CHAPTER 6

Flow of Control, Part 2: Looping

CHAPTER CONTENTS

Introduction

Have you ever watched the cashier at the grocery store? Let's call the cashier Jane. Jane's job is to determine the total cost of a grocery purchase. To begin, Jane starts with a total cost of $0.00. She then reaches for the first item and scans it to record its price, which is added to the total. Then she reaches for the second item, scans that item to record its price, which is added to the total, and so on. Jane continues scanning each item, one at a time, until there are no more items to scan. Usually, the end of an order is signaled by a divider bar lying across the conveyor belt. When Jane sees the divider bar, she knows she is finished. At that point, she tells us the total cost of the order, collects the money, and gives us a receipt.

So we see that Jane's job consists of performing some preliminary work, processing each item one at a time, and reporting the result at the end.

In computing, we often perform tasks that follow this same pattern:

1: initialize values

2: process items one at a time

3: report results

The flow of control that programmers use to complete jobs with this pattern is called **looping**, or **repetition**.

6.1 Event-Controlled Loops Using *while*

If we attempt to write pseudocode for the grocery store cashier, we may start with something like this:

```
set total to $0.00
reach for first item
if item is not the divider bar
    add price to total
reach for next item
if item is not the divider bar
    add price to total
reach for next item
if item is not the divider bar
    add price to total
... (finally)
reach for next item
item is the divider bar,
    tell the customer the total price
```

We can see a pattern here. We start with an order total of $0.00. Then we repeat a set of operations for each item. We reach for the item and check whether it's the divider bar. If the item is not the divider bar, we add the item's price to the order total. We reach for the next item and check whether it's the divider bar, and so on. When we reach for the item and find that it is the divider bar, we know there are no more items to process, so the total we have at that time is the total for the whole order. In other words, we don't know the number of items that will be placed on the conveyor belt. We just process the order, item by item, until we see the divider bar, which we do not process.

In Java, the *while* loop is designed for repeating a set of instructions for each input value when we don't know at the beginning how many input values there will be. We simply process each input value, one at a time, until a signal—an event—tells us that there is no more input. This is called **event-controlled looping**. In the cashier's case, the signal for the end of input was the divider bar. In other tasks, the signal for the end of the input may be a special value that the user enters, called a **sentinel value**, or it may be that we've reached the end of an input file.

6.2 General Form for *while* Loops

The *while* loop has this syntax:

```
// initialize variables
while ( condition )
{
    // process data; loop body
}
// process the results
```

The condition is a *boolean* expression, that is, any expression that evaluates to *true* or *false*. When the *while* loop statement is encountered, the condition is evaluated; if the value is *true*, the statements in the **loop body** are executed. The condition is then reevaluated and, if *true*, the loop body is executed again. This repetition continues until the loop condition evaluates to *false*, at which time, the loop body is skipped and execution continues at the instruction following the loop body.

The curly braces are needed only if the loop body has more than one statement—that is, if more than one statement should be executed if the condition evaluates to *true*.

The scope of any variable defined within the *while* loop body extends from its declaration to the end of the *while* loop. Thus, any variable that is

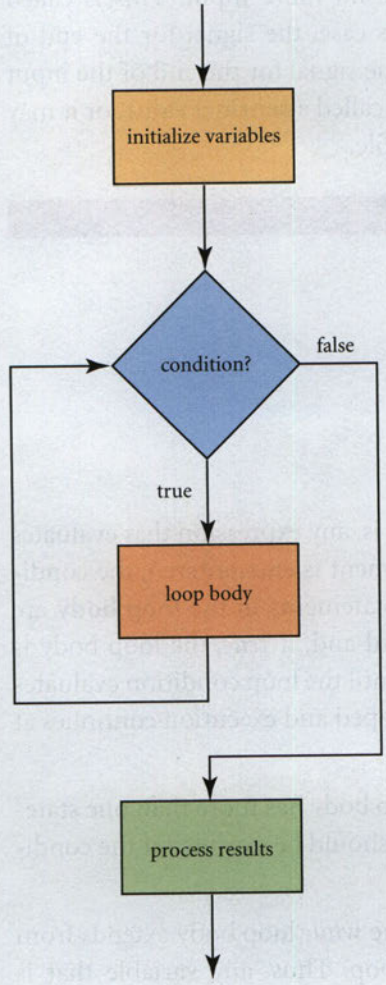
declared within a *while* loop body cannot be referenced after the *while* loop ends.

The flow of control of a *while* loop is shown in Figure 6.1.

Each execution of the loop body is called an **iteration** of the loop. Thus, if the loop body executes five times before the condition evaluates to *false*, we say there were five iterations of the *while* loop.

