CHAPTER 4

INTRODUCTION TO CONTROL STATEMENTS

OBJECTIVES

Upon completion of this chapter, you should be able to:

- Use the increment and decrement operators.
- Use standard math methods.
- Use `if` and `if-else` statements to make choices.
- Use `while` and `for` loops to repeat a process.
- Construct appropriate conditions for control statements using relational operators.
- Detect and correct common errors involving loops.

Estimated Time: 3.5 hours

VOCABULARY

Control statements
Counter
Count-controlled loop
Entry-controlled loop
Flowchart
Infinite loop
Iteration
Off-by-one error
Overloading
Sentinel
Task-controlled loop

All the programs to this point have consisted of short sequences of instructions that are executed one after the other. Such a scheme, even if we allowed the sequence of instructions to become extremely long, would not be very useful. In computer programs, as in real life, instructions must express repetition and selection. Expressing these notions in Java is the major topic of this chapter, but before doing so we present a couple of topics that we use throughout the rest of the chapter.

4.1 Additional Operators

Strange to say, the operators presented in this section are completely unnecessary, and we could easily manage without them; however, Java programmers use them frequently, and we cannot ignore them. Fortunately, they are convenient and easy to use.

Extended Assignment Operators

The assignment operator can be combined with the arithmetic and concatenation operators to provide extended assignment operators. Following are several examples:

```
int a = 17;
String s = "hi";

a += 3;           // Equivalent to a = a + 3;
a -= 3;           // Equivalent to a = a - 3;
a *= 3;           // Equivalent to a = a * 3;
a /= 3;           // Equivalent to a = a / 3;
a %= 3;           // Equivalent to a = a % 3;
s += " there";    // Equivalent to s = s + " there";
```

All of these examples have the format

```
variable op= expression;
```

which is equivalent to

```
variable = variable op expression;
```

Note that there is no space between op and =. The extended assignment operators and the standard assignment operator have the same precedence. For more information about these additional operators, see Appendix C.

Increment and Decrement

Java includes increment (++) and decrement (--) operators that increase or decrease a variable's value by one:

```
int m = 7;
double x = 6.4;

m++;        // Equivalent to m = m + 1;
x--;        // Equivalent to x = x - 1.0;
```

Here and throughout the book we use these operators only in the manner just illustrated; however, they can also appear in the middle of expressions. The rules for doing so are tricky and involve complexities that lead to programming errors and confusion. We encourage you to restrict yourself to the simplest uses of these operators. The precedence of the increment and decrement operators is the same as unary plus, unary minus, and cast.

EXERCISE 4.1

1. Translate the following statements to equivalent statements that use extended assignment operators:

a. `x = x * 2;`

b. `y = y % 2;`

EXERCISE 4.1 Continued

2. Translate the following statements to equivalent statements that do not use the extended assignment operators:

   a. `x += 5;`

   b. `x *= x;`

4.2 Standard Classes and Methods

The standard Java library includes two classes that are frequently useful. These are the `Math` and the `Random` classes. The `Math` class provides a range of common mathematical methods, whereas the `Random` class supports programs that incorporate random numbers.

The `Math` Class

The `Math` class is quite extensive; however, we limit our attention to the methods listed in Table 4-1. Notice that two methods in the table are called `abs`. They are distinguished from each other by the fact that one takes an integer and the other takes a double parameter. Using the same name for two different methods is called *overloading*.

TABLE 4-1
Seven methods in the Math class

METHOD	WHAT IT DOES
`static int abs(int x)`	Returns the absolute value of an integer `x`
`static double abs(double x)`	Returns the absolute value of a double `x`
`static double pow(double base, double exponent)`	Returns the `base` raised to the `exponent`
`static long round(double x)`	Returns `x` rounded to the nearest whole number (Note: Returned value must be cast to an `int` before assignment to an `int` variable.)
`static int max(int a, int b)`	Returns the greater of `a` and `b`
`static int min(int a, int b)`	Returns the lesser of `a` and `b`
`static double sqrt(double x)`	Returns the square root of `x`

The `sqrt` Method

The next code segment illustrates the use of the `sqrt` method:

```
// Given the area of a circle, compute its radius.
// Use the formula a = pr², where a is the area and r is the radius.

double area = 10.0, radius;
radius = Math.sqrt(area / Math.PI);
```

To understand this code we must consider two points. First, messages are usually sent to objects; however, if a method's signature is labeled `static`, the message is sent to the method's class. Thus, to invoke the `sqrt` method, we send the `sqrt` message to the `Math` class. Second, in addition to methods, the `Math` class includes good approximations to several important constants. Here we use `Math.PI`, which is an approximation for π accurate to about 17 decimal places.

The Remaining Methods

The remaining methods described in Table 4-1 are illustrated in the following program code:

```
int m;
double x;

m = Math.abs(-7);              // m equals 7
x = Math.abs(-7.5);            // x equals 7.5

x = Math.pow(3.0, 2.0);        // x equals 3.0^2.0 equals 9.0
x = Math.pow(16.0, 0.25);      // x equals 16.0^0.25 equals 2.0

m = Math.max(20, 40);          // m equals 40
m = Math.min(20, 40);          // m equals 20
m = (int) Math.round(3.14);    // m equals 3
m = (int) Math.round(3.5);     // m equals 4
```

The methods `pow` and `sqrt` both expect parameters of type `double`. If an `int` is used instead, it is automatically converted to a double before the message is sent. The methods `max` and `min` also have versions that work with doubles. Other useful methods, including trigonometric methods, are described in Appendix B.

The Random Class

Programs are often used to simulate random events such as the flip of a coin, the arrival times of customers at a bank, the moment-to-moment fluctuations of the stock market, and so forth. At the heart of all such programs is a mechanism called a *random number generator* that returns numbers chosen at random from a predesignated interval. Java's random number generator is implemented in the `Random` class and utilizes the methods `nextInt`, and `nextDouble`, as described in Table 4-2.

TABLE 4-2
Methods in the Random class

METHOD	WHAT IT DOES
`int nextInt(int n)`	Returns an integer chosen at random from among 0, 1, 2, ..., $n-1$
`double nextDouble()`	Returns a double chosen at random between 0.0, inclusive, and 1.0, exclusive

A program that uses the `Random` class first must import `java.util.Random`. Following is a segment of code that illustrates the importing of `java.util.Random` and the use of the `nextInt` method:

```
import java.util.Random;
. . .

// Generate an integer chosen at random from among 0, 1, 2

Random generator = new Random();
System.out.print(generator.nextInt(3));
```

The output from this segment of code is different every time it is executed. Following are the results from three executions:

```
2
0
1
```

The method `nextDouble` behaves in a similar fashion but returns a `double` greater than or equal to 0.0 and less than 1.0.

