain why.

7.4 Draw evaluation diagrams for the expressions `(double)3/4` and `(double)(3/4)`. Pay attention to the effects of precedence, automatic conversion, and integer division.

Programming Exercises

7.5 Run the following `main` method. Describe what happens. Based on what you know about the range of the type `byte`, why do you think this occurs?

```
public static void main(String[] args)
{ for (byte b = 0; b <= 128; b += 1)
  { System.out.println(b);
    Utilities.sleep(30);
  }
}
```

7.6 Write the following methods in the class `Name`. They all have a single `String` parameter and return a string. The argument is a full name such as "Frank Herbert," "Orson Scott Card," "Laura Elizabeth Ingalls Wilder," or "William Arthur Philip Louis Mountbatten-Windsor."

a. `firstName` returns the first name (e.g., "Laura").

b. `lastName` returns the last name (e.g., "Mountbatten-Windsor").

c. `initials` returns the first letter of each name (e.g., "WAPLM").

d. `shortName` returns the first initial and the last name (e.g., "O. Card").

e. `name` returns all of the initials except the last plus the last name (e.g., "L. E. I. Wilder").

7.7 Write a `main` method that outputs a multiplication table as shown on the left side of Figure 7-9. Then modify it to print a neater table as shown on the right side of the figure.

(figure 7-9)

Multiplication tables

```
1 2 3 4 5 6 7 8 9 10
2 4 6 8 10 12 14 16 18 20
3 6 9 12 15 18 21 24 27 30
4 8 12 16 20 24 28 32 36 40
5 10 15 20 25 30 35 40 45 50
6 12 18 24 30 36 42 48 54 60
7 14 21 28 35 42 49 56 63 70
8 16 24 32 40 48 56 64 72 80
9 18 27 36 45 54 63 72 81 90
10 20 30 40 50 60 70 80 90 100
```

```
 1   2   3   4   5   6   7   8   9  10
 2   4   6   8  10  12  14  16  18  20
 3   6   9  12  15  18  21  24  27  30
 4   8  12  16  20  24  28  32  36  40
 5  10  15  20  25  30  35  40  45  50
 6  12  18  24  30  36  42  48  54  60
 7  14  21  28  35  42  49  56  63  70
 8  16  24  32  40  48  56  64  72  80
 9  18  27  36  45  54  63  72  81  90
10  20  30  40  50  60  70  80  90 100
```

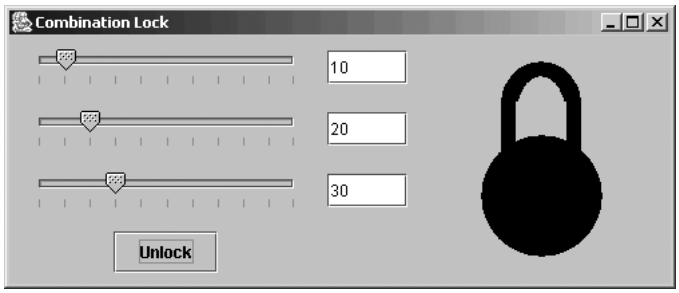
7.8 Rewrite the `SimpleBot` class to use an enumeration for the directions.

7.9 Write a class that implements `IMeter` but contains no methods whatsoever. Compile the class. What error message or messages does your compiler give you concerning the missing methods?

7.10 Write a class named `BustedBot` that extends `RobotSE`. A `BustedBot` is unreliable, occasionally turning either right or left (apparently at random) before moving. The probabilities of turning are given as two values (the probability of turning left and the probability of turning right) when the `BustedBot` is constructed. Write a `main` method that demonstrates your class.

Programming Projects

7.11 Explore the documentation for `becker.xtras.comboLock`. Write a class named `CombinationLock` that implements the `IComboLock` interface. Run it with the graphical user interface provided in `becker.xtras.comboLock.ComboLockGUI`. The result should be as shown in Figure 7-10. (*Hint*: This project is easier than the gas pump example.)



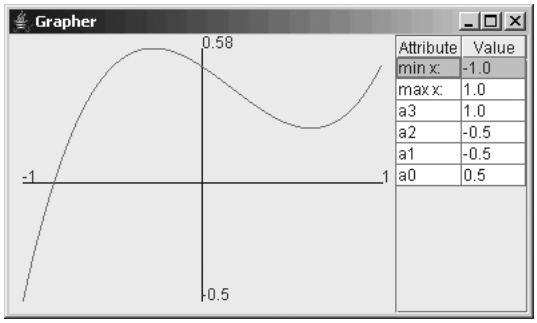
(figure 7-10)

Virtual combination lock

7.12 Implement a `Counter` class that could be used as part of the admission program for a carnival. A `Counter` object will keep track of how many people have entered the carnival so far. Each time a person enters, the `increment` method will be called. A query, `getCount`, will return the number of people who entered so far. A command, `reset`, will reset the counter back to zero to begin counting the next day. Write a main method to test your class.