What happens if the loop condition is *false* the first time it is evaluated? Because the loop condition is evaluated before executing the *while* loop body, and the loop body is executed only if the condition is *true*, it is possible that the *while* loop body is never executed. In that case, there would be **zero iterations** of the loop.

Figure 6.1
Flow of Control of a *while* Loop



Using a *while* loop construct, the pseudocode for the cashier would look like this:

```
set total to $0.00
reach for first item
while item is not the divider bar
{
    add price to total
    reach for next item
}
// if we get here, the item is the divider bar
output the total price
```

It is also possible to construct a *while* loop whose condition *never* evaluates to *false*. That results in an **endless loop**, also known as an **infinite loop**. Because the condition always evaluates to *true*, the loop body is executed repeatedly, without end. This might happen if items other than the divider bar were placed continuously on the conveyor belt. One symptom of an endless loop is that the program doesn't terminate; it appears to "hang." However, if the program writes some output in the loop body, you will see that output spewing out on the Java console. Normally, the only recourse is for the user to abort the program.

The way to ensure that the condition will eventually evaluate to *false* is to include code, called a **loop update statement**, within the loop body that appropriately changes the variable that is being tested by the loop condition. If, for example, the loop condition tests for reading the sentinel value, the loop update statement should read the next input value.

One common logic error that causes an endless loop is putting a semicolon after the condition, as in the following:

```
while ( condition );  // semicolon causes endless loop if condition is true
```

A semicolon immediately following the condition indicates an empty loop body. Although some advanced programming techniques call for the use of an empty loop body, we will not be using those techniques in this book.

COMMON ERROR TRAP

Avoid putting a semicolon after the condition of a *while* loop. Doing so creates an empty loop body and could result in an endless loop.

6.3 Event-Controlled Looping

The *while* loop is used when we don't know how many times the loop will execute; that is, when the loop begins, we don't know how many iterations

of the loop will be required. We rely on a signal, or **event**, to tell us that we have processed all the data. For example, when the cashier begins checking out an order, she doesn't (necessarily) know how many items are in the grocery cart; she only knows to stop when she sees the divider bar on the conveyor belt. We call this an event-controlled loop because we continue processing data until an event occurs, which signals the end of the data.

When we're prompting the user to enter data from the console, and we don't know at the beginning of the loop how much data the user has to be processed, we can define a special value, called the sentinel value. The sentinel value can vary from task to task and is typically a value that is outside the normal range of data for that task.

Sometimes the data our program needs is in a text file. For example, a file could store a company's monthly sales for the last five years. We may want to calculate average monthly sales or perform other statistical computations on that data. In this case, we need to read our data from the file, instead of asking the user to enter the data from the keyboard. Typically, we use a file when a large amount of data is involved because it would be impractical for a user to enter the data manually.

Reading from a file is also an event-controlled loop because we don't know at the beginning of the program how much data is in the file. Thus, we need some way to determine when we have finished processing all the data in the file. Java, and other languages, provides some indicator that we have reached the end of the file. Thus, for input from a file, sensing the end-of-file indication is the event that signals that there is no more data to read.

6.3.1 Reading Data from the User

Let's look at the general form for using a *while* loop to process data entered from the user.

```
initialize variables
read the first data item  // priming read
while data item is not the sentinel value
{
   process the data

   read the next data item  // update read
}
report the results
```

After performing any initialization, we attempt to read the first item. We call this the **priming read** because, like priming a pump, we use that value to feed the condition of the *while* loop for the first iteration. If the first item is not the sentinel value, we process it. Processing may consist of calculating a total, counting the number of data items, comparing the data to previously read values, or any number of operations. Then we read the next data item. This is called the **update read** because we update the data item in preparation for feeding its value into the condition of the *while* loop for the next iteration. This processing, followed by an update read, continues until we do read the sentinel value, at which time we do not execute the *while* loop body. Instead, we skip to the first instruction following the *while* loop. Note that the sentinel value is not meant to be processed. Like the divider bar for the cashier, it is simply a signal to stop processing.

We illustrate this pattern in Example 6.1, which prompts the user for integers and echoes to the console whatever the user enters. We chose the sentinel value to be –1; that is, when the user enters a –1, we stop processing.

```
1 /*  Working with a sentinel value
2     Anderson, Franceschi
3 */
4 import java.util.Scanner;
5
6 public class EchoUserInput
7 {
8    public static void main( String [ ] args )
9    {
10      final int SENTINEL = -1;
11      int number;
12
13      Scanner scan = new Scanner( System.in );
14
15      // priming read
16      System.out.print( "Enter an integer, or -1 to stop > " );
17      number = scan.nextInt( );
18
19      while ( number != SENTINEL )
20      {
21          // processing
22          System.out.println( number );
23
24          // update read
25          System.out.print( "Enter an integer, or -1 to stop > " );
```

```
26            number = scan.nextInt( );
27        }
28
29        System.out.println( "Sentinel value detected. Goodbye" );
30    }
31 }
```

EXAMPLE 6.1 Echoing Input from the User

Figure 6.2 shows the output from this program when the user enters 23, 47, 100, and –1.