EXERCISE 4.2

1. Assume that `x` has the value 3.6 and `y` has the value 4. State the value of the variable `z` after the following statements:
   - **a.** `z = Math.sqrt(y);`
   - **b.** `z = Math.round(x);`
   - **c.** `z = Math.pow(y, 3);`
   - **d.** `z = Math.round(Math.sqrt(x));`
2. Write code segments to print the following values in a terminal window:
   - **a.** A random integer between 1 and 20, inclusive
   - **b.** A random double between 1 and 10, inclusive

4.3 A Visit to the Farm

To introduce the main topic of this chapter, control statements, we begin with a "real world" example. Once upon a time in a faraway land, Jack visited his cousin Jill in the country and offered to milk the cow. Jill gave him a list of instructions:

```
fetch the cow from the field;
tie her in the stall;
milk her into the bucket;
pour the milk into the bottles;
drive her back into the field;
clean the bucket;
```

Although Jack was a little taken aback by Jill's liberal use of semicolons, he had no trouble following the instructions. A year later, Jack visited again. In the meantime, Jill had acquired a herd of cows, some red and some black. This time, when Jack offered to help, Jill gave him a more complex list of instructions:

```
herd the cows from the field into the west paddock;
while (there are any cows left in the west paddock){
   fetch a cow from the west paddock;
   tie her in the stall;
   if (she is red){
      milk her into the red bucket;
      pour the milk into red bottles;
  }else{
      milk her into the black bucket;
      pour the milk into black bottles;
  }
  put her into the east paddock;
}
herd the cows from the east paddock back into the field;
clean the buckets;
```

These instructions threw Jack for a loop (pun intended) until Jill explained that

```
while (some condition){
   do stuff;
}
```

means "do the stuff repeatedly as long as the condition holds true," and

```
if (some condition){
   do stuff 1;
}else{
   do stuff 2;
}
```

means "if some condition is true, do stuff 1, and if it is false, do stuff 2."

"And what about all the semicolons and braces?" asked Jack.

"Those," said Jill, "are just a habit I picked up from programming in Java, where `while` and `if-else` are called ***control statements***."

EXERCISE 4.3

1. Why does Jill use a `while` statement in her instructions to Jack?

2. Why does Jill use an `if-else` statement in her instructions to Jack? Write pseudocode control statements that are similar in style to the farm example for the following situations:
   - **a.** If a checker piece is red, then put it on a red square; otherwise, put it on a black square.
   - **b.** If your shoes are muddy, then take them off and leave them outside the door.
   - **c.** Pick up all the marbles on the floor and put them into a bag.

EXERCISE 4.3 Continued

3. Describe in English what the following code segments do:

a.

```
if (x is larger than y){
   temp = x;
   x = y;
   y = temp;
}else{
   temp = y;
   y = x;
   x = temp;
}
```

b.

```
sum = 0;
count = 1;
read an integer into total;
while (count is less than or equal to total){
   read an integer into x;
   sum = sum + Math.abs(x);
   count++;
}
if (total is greater than 0)
   print (sum / total)
```

4.4 *The* `if` *and* `if-else` *Statements*

We now explore in greater detail the `if-else` statement and the slightly simpler but related `if` statement. The meanings of `if` and `else` in Java sensibly adhere to our everyday usage of these words. Java and other third-generation programming languages achieve their programmer–friendly qualities by combining bits and pieces of English phrasing with some of the notational conventions of elementary algebra.

Principal Forms

To repeat, in Java, the `if` and `if-else` statements allow for the conditional execution of statements. For instance,

```
if (condition){
   statement;          //Execute these statements if the
   statement;          //condition is true.
}

if (condition){
   statement;          //Execute these statements if the
   statement;          //condition is true.
}else{
   statement;          //Execute these statements if the
   statement;          //condition is false.
}
```

The indicated semicolons and braces are required; however, the exact format of the text depends on the aesthetic sensibilities of the programmer, who should be guided by a desire to make the program as readable as possible. Notice that braces always occur in pairs and that no semicolon immediately follows a closing brace.

Additional Forms

The braces can be dropped if only a single statement follows the word `if` or `else`; for instance,

```
if (condition)
   statement;

if (condition)
   statement;
else
   statement;

if (condition){
   statement;
      ...
   statement;
}else
   statement;

if (condition)
   statement;
else{
   statement;
      ...
   statement;
}
```

Braces

In general, it is better to overuse braces than to under use them. Likewise, in expressions it is better to overuse parentheses. The extra braces or parentheses can never do any harm, and their presence helps to eliminate logic errors.

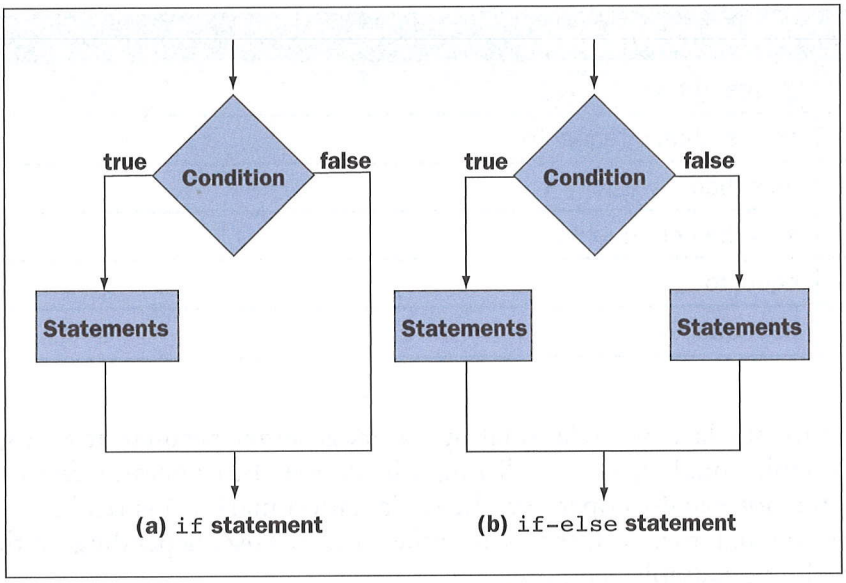
Boolean Expressions

The condition in an `if` statement must be a *Boolean expression*. This type of expression returns the value `true` or `false`.

Flowchart

Figure 4-1 shows a diagram called a *flowchart* that illustrates the behavior of `if` and `if-else` statements. When the statements are executed, either the left or the right branch is executed depending on whether the condition is `true` or `false`.

FIGURE 4-1
Flowcharts for the `if` and `if-else` statements



Examples

Following are some examples of `if` statements:

```
// Increase a salesman's commission by 10% if his sales are over $5000
if (sales > 5000)
   commission *= 1.1;

// Pay a worker $14.5 per hour plus time and a half for overtime
pay = hoursWorked * 14.5;
if (hoursWorked > 40){
   overtime = hoursWorked - 40;
   pay += overtime * 21.75;
}

// Let c equal the larger of a and b
if (a > b)
   c = a;
else
   c = b;
```

Relational Operators

The previous examples all use the relational operator for greater than (>); however, there are five other relational operators. Table 4-3 shows the complete list of relational operators available for use in Java.

TABLE 4-3
Relational operators

OPERATOR	WHAT IT MEANS
>	greater than
>=	greater than or equal to
<	less than
<=	less than or equal to
==	equal to
!=	not equal to

The notation for the last two relational operators is rather peculiar at first glance, but it is necessary. The double equal signs (==) distinguish the equal-to operator from the assignment operator (=). In the not-equal-to operator, the exclamation mark (!) is read as *not*. When these expressions are evaluated, their values will be either true or false, depending on the values of the operands involved. For example, suppose

```
a = 3              c = 10
b = 7              d = -20
```

then

```
a < b              is true
a <= b             is true
a == b             is false
a != b             is true
a - b > c + d      is true (the precedence of > is lower than + and -)
a < b < c          is invalid (syntactically incorrect)
a == b == c        is invalid
```

Checking Input for Validity

`if-else` statements are commonly used to check user inputs before processing them. For example, consider an admittedly trivial program that inputs the radius of a circle and outputs its area. If the user enters a negative number, the program should not go ahead and use it to compute an area that would be meaningless. Instead, the program should detect this problem and output an error message. Our next program example does this.

```
// Example 4.1: Computes the area of a circle if the
// radius >= 0, or displays an error message otherwise

import java.util.Scanner;

public class CircleArea{

   public static void main(String[] args){
      Scanner reader = new Scanner(System.in);
      System.out.print("Enter the radius: ");
      double radius = reader.nextDouble();
      if (radius < 0)
         System.out.println("Error: Radius must be >= 0");
      else{
         double area = Math.PI * Math.pow(radius, 2);
         System.out.println("The area is " + area);
      }
   }
}
```