7.13 Implement a class, `FuelUse`, to track the fuel use in an automobile. The `fillTank` method is called each time fuel is added to the automobile. It requires two arguments: the amount of fuel added and the distance driven since the last time the tank was filled. Provide two queries. One, `getMileage`, returns the miles per gallon or liters per 100 km (depending on your local convention) since record keeping began. The other query, `getTripMileage`, returns the miles per gallon or liters per 100 km since the most recent invocation of the command `resetTrip`. Return `-1` if mileage is requested when no miles have actually been traveled. Write a `main` method to test your class.

7.14 Explore the documentation for `becker.xtras.grapher`. The provided graphical user interface, `GrapherGUI`, will display the graph of a mathematical function when given a class that implements one of the interfaces `IFunction`, `IQuadraticFunction`, or `IPolynomialFunction` (see Figure 7-11).



(figure 7-11)

Graphing a mathematical function

a. Write a class named `FuncA` that extends `Object`, implements `IFunction`, and evaluates $\sin(x) + \cos(x)$.

b. Write a class named `FuncB` that extends `Object`, implements `QuadraticFunction`, and evaluates $ax^2 + bx + c$.

7.15 Write a class named `Time` that represents the time of day in hours and minutes. It should provide a constructor that can initialize the object to a specific time of day, and accessor methods `getHour`, `getMinute` as well as `toString`. Write four additional commands: `addHour()` and `addMinute()` to add a single hour and minute, respectively; and `addHours(int n)` and `addMinutes(int n)` to add the specified number of hours or minutes. Thoroughly test your class.

a. Write the `Time` class assuming a 24-hour clock. `toString` should return strings such as "00:15" and "23:09."

b. Write the `Time` class assuming a 12-hour clock—that is, `getHour` will always return a number between 0 and 12. Add an additional accessor method, `getPeriodDesignator`. The last accessor method returns a value for "AM" if the time is between midnight and noon and "PM" if the time is between noon and 1 minute before midnight. Use an enumerated type if you can; otherwise, use a `String`. `toString` should return values such as "00:15AM," "12:00PM" (noon), and "11:59PM" (1 minute to midnight).

7.16 Write a class named `Account`. Each `Account` object has an account owner such as "Suelyn Wang" and an account balance such as $349.12. Add an appropriate constructor and methods with the following signatures:

```
public int getBalance()
public String getOwner()
public void deposit(double howMuch)
public void withdraw(double howMuch)
public void payInterest(double rate)
```

The last method adds one month's interest by multiplying the `rate` divided by 12 times the current balance and adding the result to the current balance. Write a test harness.

7.17 Explore the documentation for `becker.xtras.radio`. Write two classes, one named `RadioTuner` that extends `Radio` and implements the `ITuner` interface, and another named `Main` that runs the program. The result should be similar to Figure 7-12. The graphical user interface will use `RadioTuner` to keep track of the current frequency, to search up and down for the next available frequency, and to remember up to five preset frequencies.

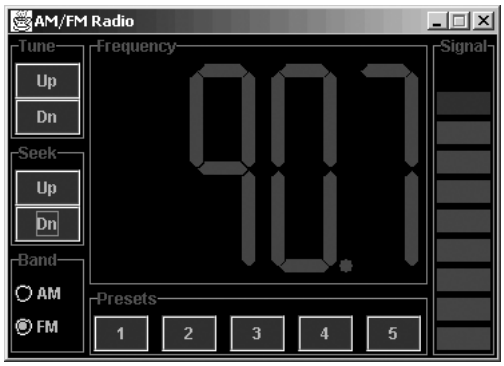
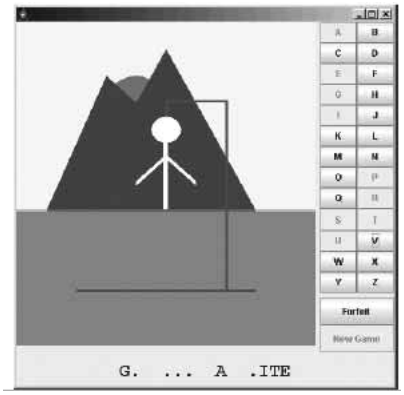


figure 7-12

Graphical user interface for an AM/FM radio

7.18 Explore the documentation for `becker.xtras.hangman`. Write two classes, one named `Hangman` that implements the `IHangman` interface and another named `HangmanMain` that includes a `main` method to run the program. The result should be similar to Figure 7-13. Your `Hangman` class will use a `String` to store the phrase the player is trying to guess and a second `String` to store the letters the player has guessed so far. You could use a `String` to store the phrase as the player has guessed it, but an instance of `StringBuffer` would be easier. `StringBuffer` is very similar to `String` but allows you to change individual characters.



(figure 7-13)

Graphical user interface for a game of Hangman