On line 10, we declare the sentinel value, –1, as a constant because the value of the sentinel will not change during the execution of the program, and it lets us clearly state via the *while* loop condition (line 19) that we want to execute the loop body only if the input is not the sentinel value.

Then on lines 16–17, we perform the priming read. The *while* loop condition on line 19 checks for the sentinel value. If the user enters the sentinel value first, we skip the *while* loop altogether and execute line 29, which prints a message that the sentinel value was entered, and we exit the program. If the user enters a number other than the sentinel value, we execute the body of the *while* loop (lines 21–26). In the *while* loop, we simply echo the user's input to the console, then perform the update read. Control then skips to the *while* loop condition, where the value the user entered in the update read is compared to the sentinel value. If this entry is the sentinel value, the loop is skipped; otherwise, the body of the loop is executed: The value is echoed, then a new value is read. This same processing continues until the user does enter the sentinel value.

Figure 6.2
Output from Example 6.1, Using a Sentinel Value

```
Enter an integer, or -1 to stop > 23
23
Enter an integer, or -1 to stop > 47
47
Enter an integer, or -1 to stop > 100
100
Enter an integer, or -1 to stop > -1
Sentinel value detected. Goodbye
```

A common error in constructing *while* loops is forgetting the update read. Without the update read, the *while* loop continually processes the same data item, leading to an endless loop.

COMMON ERROR TRAP

Omitting the update read may result in an endless loop.

Another common error is omitting the priming read and, instead, reading data inside the *while* loop before the processing, as in the following pseudocode:

```
initialize variables
while data item is not the sentinel value
{
   read the next data
   process the data
}
report the results
```

This structure has several problems. The first time we evaluate the *while* loop condition, we haven't read any data, so the result of that evaluation is unpredictable. Second, when we do read the sentinel value, we will process it, leading to incorrect results.

COMMON ERROR TRAP

Omitting the priming read leads to incorrect results.

6.3.2 Reading Data from a Text File

The *Scanner* class enables us to read data easily from a text file. Java also provides a whole set of classes in the *java.io* package to enable programmers to perform user input and output from a file.

For the *Scanner* class, the general form for reading data from a text file is a little different from reading the data from the user. First, instead of reading a value and checking whether it is the sentinel value, we check whether there is more data in the file, then read a value. Second, we don't need to print a prompt because the user doesn't enter the data; we just read the next value from the file. For the *Scanner* class, the pseudocode for reading from a text file is shown here:

```
initialize variables
while we have not reached end of file
```

```
{
  read the next data item
  process the data
}
report the results
```

Scanner class methods, including a constructor for reading from a text file, are shown in Table 6.1. Another class we will use is the *File* class, which associates a file name with a file. The constructor for the *File* class is shown in Table 6.2.

The constructor shown in Table 6.1 can be used to associate a *Scanner* object with a file. The *Scanner* object will tokenize the contents of the file and return the tokens as we call the *next* methods. The *hasNext* method in the *Scanner* class returns *true* if the input has another token, and *false* otherwise. Thus, when the *hasNext* method returns *false*, we know we have reached the end of the file.

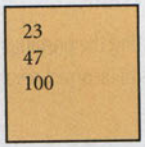


Figure 6.3
Contents of *input.txt*

Example 6.2 reads integers from a file named *input.txt* and echoes the integers to the console. The contents of *input.txt* are shown in Figure 6.3 and the output from the program is shown in Figure 6.4.

On line 14 of Example 6.2, we use the constructor of the *File* class to convert the file name, *input.txt*, to a platform-independent file name. Because we are specifying the simple file name, the JVM will look for the file in the same directory as our source file. If the file is located in another directory, we need to specify the path as well as the file name. For example, if the file were located on a flash drive in a Windows system, we would pass the *String* "*e:\\input.txt*" to the constructor. Notice that we need to use an escape sequence of two backslashes in order to specify the pathname, *e:\input.txt*.

The *File* class belongs to the *java.io* package, so we include an *import* statement for that class in line 5.

REFERENCE POINT

The *String* escape sequences are discussed in Chapter 2.