EXERCISE 4.4

1. What type of expression must the condition of an `if` statement contain?

2. Describe the role of the curly braces ({}) in an `if` statement.

3. What is the difference between an `if` statement and an `if-else` statement?

4. Assume that `x` is 5 and `y` is 10. Write the values of the following expressions:

a. `x <= 10`

b. `x - 2 != 0`

c. `x > y`

5. Given the following mini-specifications, write expressions involving relational operators:

a. Determine if an input value `x` is greater than 0.

b. Determine if a given number of seconds equals a minute.

c. If `a`, `b`, and `c` are the lengths of the sides of a triangle and `c` is the largest side, determine if the triangle is a right triangle. (*Hint*: Use the Pythagorean equation and round the operand before comparing.)

6. Write the outputs of the following code segments:

a.

```
int x = 20, y = 15, z;

if (x < y)
   z = 10;
else
   z = 5;
System.out.println(z);
```

EXERCISE 4.4 Continued

b.

```
int x = 2;

if (Math.round(Math.sqrt(x)) == 1)
   System.out.println("Equal");
else
   System.out.println("Not equal");
```

7. Given the following mini-specifications, write expressions involving `if-else` statements and output statements:

a. Print the larger of two numbers.

b. Prompt the user for two whole numbers and input them. Then print the numbers in numeric order.

4.5 The `while` Statement

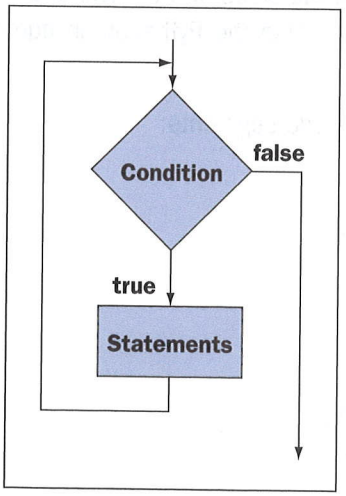
The `while` statement provides a looping mechanism that executes statements repeatedly for as long as some condition remains true. Following is the `while` statement's format:

```
while (condition)          // loop test
   statement;                 // one statement inside the loop body


while (condition){         // loop test
   statement;                 // many statements
   statement;                 // inside the
   ...                        // loop body
}
```

If the condition is false from the outset, the statement or statements inside the loop never execute. Figure 4-2 uses a flowchart to illustrate the behavior of a `while` statement.

FIGURE 4-2
Flowchart for a `while` statement



To help you become familiar with `while` statements, several short examples follow.

Compute 1 + 2 + ... + 100

The first example computes and displays the sum of the integers between 1 and 100, inclusive:

```
// Compute 1 + 2 + … + 100

int sum = 0, cntr = 1;
while (cntr <= 100){
   sum += cntr;      // point p (we refer to this location in Table 4-4)
   cntr++;           // point q (we refer to this location in Table 4-4)
}
System.out.println (sum);
```

The behavior of this portion of code is clear. The variable `cntr` acts as a counter that controls how many times the loop executes. The counter starts at 1. Each time around the loop, it is compared to 100 and incremented by 1. Clearly the code inside the loop is executed exactly 100 times, and each time through the loop, `sum` is incremented by increasing values of `cntr`.

Count-Controlled Loops

This is an example of what is called a ***count-controlled loop***. The variable `cntr` is called the ***counter***.

Tracing the Variables

To fully understand the loop, we must analyze the way in which the variables change on each pass or ***iteration*** through the loop. Table 4-4 helps in this endeavor. On the 100th iteration, `cntr` is increased to 101, so there is never a 101st iteration, and we are confident that the `sum` is computed correctly.

TABLE 4-4
Trace of how variables change on each iteration through a loop

ITERATION NUMBER	VALUE OF CNTR AT POINT P	VALUE OF SUM AT POINT P	VALUE OF CNTR AT POINT Q
1	1	1	2
2	2	1 + 2	3
...	...	...	...
100	100	1 + 2 +...+ 100	101

Adding Flexibility

In the next example, we again add a sequence of integers, but vary the counter's starting value, ending value, and increment:

```
// Display the sum of the integers between a startingValue
// and an endingValue, using a designated increment.

int cntr, sum, startingValue, endingValue, increment;

startingValue = 10;
endingValue = 100;
increment = 7;

sum = 0;
cntr = startingValue;
while (cntr <= endingValue){
   sum += cntr;
   cntr += increment;
}
System.out.println (sum);
```

This portion of code computes the value of 10 + 17 + 24 + ... + 94. For greater flexibility the code could be modified to ask the user for the starting value, the ending value, and the increment.

Counting Backward

We can also run the counter backward as in the next example, which displays the square roots of the numbers 25, 20, 15, and 10. Here the counter variable is called `number`:

```
// Display the square roots of 25, 20, 15, and 10

int number = 25;
while (number >= 10){
   System.out.println ("The square root of " + number +
                       " is " + Math.sqrt (number));
   number -= 5;
}
```

The output is

```
The square root of 25 is 5.0
The square root of 20 is 4.47213595499958
The square root of 15 is 3.872983346207417
The square root of 10 is 3.1622776601683795
```

Task-Controlled Loop

Sometimes loops are structured so that they continue to execute until some task is accomplished. These are called ***task-controlled loops***. To illustrate, we write code that finds the first integer for which the sum 1 + 2 + ... + n is over 1 million:

```
// Display the first value n for which 1 + 2 + . . . + n
// is greater than 1 million

int sum = 0;
int number = 0;
while (sum <= 1000000){
   number++;
   sum += number;          // point p
}

System.out.println (number);
```

To verify that the code works as intended, we can reason as follows:

- The first time we reach point p, number = 1, sum = 1, and sum <= 1,000,000.
- The second time we reach point p, number = 2, sum = 1 + 2, and sum <= 1,000,000.
- Etc. ...
- The last time we reach point p, number = n, sum = 1 + 2 + ... + n, and sum > 1,000,000.
- After that we will not enter the loop again, and number contains the first value to force the sum over a million.

Common Structure

All the preceding examples share a common structure:

```
initialize variables                                    // initialize
while (condition){                                         // test
   perform calculations and                                // loop body
   change variables involved in the condition
}
```

For the loop to terminate, each iteration through the loop must move the variables involved in the condition significantly closer to satisfying the condition.

Computing Factorial

The factorial of a given number n is the product of the numbers between 1 and n, inclusive (1 * 2 * 3 * . . . * n). The following program prompts the user for n and displays its factorial:

```
// Example 4.2: Compute and display the factorial of n

import java.util.Scanner;

public class Factorial{

   public static void main(String[] args){
      Scanner reader = new Scanner(System.in);
      System.out.print("Enter a number greater than 0: ");
      int number = reader.nextInt();
      int product = 1;
      int count = 1;
      while (count <= number){
         product = product * count;
         System.out.println(product);
         count++;
      }
      System.out.println("The factorial of " + number +
                         " is " + product);
   }
}
```

EXERCISE 4.5

1. When does a `while` loop terminate execution?
2. List the three components of a `while` loop.
3. What happens if the condition of a `while` loop is false from the outset?
4. Describe in English what the following code segments do:

a.

```
int expo = 1, limit = 10;

while (expo <= limit){
   System.out.println(expo + " " + Math.pow(2, expo));
expo++;
   }
```

b.

```
Scanner reader = new Scanner(System.in);
int product = 1;
System.out.print("Enter a positive number or -999 to halt");
int x = reader.nextInt();

while (x != -999){
   product *= x;
   System.out.print("Enter a positive number or -999 to halt");
   x = reader.nextInt();
}
```

5. Write code segments to perform the following tasks:
   a. Print the squares and cubes of the first 10 positive integers.
   b. Print 10 random integers between 1 and 10 inclusive.
   c. Input names and ages of people until a person's age is 100.

4.6 The `for` Statement

Count-controlled loops are used so frequently that many programming languages (including Java) include a special statement to make them easy to write. It is called *the `for` statement*, and it combines counter initialization, condition test, and update into a single expression. Following is its form:

```
for (initialize counter; test counter; update counter)
   statement;      // one statement inside the loop body

for (initialize counter; test counter; update counter){
   statement;     // many statements
   statement;     // inside the
   . . .;         // loop body
}
```

When the statement is executed, the counter is initialized. Then, as long as the test yields `true`, the statements in the loop body are executed, and the counter is updated. It is essential to understand that the counter is updated at the bottom of the loop, after the statements in the body have been executed. Even though the update appears at the top of the loop, it is executed at the bottom.

To demonstrate how the `for` statement works, we have rewritten the count-controlled loops presented in the previous sections:

```
// Compute 1 + 2 + ... + 100

int sum = 0, cntr;
for (cntr = 1; cntr <= 100; cntr++)
   sum += cntr;
System.out.println (sum);



// Display the sum of the integers between a startingValue
// and an endingValue, using a designated increment.

int cntr, sum, startingValue, endingValue, increment;

startingValue = 10;
endingValue = 100;
increment = 7;

sum = 0;
for (cntr = startingValue; cntr <= endingValue; cntr += increment)
   sum += cntr;
System.out.println (sum);



// Display the square roots of 25, 20, 15, and 10

int number;
for (number = 25; number >= 10; number -= 5)
   System.out.println ("The square root of " + number +
                       " is " + Math.sqrt (number));
```

Count-Controlled Input

Programs often need to read and process repeating inputs. For instance, consider a program that computes the average of a list of numbers. Here the repeating input is a single number. As each number is read into the program, it is added to a sum. When all the numbers have been read, the program computes and prints the average. For maximum flexibility the program must process a list of any length, but this requirement seemingly creates a dilemma. When we write the program, we have no way of knowing how many numbers will be in the list. So how can we write the program? There are two approaches. We illustrate one method in this section and the other in Section 4.8.

In the first method, we begin by asking the user for the length of the list and then we read exactly that many additional values:

```
Scanner reader = new Scanner(System.in);
double number, sum = 0;
int i, count;

System.out.print("How long is the list? ");
count = reader.nextInt();
for (i = 1; i <= count; i++){
   System.out.print("Enter a positive number: ");
   number = reader.nextDouble();
   sum += number;
}

if (count == 0)
   System.out.println ("You entered no numbers.");
else
   System.out.println ("The average is " + sum / count);
```

Following is a sample run:

```
How long is the list? 3
Enter a positive number: 1.1
Enter a positive number: 2.2
Enter a positive number: 3.3
The average is 2.1999999999999997
```

Declaring the Loop Control Variable in a `for` Loop

The loop control variables in the examples shown thus far have been declared outside of and above the loop structure. However, the `for` loop allows the programmer to declare the loop control variable inside of the loop header. Following are equivalent loops that show these two alternatives:

```
int i;                                  // Declare control variable above loop

for (i = 1; i <= 10; i++)
   System.out.println(i);



for (int i = 1; i <= 10; i++)           // Declare control variable in loop header
   System.out.println(i);
```

Although both loops are equivalent in function, the second alternative is considered preferable on most occasions for two reasons:

1. The loop control variable is visible only within the body of the loop where it is intended to be used.
2. The same name can be declared again in other `for` loops in the same program.

We discuss this important property of variable names, called *scope*, in more detail in Chapter 5.

Choosing a `while` Loop or a `for` Loop

Because `while` loops and `for` loops are quite similar, the following question arises: Which type of loop is more appropriate in a given situation? Both `for` loops and `while` loops are *entry-controlled loops*. This means that a continuation condition is tested at the top of the loop on each pass, before the loop's body of code is executed. If this condition is `true`, the loop continues. If it is `false`, the loop terminates. This also means that the loop's body of code might not execute at all (when its continuation condition is initially `false`). Thus, both types of loops can be used in exactly the same situations.

The choice between a `while` loop and a `for` loop is often a matter of programmer style or taste. However, some programmers argue that there are two minor advantages in using a `for` loop:

1. All of the loop control information, including the initial setting of the loop control variable, its update, and the test of the continuation condition, appears within the `for` loop's header. By contrast, only the test appears in the header of a `while` loop. Some programmers argue that you are more likely to forget the update in a `while` loop, which can cause a logic error.
2. The loop control variable of a `for` loop can be declared in its header. This restricts the variable's visibility to the body of the loop, where it is relevant. Although there are exceptions to this practice, restricting the visibility of variables in a program can make it more likely to be free of logic errors. We discuss this idea in more detail in Chapter 5.

EXERCISE 4.6

1. Describe in English what the following code segments do:

a.

```
for (int expo = 1; expo <= limit; expo++)
                System.out.println(expo + " " + Math.pow(2, expo));
```

b.

```
            int base = 2;

            for (int count = expo; count > 1; count--)
        base = base * base;
```

2. Write code segments that use `for` loops to perform the following tasks:

a. Print the squares and cubes of the first 10 positive integers.

b. Build a string consisting of the first 10 positive digits in descending order.

3. Translate the following `for` loops to equivalent `while` loops:

a.

```
Scanner reader new Scanner(System.in);

for (int i = 1; i <= 5; i++){
   System.out.print("Enter an integer: ");
   int number = reader.nextInt();
   System.out.println(Math.pow(number, 2));
}
```

EXERCISE 4.6 Continued

b.

```
int base = 2;

for (int count = expo; count > 1; count-)
   base = base * base;
```

4.7 Nested Control Statements and the `break` Statement

Control statements can be nested inside each other in any combination that proves useful. We now present several illustrative examples and also demonstrate a mechanism for breaking out of a loop early, that is, before the loop condition is false. All the examples use `for` loops, but similar examples can be constructed using `while` loops.

Print the Divisors

As a first example, we write a code segment that asks the user for a positive integer `n`, and then prints all its proper divisors, that is, all divisors except 1 and the number itself. For instance, the proper divisors of 12 are 2, 3, 4, and 6. A positive integer `d` is a divisor of `n` if `d` is less than `n` and `n % d` is zero. Thus, to find `n`'s proper divisors, we must try all values of `d` between 2 and `n / 2`. Here is the code:

```
// Display the proper divisors of a number

System.out.print("Enter a positive integer: ");
int n = reader.nextInt();

int limit = n / 2;

for (int d = 2; d <= limit; d++){
   if (n % d == 0)
      System.out.print (d + " ");
}
```

Is a Number Prime?