In line 15, we construct a *Scanner* object associated with the *inputFile* object. If the file is not found, the constructor generates a *FileNotFoundException*. It is also possible that an *IOException* may be generated if we encounter problems reading the file. Java requires us to acknowledge that these exceptions may be generated. One way to do that is to include the phrase *throws IOException* in the header for *main* (line 10). We also import the *IOException* class on line 6.

TABLE 6.1 Selected Methods of the *Scanner* Class

Selected Methods of the *Scanner* Class	
Constructor	
`Scanner( File file )`	
creates a *Scanner* object and associates it with a file	
Return value	**Method name and argument list**
`boolean`	`hasNext( )`
	returns *true* if there is another token in the input stream; *false*, otherwise
`byte`	`nextByte( )`
	returns the next input as a *byte*
`short`	`nextShort( )`
	returns the next input as a *short*
`int`	`nextInt( )`
	returns the next input as an *int*
`long`	`nextLong( )`
	returns the next input as a *long*
`float`	`nextFloat( )`
	returns the next input as a *float*
`double`	`nextDouble( )`
	returns the next input as a *double*
`boolean`	`nextBoolean( )`
	returns the next input as a *boolean*
`String`	`next( )`
	returns the next token in the input line as a *String*

TABLE 6.2 *File* Class Constructor

A Constructor for the *File* Class
`File( String pathname )`
constructs a *File* object with the *pathname* file name so that the file name is platform-independent

REFERENCE POINT

You can read more about the *Scanner* class on Oracle's Java website: *www.oracle.com/technetwork/java.*

On line 17, the first time our *while* loop condition is evaluated, we check whether there is any data in the file. If the file is empty, the *hasNext* method will return *false*, and we will skip execution of the loop body, continuing at line 25, where we print a message and exit the program.

The body of the *while* loop (lines 19–22) calls the *nextInt* method to read the next integer in the file and echoes that integer to the console. We then reevaluate the *while* loop condition (line 17) to determine if more data is in the file. When no more integers remain to be read, the *hasNext* method returns *false*, and we skip to line 25, where we print a message and exit the program.

Notice that we do not use a priming read because the *hasNext* method essentially peeks ahead into the file to see if there is more data. If the *hasNext* method returns *true*, we know that there is another integer to read, so we perform the read in the first line of the *while* loop body (line 20).

```
1 /* Reading a Text File
2    Anderson, Franceschi
3 */
4 import java.util.Scanner;
5 import java.io.File;
6 import java.io.IOException;
7
8 public class EchoFileData
9 {
10    public static void main( String [ ] args ) throws IOException
11    {
12       int number;
13
14       File inputFile = new File( "input.txt" );
15       Scanner scan = new Scanner( inputFile );
16
17       while ( scan.hasNext( ) )
18       {
19           // read next integer
20           number = scan.nextInt( );
21           // process the value read
22           System.out.println( number );
23       }
24
25       System.out.println( "End of file detected. Goodbye" );
26    }
27 }
```

EXAMPLE 6.2 Echoing Input from a File

```
23
47
100
End of file detected. Goodbye
```

Figure 6.4
Output from Example 6.2, Reading from a File

6.4 Looping Techniques

You will find that the *while* loop is an important tool for performing many common programming operations on a set of input values. For example, the *while* loop can be used to calculate the sum of values, count the number of values, find the average value, find the minimum and maximum values, animate an image, and perform other operations.

6.4.1 Accumulation

Let's look at a common programming operation for which a *while* loop is useful: calculating the sum of a set of values. To do this, we will build a simple calculator that performs one function: addition. We will prompt the user for numbers one at a time. We'll make the sentinel value a 0; that is, when the user wants to stop, the user will enter a 0. At that point, we will print the total.

The calculator can be developed using an event-controlled *while* loop and a standard computing technique: **accumulation**. In the accumulation operation, we initialize a *total* variable to 0. Each time we input a new value, we add that value to the *total*. When we reach the end of the input, the current value of *total* is the total for all the input.

Here is the pseudocode for the addition calculator:

```
set total to 0
read a number  // priming read
while the number is not the sentinel value
{
   add the number to total
   read the next number  // update read
}
output the total
```

Notice that this operation is almost identical to the grocery cashier's job in that we perform a priming read before the *while* loop. Inside the *while* loop, we process each number one at a time—adding each number to the total, then we read the next value, until we see the sentinel value, which is the signal to stop.