A number is prime if it has no proper divisors. We can modify the previous code segment to determine if a number is prime simply by counting its proper divisors. If there are none, the number is prime. Following is code that implements this plan:

```
// Determine if a number is prime

System.out.print(("Enter an integer greater than 2: ");
int n = reader.nextInt();

int count = 0;
int limit = n/2;
```

```
for (int d = 2; d <= limit; d++){
   if (n % d == 0)

      count++;
}

if (count != 0)
   System.out.println ("Not prime.");
else
   System.out.println ("Prime.");
```

The break Statement

Most programmers, including the authors of this text, enjoy the challenge of trying to write efficient programs. You can do two things to improve the efficiency of the previous segment of code. First, the limit does not need to be as large as `n / 2`. If `a*b` equals `n`, then either `a` or `b` must be less than or equal to the square root of `n`. Second, as soon as we find the first divisor of `n`, we know `n` is not prime, so there is no point in going around the loop again. To get out of a loop prematurely, that is, before the loop condition is `false`, we can use a `break` statement. A loop, either `for` or `while`, terminates immediately when a `break` statement is executed.

In the following segment of code, we check `d` after the `for` loop terminates. If `n` has a divisor, the `break` statement executes, the loop terminates early, and `d` is less than or equal to the `limit`. Following is the code:

```
// Determine if a number is prime

System.out.print("Enter an integer greater than 2: ");
int n = reader.nextInt();

int limit = (int)Math.sqrt (n);

int d;                                          // Declare control variable here

for (d = 2; d <= limit; d++){
   if (n % d == 0)
      break;
}

if (d <= limit)                                 // So it's visible here
   System.out.println ("Not prime.");
else
   System.out.println ("Prime.");
```

Note that the loop control variable `d` must now be declared above the loop so that it will be visible below it.

Sentinel-Controlled Input

In addition to the count-controlled input mentioned in the previous section, there is a second method for handling repeating user inputs. We again use the example of finding the average of a list of numbers. Now we read numbers repeatedly until we encounter a special value called a ***sentinel*** that

marks the end of the list. For instance, if all the numbers in the list are positive, then the sentinel could be –1, as shown in the following code:

```
Scanner reader = new Scanner(System.in);
double number, sum = 0;
int count = 0;

while (true){
   System.out.print("Enter a positive number or -1 to quit: ");
   number = reader.nextDouble();
   if (number == -1) break;
   sum += number;
   count++;
}

if (count == 0)
   System.out.println ("The list is empty.");
else
   System.out.println ("The average is " + sum / count);
```

Following is a sample run:

```
Enter a positive number or -1 to quit: 1.1
Enter a positive number or -1 to quit: 2.2
Enter a positive number or -1 to quit: 3.3
Enter a positive number or -1 to quit: -1
The average is 2.1999999999999997
```

Like any other high-level language, Java is rich in control statements. For example, the `switch` statement, the `do-while` statement, and the `continue` statement allow the programmer to express selection and repetition in a different manner than the control statements in this chapter. For details on these statements, see Appendix B.

EXERCISE 4.7

1. Describe in English what the following code segments do:

a.

```
for (int i = 1; i <= limit; i++)
   if (i % 2 == 0)
      System.out.println(i);
```

b.

```
Random gen = new Random();
int myNumber = gen.nextInt(10);
int x = 0;
int yourNumber;

while (x == 0){
   System.out.println("I'm guessing a number between 1 and 10.");
   System.out.print("Which number is it? ");
   yourNumber = reader.nextInt();
```

EXERCISE 4.7 Continued

```
        if (myNumber == yourNumber){
            System.out.println("That's it!");
            break;
        }else System.out.println("Sorry, try again");
    }
```

2. Write code segments that use loops to perform the following tasks:
   a. Print the squares and cubes of the first 10 positive, odd integers.
   b. Build a string consisting of the first 10 positive, even digits in descending order.

4.8 Using Loops with Text Files

Thus far in this book, we have seen examples of programs that have taken input data from human users at the keyboard. Most of these programs can receive their input data from text files as well. A text file is a software object that stores data on a permanent medium such as a disk, CD, or flash memory. When compared to keyboard input from a human user, the main advantages of taking input data from a file are as follows:

- The data set can be much larger.
- The data can be input much more quickly and with less chance of error.
- The data can be used repeatedly with the same program or with different programs.

Text Files and Their Format

The data in a text file can be created, saved, and viewed with a text editor, such as Notepad. Alternatively, a program can output the data to a file, a procedure that we discuss shortly. The data in a text file can be viewed as characters, words, numbers, or lines of text, depending on the text file's format and on the purposes for which the data is used. When the data are treated as numbers (either integers or floating-points), they must be separated by whitespace characters. These are the space, tab, and newline characters. For example, a text file containing six floating-point numbers might look like

```
34.6 22.33 66.75
77.12 21.44 99.01
```

when examined with a text editor.

Any text file, including the special case of an empty file, must contain at least one special character that marks the end of the file. This character can serve as a sentinel for a program loop that reads each datum until the end-of-file condition is encountered. When a program opens a file, an input pointer is set to the first character in it. Assuming that the end-of-file condition has not yet been reached, when each datum is read, the pointer moves to the next datum.

The Scanner and File Classes

Fortunately, we can use the same `Scanner` class for text file input that we use for keyboard input. The first step is to create a scanner object by opening it on a file object rather than the

keyboard object. File objects are instances of the class `File`, which appears in the package `java.io`. We obtain a file object by running the code `new File(aFileName)`, where `afileName` is a pathname or the name of a file in the current working directory. For example, let's assume that we have a text file of floating-point numbers named `"numbers.txt"`. The following code opens a scanner on this file:

```
Scanner reader = new Scanner(new File("numbers.txt"));
```

A sentinel-controlled loop can then be used to read all of the data from the file. Before each datum is read, we must check to see if the end of the file condition has been reached.The `Scanner` method `hasNext()` returns `true` if the end of file is not yet present. Now let's assume that we want to compute the average of the numbers in the file. A slight modification of the sentinel-controlled loop from section 4.7 accomplishes this:

```
Scanner reader = new Scanner(new File("numbers.txt"));
double number, sum = 0;
int count = 0;

while (reader.hasNext()){
   number = reader.nextDouble();
   sum += number;
   count++;
}
```

Note that we test for the sentinel before reading the data and we omit the output of the prompt to the user.

Files and Exceptions

When a Java program encounters an operation that it cannot perform at runtime, the JVM throws an exception. We have seen examples of exceptions thrown in earlier chapters when a program commits run-time errors, such as an attempt to divide an integer by zero. When working with files, a program may encounter errors that it cannot handle. For example, the file could be missing when the program tries to open it, or the file's data may be corrupted on the storage medium. In these cases, an I/O exception is thrown. I/O exceptions are generally so serious that they belong to a category called *checked exceptions*. This means that a program must at least acknowledge they might be thrown, if not do something to recover from them. We see how to recover from an exception in Chapter 10. For now, to meet the minimal requirement, we can place the simple phrase `throws IOException` after the header of the `main` method of any program that does text file input. If the program encounters an `IOException`, the JVM will halt execution with an error message.

The following program uses our sentinel-controlled loop to compute and display the average of the numbers in a text file:

```
// Example 4.3: Computes and displays the average of
// a file of floating-point numbers

import java.io.*;              // For File and IOException
import java.util.Scanner;
```

```
public class ComputeAverage{
  public static void main(String[] args) throws IOException {
     Scanner reader = new Scanner(new File("numbers.txt"));
     double number, sum = 0;
     int count = 0;

     while (reader.hasNext()){
        number = reader.nextDouble();
        sum += number;
        count++;
     }
     if (count == 0)
        System.out.println("The file had no numbers");
     else
        System.out.println("The average of " + count + " numbers is " +
                           sum / count);
     }
  }
```

Output to Text Files

Data can be output to a text file using the class `PrintWriter`. This class includes the methods `print` and `println` that you have already used for terminal output, so there is not much new to be learned. A `PrintWriter` is opened like scanner, using the form

```
PrintWriter writer = new PrintWriter(new File(aFileName));
```

After the outputs have been completed, you must close the `PrintWriter` with the statement

```
writer.close();
```

Failure to close the output file may result in no data being saved to it. Because an `IOException` might be thrown during these operations, the `main` method must use a `throws IOException` clause to keep the compiler happy.

Our next example program filters a set of integers by removing all of the zeroes and retaining the other values. The inputs are in a text file named `numbers.txt`, whereas the outputs go to a file named `newnumbers.txt`. Here is the code:

```
// Example 4.4: Inputs a text file of integers and writes these
// to an output file without the zeroes

import java.io.*;          // For File, IOException, and PrintWriter
import java.util.Scanner;

public class FilterZeroes{

   public static void main(String[] args) throws IOException {

      // Open the scanner and print writer
      Scanner reader = new Scanner(new File("numbers.txt"));
      PrintWriter writer = new PrintWriter(new File("newnumbers.txt"));
```

```
        // Read the numbers and write all but the zeroes
        while (reader.hasNext()){
            int number = reader.nextInt();
            if (number != 0)
                writer.println(number);
        }

        // Remember to close the output file
        writer.close();
    }
}
```

Note that this program also runs correctly with an empty input file or one that contains no zeroes.

EXERCISE 4.8

1. Describe in English what the following code segment does:

```
Scanner reader = new Scanner(new File("myfile.txt"));
```

2. Write code segments that use loops to perform the following tasks:
   a. Read integers from a file and display them in the terminal window.
   b. Assume that there are an even number of integers in a file. Read them in pairs from the file and display the larger value of each pair in the terminal window.

Case Study: The Folly of Gambling

It is said, with some justification, that only the mathematically challenged gamble. Lotteries, slot machines, and gambling games in general are designed to take in more money than they pay out. Even if gamblers get lucky and win a few times, in the long run they lose. For this case study we have invented a game of chance called Lucky Sevens that seems like an attractive proposition, but which is, as usual, a sure loser for the gambler. The rules of the game are simple:

- Roll a pair of dice.
- If the sum of the spots equals 7, the player wins $4; else the player loses $1.

To entice the gullible, the casino tells players that there are lots of ways to win: (1, 6), (2, 5), etc. A little mathematical analysis reveals that there are not enough ways to win to make the game worthwhile; however, many people's eyes glaze over at the first mention of mathematics, so the challenge is to write a program that demonstrates the futility of playing the game.

Request

Write a program that demonstrates the futility of playing Lucky Sevens.

Analysis

We use the random number generator to write a program that simulates the game. The program asks the user how many dollars he has, plays the game repeatedly until the money is gone, and displays the number of rolls taken. The program also displays the maximum amount of money held by the player, thus demonstrating that getting ahead at some point does not avoid the inevitable outcome. Figure 4-3 shows the proposed interface.

FIGURE 4-3
Interface for the lucky sevens simulator

```
How many dollars do you have? 100
You are broke after 543 rolls.
You should have quit after 47 rolls when you had $113
```

Design

The design is captured in the following pseudocode:

```
read the initial amount the gambler has to wager
initialize to zero a counter representing the number of rolls

the maximum amount equals the initial amount
the count at the maximum equals zero

while (there is any money left){
   increment the rolls counter
   roll the dice

   if (the dice add to seven)
      add $4 to the gambler's amount
   else
      subtract $1 from the gambler's amount

   if (the amount is now greater than ever before){
      remember this maximum amount
      remember the current value of the rolls counter
  }
}
display the rolls counter
display the maximum amount
display the count at the maximum amount
```

Implementation

Following is a program based on the pseudocode:

```
/*LuckySevens.java
Simulate the game of lucky sevens until all funds are depleted.
1) Rules:
        roll two dice
        if the sum equals 7, win $4, else lose $1
2) The inputs are:
        the amount of money the user is prepared to lose
3) Computations:
        use the random number generator to simulate rolling the dice
        loop until the funds are depleted
        count the number of rolls
        keep track of the maximum amount
4) The outputs are:
        the number of rolls it takes to deplete the funds
        the maximum amount
*/

import java.util.Scanner;
import java.util.Random;

public class LuckySevens {
   public static void main (String [] args) {

      Scanner reader = new Scanner(System.in);
      Random generator = new Random();

      int die1, die2,        // two dice
          dollars,           // initial number of dollars (input)
          count,             // number of rolls to reach depletion
          maxDollars,        // maximum amount held by the gambler
          countAtMax;        // count when the maximum is achieved

      // Request the input
      System.out.print("How many dollars do you have? ");
      dollars = reader.nextInt();

      // Initialize variables
      maxDollars = dollars;
      countAtMax = 0;
      count = 0;

      // Loop until the money is gone
      while (dollars > 0){
         count++;

         // Roll the dice.
         die1 = generator.nextInt (6) + 1; // 1-6
         die2 = generator.nextInt (6) + 1; // 1-6
```

```
            // Calculate the winnings or losses
            if (die1 + die2 == 7)
               dollars += 4;
            else
               dollars -= 1;

            // If this is a new maximum, remember it
            if (dollars > maxDollars){
               maxDollars = dollars;
               countAtMax = count;
            }
         }

         // Display the results
         System.out.println
            ("You are broke after " + count + " rolls.\n" +
             "You should have quit after " + countAtMax +
             " rolls when you had $" + maxDollars + ".");
      }
   }
```

Output

Running this program is just about as exciting (in our opinion) as going to Las Vegas, and it's a lot cheaper. (Perhaps we should translate it into a Java applet as shown in Chapter 8 of this book, make it part of a Web page, and charge people 10 cents each to run it.) Following are the results from several trial runs:

```
How many dollars do you have? 100
You are broke after 255 rolls.
You should have quit after 35 rolls when you had $110.

How many dollars do you have? 100
You are broke after 500 rolls.
You should have quit after 179 rolls when you had $136.

How many dollars do you have? 1000000
You are broke after 6029535 rolls.
You should have quit after 97 rolls when you had $1000008.
```

These results show that there is very little money to be gained in this game of chance—regardless of how much you have available to gamble.

4.9 Errors in Loops

We can easily make logic errors when coding loops, but we can avoid many of these errors if we have a proper understanding of a loop's typical structure. A loop usually has four component parts:

1. *Initializing statements*. These statements initialize variables used within the loop.
2. *Terminating condition*. This condition is tested before each pass through the loop to determine if another iteration is needed.
3. *Body statements*. These statements execute with each iteration and implement the calculation in question.

4. *Update statements*. These statements, which usually are executed at the bottom of the loop, change the values of the variables tested in the terminating condition.

A careless programmer can introduce logic errors into any one of these components. To demonstrate, we first present a simple but correct `while` loop and then show several revised versions, each with a different logic error. The correct version is

```
//Compute the product of the odd integers from 1 to 100
//Outcome - product will equal 3*5*...*99
int product = 1;
int i = 3;
while (i <= 100){
   product = product * i;
   i = i + 2;
}
System.out.println (product);
```

Initialization Error

We first introduce an error into the initializing statements. Because we forget to initialize the variable `product`, it retains its default value of zero.

```
//Error - failure to initialize the variable product
//Outcome - zero is printed
int product;
int i = 3;
while (i <= 100){
   product = product * i;
   i = i + 2;
}
System.out.println (product);
```

Off-by-One Error

The next error involves the terminating condition:

```
//Error - use of "< 99" rather than "<= 100" in the
//    terminating condition
//Outcome - product will equal 3*5...*97
int product = 1;
int i = 3;
while (i < 99){
   product = product * i;
   i = i + 2;
}
System.out.println (product);
```

This is called an ***off-by-one error***, and it occurs whenever a loop goes around one too many or one too few times. This is one of the most common types of looping errors and is often difficult to detect. Do not be fooled by the fact that, in this example, the error is glaringly obvious.