Example 6.3 provides the code for the addition calculator and Figure 6.5 shows the output for a sample execution of the calculator.

```
1 /*  Addition Calculator
2     Anderson, Franceschi
3 */
4
5 import java.util.Scanner;
6
7 public class Calculator
8 {
9    public static void main( String [ ] args )
10   {
11      final int SENTINEL = 0;
12      int number;
13      int total = 0;
14
15      Scanner scan = new Scanner( System.in );
16
17      System.out.println( "Welcome to the addition calculator.\n" );
18
19      System.out.print( "Enter the first number"
20                        + " or 0 for the total > " );
21      number = scan.nextInt( );
22
23      while ( number != SENTINEL )
24      {
25          total += number;
26
27          System.out.print( "Enter the next number"
28                            + " or 0 for the total > " );
29          number = scan.nextInt( );
30      }
31
32      System.out.println( "The total is " + total );
33   }
34 }
```

EXAMPLE 6.3 An Addition Calculator

```
Welcome to the addition calculator.

Enter the first number or 0 for the total > 34
Enter the next number or 0 for the total > -10
Enter the next number or 0 for the total > 2
Enter the next number or 0 for the total > 5
Enter the next number or 0 for the total > 8
Enter the next number or 0 for the total > 0
The total is 39
```

Figure 6.5
Output from a Sample Run of the Addition Calculator

Line 13 declares and initializes the *total* to *0*. This is an important step because the loop body will add each input value to the total. If the total is not set to 0 before the first input, we will get incorrect results. Furthermore, if *total* is declared but not initialized, our program will not compile.

COMMON ERROR TRAP

Forgetting to initialize the total to 0 will produce incorrect results.

Lines 19–21 read the first input value (the priming read). The *while* loop begins at line 23, and its condition checks for the sentinel value. The first time the *while* loop is encountered, this condition will check the value of the input from the priming read.

The loop body processes the input (line 25), which consists of adding the input value to the *total*. The final step in the loop body (lines 27–29) is to read the next input (the update read).

When the end of the loop body is reached, control is transferred back to line 23, where the loop condition is again tested with the input value read on line 29. If the condition is *true*, that is, if the input just read is not the sentinel value, then the loop body is reexecuted and the condition is retested, continuing until the input *is* the sentinel value, which causes the condition to evaluate to *false*. At that time, the loop body is skipped and line 32 is executed, which reports the results by printing the *total*.

SOFTWARE ENGINEERING TIP

Indent the body of a *while* loop to clearly illustrate the logic of the program.

Notice that the body of the *while* loop is indented and that the opening and closing curly braces are aligned in the same column as the *w* in the *while*. This style lets you easily see which statements belong to the *while* loop body.

COMMON ERROR TRAP

Choosing the wrong sentinel value may result in logic errors.

It is important to choose the sentinel value carefully. Obviously, the sentinel value cannot be a value that the user might want to be processed. In the addition calculator, we want to allow the user to enter positive or negative integers. We chose 0 as the sentinel value for two reasons. First, adding 0 to a total has no effect, so it is unlikely that the user will want to enter that value to be processed. Second, to the user, it is logical to enter a 0 to signal that there are no more integers to be added.

CODE IN ACTION

To see a step-by-step illustration of a *while* loop with a sentinel value, look for the Flash movie in the Chapter 6 folder on the CD-ROM included with this book. Click on the link for Chapter 6 to start the movie.

6.4.2 Counting Items

Counting is used when we need to know how many items are input or how many input values fit some criterion, for example, how many items are positive numbers or how many items are odd numbers. Counting is similar to accumulation in that we start with a count of 0 and increment (add 1 to) the count every time we read a value that meets the criterion. When there are no more values to read, the count variable contains the number of items that meet our criterion.

For example, let's count the number of students who passed a test. The pseudocode for this operation is as follows:

```
set countPassed to 0
read a test score
while the test score is not the sentinel value
{
  if the test score >= 60
  {
    add 1 to countPassed
  }
  read the next test score
}
output countPassed
```

The application in Example 6.4 counts the number of students that passed a test. We also calculate the percentage of the class that passed the test. To do this, we maintain a second count: the number of scores entered. This value will be incremented each time we read a score, whereas the *countPassed* value will be incremented only if the score is greater than or equal to 60. The sentinel value is –1. A sample run of this program is shown in Figure 6.6.

```
1 /* Counting passing test scores
2    Anderson, Franceschi
3 */
4
5 import java.util.Scanner;
6 import java.text.DecimalFormat;
7
8 public class CountTestScores
9 {
10    public static void main( String [ ] args )
11    {
12      int countPassed = 0;
13      int countScores = 0;
14      int score;
15      final int SENTINEL = -1;
16
17      Scanner scan = new Scanner( System.in );
18
19      System.out.println( "This program counts "
20                  + "the number of passing test scores." );
21      System.out.println( "Enter a -1 to stop." );
22
23      System.out.print( "Enter the first score > " );
24      score = scan.nextInt( );
25
26      while ( score != SENTINEL )
27      {
28          if ( score >= 60 )
29          {
30                countPassed++;
31          }
32
33          countScores++;
34
35          System.out.print( "Enter the next score > " );
36          score = scan.nextInt( );
37      }
38
```

```
39      System.out.println( "You entered " + countScores + " scores" );
40      System.out.println( "The number of passing test scores is "
41                                  + countPassed );
42      if ( countScores != 0 )
43      {
44         DecimalFormat percent = new DecimalFormat( "#0.0%" );
45         System.out.println(
46           percent.format( (double) ( countPassed ) / countScores )
47           + " of the class passed the test." );
48      }
49   }
50 }
```