Infinite Loop

Following is another error in the terminating condition:

```
//Error - use of "!= 100" rather than "<= 100" in the terminating condition
//Outcome - the program will never stop
int product = 1;
int i = 3;

while (i != 100){
   product = product * i;
   i = i + 2;
}
System.out.println (product);
```

The variable `i` takes on the values 3, 5, ..., 99, 101, ... and never equals 100. This is called an *infinite loop*. Anytime a program responds more slowly than expected, it is reasonable to assume that it is stuck in an infinite loop. Do not pull the plug. Instead, on a PC, select the terminal window and press Ctrl+C; that is, press the Control and "C" keys simultaneously. This will stop the program.

Error in Loop Body

Following is an error in the body of the loop. Again, the error is comically obvious because we are pointing it out, but these kinds of errors often can be difficult to detect—particularly for the person who wrote the program.

```
//Error - use of + rather than * when computing product
//Outcome - product will equal 3+5+...+99
int product = 1;
int i = 3;
while (i <= 100){
   product = product + i;
   i = i + 2;
}
System.out.println (product);
```

Update Error

If the update statement is in the wrong place, the calculations can be thrown off even if the loop iterates the correct number of times:

```
//Error - placement of the update statement in the wrong place
//Outcome - product will equal 5*7*...*99*101
int product = 1;
int i = 3;
while (i <= 100){
   i = i + 2;          //this update statement should follow the calculation
   product = product * i;
}
System.out.println (product);
```

Effects of Limited Floating-Point Precision

Numbers that are declared as `double` have about 18 decimal digits of precision. This is very good, but it is not perfect and can lead to unexpected errors. Consider the following lines of code, which seem to be free of logic errors, yet produce an infinite loop:

```
double x;
for (x = 0.0; x != 1.0; x += 0.1)
   System.out.print (x + " ");
```

When this code runs, the expected output is

```
 0.0  0.1  0.2  0.3  0.4  0.5  0.6  0.7  0.8  0.9  1.0
```

However, the actual output is

```
0.0  0.1  0.2  0.30000000000000004  0.4  0.5  0.6  0.7  0.7999999999999999
0.8999999999999999  0.9999999999999999  1.0999999999999999  1.2  1.3
1.4000000000000001  1.5000000000000002  ... etc ...
```

To understand what went wrong, consider the decimal representation of ⅓. It is 0.33333... The 3s go on forever, and consequently no finite representation is exact. The same sort of thing happens when 1/5 is represented in binary as a `double`. Consequently, in the previous code, `x` never exactly equals 1.0 and the loop never terminates. To fix the code, we rewrite it as

```
double x, delta;
delta = 0.01;
for (x = 0.0; x <= 1.0 + delta; x += 0.1)
   System.out.print (x + " ");
```

The code now works correctly, provided `delta` is less than the increment 0.1. The new output is

```
0.0  0.1  0.2  0.30000000000000004  0.4  0.5  0.6  0.7  0.7999999999999999
0.8999999999999999  0.9999999999999999
```

Debugging Loops

If you suspect that you have written a loop that contains a logic error, inspect the code and make sure the following items are true:

- Variables are initialized correctly before entering the loop.
- The terminating condition stops the iterations when the test variables have reached the intended limit.
- The statements in the body are correct.
- The update statements are positioned correctly and modify the test variables in such a manner that they eventually pass the limits tested in the terminating condition.

In addition, when writing terminating conditions, it is usually safer to use one of the operators

```
<     <=     >     >=
```

than either of the operators

```
==     !=
```

as demonstrated earlier.

Also, if you cannot find an error by inspection, then use `System.out.println` statements to "dump" key variables to the terminal window. Good places for these statements are

- Immediately after the initialization statements
- Inside the loop at the top
- Inside the loop at the bottom

You will then discover that some of the variables have values different than expected, and this will provide clues that reveal the exact nature and location of the logic error.

EXERCISE 4.9

1. Describe the logic errors in the following loops:

a.

```
// Print the odd numbers between 1 and limit, inclusive
for (int i = 1; i < limit; i++)
   if (i % 2 == 1)
      System.out.println(i);
```

b.

```
// Print the first ten positive odd numbers
int number = 1;
while (number != 10)
   System.out.println(number);
   number += 2;
}
```

4.10 Graphics and GUIs: I/O Dialog Boxes and Loops

Our graphics and GUI programs in earlier chapters displayed images but were not interactive, in that they did not accept user input. In this section we explore the use of dialog boxes for I/O. We also examine the use of loops and selection statements in graphics and GUI-based programs.

Extra Challenge

This Graphics and GUIs section gives you the opportunity to explore concepts and programming techniques required to develop modern graphics applications and graphical user interfaces. This material is not required in order to proceed with the other chapters of the book.

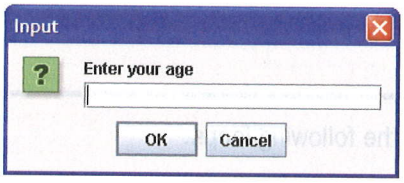
I/O Dialogs

A very convenient way to accept input from a user is to pop up an *input dialog box*. A typical input dialog box is a small window that contains a message asking for information, a text field for entering it, and command buttons labeled OK and Cancel. The text field can either be empty or contain a default data value, such as a number, when the dialog box pops up. The user places her mouse cursor in the text field and edits the data. She then clicks **OK** to accept the data or **Cancel** to back out of the interaction.

The class `JOptionPane` in the package `javax.swing` includes several versions of the method `showInputDialog` for input dialog boxes. One such version expects two strings as parameters. The first string is the message to be displayed in the dialog box. The second string is the default data value to be displayed in the input text field. If you want the field to be empty, you can pass the empty string (`""`) in this position. The following code segment pops up the input dialog box shown in Figure 4-4, which asks for the user's age:

```
String inputStr = JOptionPane.showInputDialog("Enter your age", "");
```

FIGURE 4-4
An input dialog



If the user clicks **OK**, the dialog box closes and returns the string that happens to be in the text input field. In our example, the code assigns this string to the variable `inputStr`. If the user clicks **Cancel** to back out, the dialog box closes and returns the value `null`, which our code also assigns to `inputStr`. Clearly, the program code must then check for `null` before passing the input datum on to the next step. The simplest way to handle the `null` value is shown in the next code segment:

```
if (inputStr == null)
   return;
```

The `return` statement in this code quits the method if the input string is `null`.

If the expected input is a number, one other thing must be done before it can be processed. The dialog box returns a string of digits, which must be converted to an `int` or a `double`. The methods `Integer.parseInt(aString)` and `Double.parseDouble(aString)` accomplish this for integers and floating-point numbers, respectively.

To output a message, we can use a *message dialog box*. The method `JOptionPane.showMessageDialog(anObserver, aString)` pops up a small window that displays the second parameter and waits for the user to click the OK button. For now, the first parameter is `null`.

The next program example incorporates dialog box I/O into the `CircleArea` program of Section 4.4. Shots of the dialog boxes from an interaction are shown in Figure 45.