EXAMPLE 6.4 Counting Passing Test Scores

Figure 6.6
Counting Passing Test Scores

```
This program counts the number of passing test scores.
Enter a -1 to stop.
Enter the first score > 98
Enter the next score > 75
Enter the next score > 60
Enter the next score > 59
Enter the next score > 45
Enter the next score > 88
Enter the next score > 94
Enter the next score > 96
Enter the next score > 56
Enter the next score > 77
Enter the next score > 82
Enter the next score > 89
Enter the next score > 100
Enter the next score > 78
Enter the next score > 95
Enter the next score > -1
You entered 15 scores
The number of passing test scores is 12
80.0% of the class passed the test.
```

COMMON ERROR TRAP

Forgetting to initialize the count variables will produce a compiler error.

Lines 12 and 13 declare the variables *countPassed* and *countScores* and initialize both to 0. Initializing these values to 0 is critical; otherwise, we will get the wrong results or a compiler error. We initialize these values to 0 because at that point, we have not yet processed any test scores.

Our *while* loop framework follows the familiar pattern. We perform the priming read for the first input (lines 23–24); our *while* loop condition

checks for the sentinel value (line 26); and the last statements of the *while* loop (lines 35–36) read the next value.

In the processing portion of the *while* loop, line 28 checks if the score just read is a passing score, and if so, line 30 adds 1 to *countPassed*. For each score entered, regardless of whether the student passed, we increment *countScores* (line 33).

When the sentinel value is entered, the *while* loop condition evaluates to *false* and control skips to line 39, where we output the number of scores entered and the number of passing scores. So that we avoid dividing by 0, note that line 42 checks whether no scores were entered. Note also that in line 46 we type cast *countPassed* to a *double* to force floating-point division, rather than integer division, so that the fractional part of the quotient will be maintained.

6.4.3 Calculating an Average

Calculating an average is a combination of accumulation and counting. We use accumulation to calculate the total and we use counting to count the number of items to average.

Here's the pseudocode for calculating an average:

```
set total to 0
set count to 0
read a number
while the number is not the sentinel value
{
   add the number to total
   add 1 to the count

   read the next number
}
set the average to total / count
output the average
```

Thus, to calculate an average test score for the class, we need to calculate the total of all the test scores, then divide by the number of students who took the test.

```
average = total / count;
```

It's important to remember that if we declare *total* and *count* as integers, then the *average* will be calculated using integer division, which truncates the remainder. To get a floating-point average, we need to type cast one of

the variables (either *total* or *count*) to a *double* or a *float* to force the division to be performed as floating-point.

```
double average = (double) ( total ) / count;
```

The application in Example 6.5 calculates an average test score for a class of students. The output is shown in Figure 6.7.

```
1 /* Calculate the average test score
2    Anderson, Franceschi
3 */
4
5 import java.util.Scanner;
6 import java.text.DecimalFormat;
7
8 public class AverageTestScore
9 {
10   public static void main( String [ ] args )
11   {
12     int count = 0;
13     int total = 0;
14     final int SENTINEL = -1;
15     int score;
16
17     Scanner scan = new Scanner( System.in );
18
19     System.out.println( "To calculate a class average," );
20     System.out.println( "enter each test score." );
21     System.out.println( "When you are finished, enter a -1" );
22
23     System.out.print( "Enter the first test score > " );
24     score = scan.nextInt( );
25
26     while ( score != SENTINEL )
27     {
28        total += score;   // add score to total
29        count++;          // add 1 to count of test scores
30
31        System.out.print( "Enter the next test score > " );
32        score = scan.nextInt( );
33     }
34
35     if ( count != 0 )
36     {
37        DecimalFormat oneDecimalPlace = new DecimalFormat( "##.0" );
38        System.out.println( "\nThe class average is "
39          + oneDecimalPlace.format( (double) ( total ) / count ) );
40     }
```

```
41     else
42         System.out.println( "\nNo grades were entered" );
44   }
45 }
```