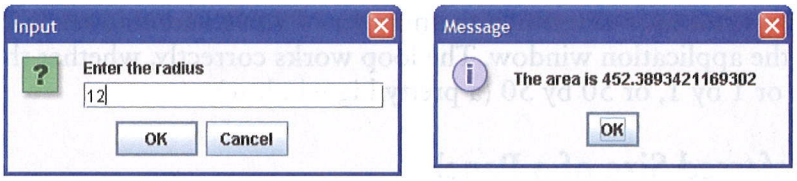
```
// Example 4.5: CircleARea with dialog I/O

import javax.swing.JOptionPane;

public class CircleArea{

   public static void main(String[] args){
      String inputStr = JOptionPane.showInputDialog("Enter the radius", "0");
      if (inputStr == null)
         return;
      double radius = Double.parseDouble(inputStr);
      if (radius < 0)
         JOptionPane.showMessageDialog(null, "Error: Radius must be >= 0");
      else{
         double area = Math.PI * Math.pow(radius, 2);
         JOptionPane.showMessageDialog(null, "The area is " + area);
      }
   }
}
```

FIGURE 4-5
Dialog I/O user interface for the circle area program



Unlike the graphics and GUI programs shown earlier, this one runs in the terminal window and does not pop up a main program window. Note also that if the user cancels the input dialog box, the program simply quits by returning from the `main` method.

Setting Up Lots of Panels

Some of our early graphics programs used multiple panels. Occasionally, many panels are called for. For example, a program might display a regular pattern such as those seen on quilts. In these situations, a loop can be used to initialize and install the panels. Consider the following fanciful program, which prompts the user for the dimensions of a grid of randomly colored panels and then displays them:

```
// Example 4.6: Display random colors in a grid
// whose dimensions are input by the user

import javax.swing.*;
import java.awt.*;
import java.util.Random;

public class GUIWindow{

   public static void main(String[] args){
      JFrame theGUI = new JFrame();
      theGUI.setTitle("GUI Example");
      String inputStr = JOptionPane.showInputDialog("Number of rows", "5");
```

```
        if (inputStr == null) return;
        int rows = Integer.parseInt(inputStr);
        inputStr = JOptionPane.showInputDialog("Number of columns", "5");
        if (inputStr == null) return;
        int cols = Integer.parseInt(inputStr);
        theGUI.setSize(cols  * 50, rows * 50);
        theGUI.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        Container pane = theGUI.getContentPane();
        pane.setLayout(new GridLayout(rows, cols));
        Random gen = new Random();
        for (int i = 1; i <= rows * cols; i++){
           int red = gen.nextInt(256);
           int green = gen.nextInt(256);
           int blue = gen.nextInt(256);
           Color backColor = new Color(red, green, blue);
           ColorPanel panel = new ColorPanel(backColor);
           pane.add(panel);
        }
        theGUI.setVisible(true);
     }
}
```

The user's inputs not only determine the number of rows and columns in the grid, but also the initial size of the application window. The loop works correctly, whether these values are the defaults (5 by 5), or 1 by 1, or 50 by 50 (a pretty big window).

Setting the Preferred Size of a Panel

In the graphics program we have seen thus far, the main window class sets its dimensions at program startup. The dimensions of any panels that appear within the main window are then adjusted to fit within the window when it is displayed. Another way to arrange things is to give each panel a preferred size and ask the window to shrink-wrap its dimensions to accommodate all of the panels. This alternative is useful when we want to fix the exact size of each panel before it is displayed. For instance, if we want the size of each panel to be exactly 50 by 50, the code in the previous example will not do because the window's title bar and borders occupy some of its "real estate."

To solve this problem, we drop the call of the method `setSize` from the main window class and add a call of the method `pack()`. This method asks the window to wrap itself around the minimal area necessary to display all of its components (panels or other widgets) at their preferred sizes. Here is the code for the `main` method of that program with the needed changes:

```
public static void main(String[] args){
   JFrame theGUI = new JFrame();
   theGUI.setTitle("GUI Example");
   String inputStr = JOptionPane.showInputDialog("Number of rows", "5");
   if (inputStr == null) return;
   int rows = Integer.parseInt(inputStr);
   inputStr = JOptionPane.showInputDialog("Number of columns", "5");
   if (inputStr == null) return;
   int cols = Integer.parseInt(inputStr);
   //theGUI.setSize(cols  * 50, rows * 50);   Dropped!!!
   theGUI.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
   Container pane = theGUI.getContentPane();
```

```
        pane.setLayout(new GridLayout(rows, cols));
        Random gen = new Random();
        for (int i = 1; i <= rows * cols; i++){
            int red = gen.nextInt(256);
            int green = gen.nextInt(256);
            int blue = gen.nextInt(256);
            Color backColor = new Color(red, green, blue);
            //Use new constructor to specify the preferred size of the panel
            ColorPanel panel = new ColorPanel(backColor, 50, 50);
            pane.add(panel);
        }
        theGUI.pack();                                    //Added!!
        theGUI.setVisible(true);
    }
```

If a panel does not set its own preferred size, its default size is 0 by 0. We add a second constructor to the `ColorPanel` class that receives a preferred width and height from the client. This constructor calls the method `setPreferredSize`, which expects an object of class `Dimension` as a parameter. This object, in turn, is created with the width and height received from the client. Here is the modified code for the `ColorPanel` class:

```
// Example 4.7: A color panel whose background is
// a color provided by the client
// A client-specified preferred size is optional

import javax.swing.*;
import java.awt.*;

public class ColorPanel extends JPanel{

    // Client provides color and preferred width and height
    public ColorPanel(Color backColor, int width, int height){
        setBackground(backColor);
        setPreferredSize(new Dimension(width, height));
    }

    // Client provides color
    // Preferred width and height are 0, 0 by default
    public ColorPanel(Color backColor){
        setBackground(backColor);
    }
}
```

Drawing Multiple Shapes

Consider the problem of displaying a bull's-eye, as shown in Figure 4-6. This particular bull's-eye is a pattern of five concentric filled ovals, whose colors alternate. Each oval is centered in the panel. The outermost oval's width and height are the width and height of the panel minus 1. Each oval's radius is a constant amount (called the thickness) larger than the next smaller oval's radius. One can create this pattern by drawing the largest oval first and then drawing the other ovals on top, layering them in descending order of size. The algorithm starts with a thickness of the panel's width divided by 10, an initial corner point of (0, 0), a size of the

panel's width – 1 and height – 1, and the color red. The algorithm then runs a loop that performs the following steps five times:

1. Draw the oval with the current color, corner point, and size.
2. Adjust the corner point by subtracting the thickness from each coordinate.
3. Adjust the size by subtracting twice the thickness from each dimension.
4. Adjust the color (to white if red, or to red otherwise).

FIGURE 4-6
A bull's eye

This algorithm is implemented in the `paintComponent` method of the following `ColorPanel` class:

```
// Example 4.8: A color panel containing
// a red and white bull's eye

import javax.swing.*;
import java.awt.*;

public class ColorPanel extends JPanel{

   // Client provides color and preferred width and height
   public ColorPanel(Color backColor, int width, int height){
      setBackground(backColor);
      setPreferredSize(new Dimension(width, height));
   }

   // Client provides color
   // Preferred width and height are 0, 0 by default
   public ColorPanel(Color backColor){
      setBackground(backColor);
   }

   public void paintComponent(Graphics g){
      super.paintComponent(g);

       //Set the attributes of the outermost oval
      int thickness = getWidth() / 10;
      int x = 0;
      int y = 0;
      int width = getWidth() - 1;
      int height = getHeight() - 1;
      Color ringColor = Color.red;
```