EXAMPLE 6.5 Calculating an Average Test Score

```
To calculate a class average,
enter each test score.
When you are finished, enter a -1
Enter the first test score > 88
Enter the next test score > 78
Enter the next test score > 96
Enter the next test score > 75
Enter the next test score > 99
Enter the next test score > 56
Enter the next test score > 78
Enter the next test score > 84
Enter the next test score > 93
Enter the next test score > 79
Enter the next test score > 90
Enter the next test score > 85
Enter the next test score > 79
Enter the next test score > 92
Enter the next test score > 99
Enter the next test score > 94
Enter the next test score > -1

The class average is 85.3
```

Figure 6.7
Calculating the Average Test Score

In Example 6.5, lines 12 and 13 declare both *count* and *total* variables as *ints* and initialize each to 0. Again, our *while* loop structure follows the same pattern. Lines 23–24 read the first input value; the *while* loop condition (line 26) checks for the sentinel value; and the last statements in the *while* loop (lines 31–32) read the next score. For the processing portion of the *while* loop, we add the score to the total and increment the count of scores (lines 28–29). When the sentinel value is entered, we stop executing the *while* loop and skip to line 35.

In line 35, we avoid dividing by 0 by checking whether *count* is 0 (that is, if no scores were entered) before performing the division. If *count* is 0, we

COMMON ERROR TRAP

Forgetting to check whether the denominator is 0 before performing division is a logic error.

REFERENCE POINT

The difference between floating-point division and integer division is explained in Chapter 2.

simply print a message saying that no grades were entered. If *count* is not 0, we calculate and print the average. We first instantiate a *DecimalFormat* object (line 37) so that we can output the average to one decimal place. Remember that we need to type cast the *total* to a *double* (lines 38–39) to force floating-point division, rather than integer division.

6.4.4 Finding Maximum or Minimum Values

In Chapter 5, we illustrated a method for finding the maximum or minimum of three numbers. But that method won't work when we don't know how many numbers will be input. To find the maximum or minimum of an unknown number of input values, we need another approach.

In previous examples, we calculated a total for a group of numbers by keeping a running total. We started with a total of 0, then added each new input value to the running total. Similarly, we counted the number of input items by keeping a running count. We started with a count of 0 and incremented the count each time we read a new value. We can apply that same logic to calculating a maximum or minimum. For example, to find the maximum of a group of values, we can keep a "running," or current, maximum. We start by assuming that the first value we read is the maximum. In fact, it is the largest value we have seen so far. Then as we read each new value, we compare it to our current maximum. If the new value is greater, we make the new value our current maximum. When we come to the end of the input values, the current maximum is the maximum for all the input values.

Finding the minimum value, of course, uses the same approach, except that we replace the current minimum only if the new value is less than the current minimum.

Here's the pseudocode for finding a maximum value in a file:

```
read a first number and make it the maximum
while there is another number to read
{
   read the next number
   if number > maximum
```

```
  {
    set maximum to number
  }
}
output the maximum
```

Example 6.6 shows the code to find a maximum test grade in a file. As shown in Figure 6.8, the grades are stored as integers, one per line, in the file *grades.txt*. When this program runs, its output is shown in Figure 6.9.

```
88
78
96
75
99
56
78
84
93
79
90
85
79
90
85
79
92
99
94
```

Figure 6.8
The Contents of *grades.txt*

```
1  /* Find the maximum test grade
2     Anderson, Franceschi
3  */
4
5  import java.util.Scanner;
6  import java.io.*;
7
8  public class FindMaximumGrade
9  {
10    public static void main( String [ ] args ) throws IOException
11    {
12       int maxGrade;
13       int grade;
14
15       Scanner scan = new Scanner( new File( "grades.txt" ) );
16
17       System.out.println( "This program finds the maximum grade "
18                           + "for a class" );
19
20       if ( ! scan.hasNext( ) )
21       {
22           System.out.println( "No test grades are in the file" );
23       }
24       else
25       {
26          maxGrade = scan.nextInt( );  // make first grade the max
27
28          while ( scan.hasNext( ) )
29          {
30             grade = scan.nextInt( );  // read next grade
```

```
31
32              if ( grade > maxGrade )
33                 maxGrade = grade;     // save as current max
34           }
35
36           System.out.println( "The maximum grade is " + maxGrade );
37        }
38     }
39  }
```

EXAMPLE 6.6 Finding the Maximum Value

Figure 6.9
Finding the Maximum Value

```
This program finds the maximum grade for a class
The maximum grade is 99
```

In line 20, we call the *hasNext* method to test whether the file is empty. If so, we print a message (line 22) and the program ends. If, however, the file is not empty, we read the first value and automatically make it our maximum by storing the grade in *maxGrade* (line 26). In line 28, our *while* loop condition tests whether we have reached end of file. If not, we execute the body of the *while* loop (lines 30–33). We read the next grade and check whether that grade is greater than the current maximum. If so, we assign that grade to *maxGrade*; otherwise, we leave *maxGrade* unchanged. Then control is transferred to line 28 to retest the *while* loop condition.

When we do reach end of file, the *while* loop condition becomes *false*; control is transferred to line 36, and we output *maxGrade* as the maximum value.

COMMON ERROR TRAP

Initializing a maximum or a minimum to an arbitrary value, such as 0 or 100, is a logic error and could result in incorrect results.

A common error is to initialize the maximum or minimum to an arbitrary value, such as 0 or 100. This will not work for all conditions, however. For example, let's say we are finding the maximum number and we initialize the maximum to 0. If the user enters all negative numbers, then when the end of data is encountered, the maximum will still be 0, which is clearly an error. The same principle is true when finding a minimum value. If we initialize the minimum to 0, and the user enters all positive numbers greater

than 0, then at the end of our loop, our minimum value will still be 0, which is also incorrect.

Skill Practice
with these end-of-chapter questions

6.14.1 Multiple Choice Exercises

Question 1

6.14.2 Reading and Understanding Code

Questions 5, 6, 7, 8, 20

6.14.3 Fill In the Code

Questions 21, 22, 23, 24, 25, 26

6.14.4 Identifying Errors in Code

Questions 30, 31

6.14.5 Debugging Area

Question 37

6.14.6 Write a Short Program

Questions 44, 45

6.14.8 Technical Writing

Questions 70, 71

6.4.5 Animation

Animation is another operation that can be performed using *while* loops. For example, to move an object across a graphics window, we change the x or y values and draw the object in a new position in each iteration of the loop. We stop moving the object when we reach the edges of the window. Therefore, the sentinel value for the loop is that the x or y coordinate has reached the edge of the window.

If we want to roll a ball from left to right along an imaginary line, we can represent the ball by a filled circle. We start with an *x* value of 0 and some *y* value; our *while* loop draws the object at the current *x*, *y* position, then increments the *x* value.

Thus, to test for the sentinel value—that is, whether the ball has reached the right edge of the window—we add the diameter of the ball to the current *x* position of the ball and compare that result to the *x* coordinate of the right edge of the window.

For this animation, we'll use a *Circle* class written by the authors. Table 6.3 shows the constructors and methods of the *Circle* class.

Example 6.7 shows a *while* loop that simulates rolling a ball from the left edge of the window to the right edge.

```
1 /* RollABall, Version 1
2    Anderson, Franceschi
3 */
4
5 import java.awt.Graphics;
6 import java.awt.Color;
7 import javax.swing.JApplet;
8
9 public class RollABall1 extends JApplet
10 {
11    public void paint( Graphics g )
12    {
13       super.paint( g );
14
15       final int X = 0;          // the x value of the ball
16       final int Y = 50;         // the y value of the ball
17       final int DIAMETER = 15;  // the diameter of the ball
18       final Color COLOR = Color.BLUE; // the color of the ball
19       final int SPACER = 5;     // space between balls
20
21       // instantiate the ball as a Circle object
22       Circle ball = new Circle( X, Y, DIAMETER, COLOR );
23
24       // get ball diameter
25       int ballDiameter = ball.getDiameter( );
26       int sentinel = getWidth( ); // edge of the window is sentinel
27
```

TABLE 6.3 The *Circle* Class API

The *Circle* Class API

Constructors

`Circle( )`

constructs a *Circle* object with default values; *x* and *y* are set to 0, diameter to 10, and color to black

```
Circle( int startX, int startY, int sDiameter,
            Color circleColor )
```

constructs a *Circle* object; sets *x* and *y* to *startX* and *startY*, respectively; diameter to *sDiameter*; and color to *circleColor*

Return value	Method name and argument list
`int`	`getX( )` returns the ball's current *x* value
`int`	`getY( )` returns the ball's current *y* value
`int`	`getDiameter( )` returns the current diameter of the circle
`Color`	`getColor( )` returns the current color of the circle
`void`	`setX( int newX )` sets the *x* value to *newX*
`void`	`setY( int newY )` sets the *y* value to *newY*
`void`	`setDiameter( int newDiameter )` sets the diameter to *newDiameter*
`void`	`setColor( Color newColor )` sets the circle color to *newColor*
`void`	`draw( Graphics g )` draws a filled circle with *x* and *y* being the upper-left corner of a bounding rectangle, with the diameter and color set in the object

```
28          while ( ball.getX( ) + ballDiameter < sentinel )
29          {
30             ball.draw( g );    // draw the ball
31
32             // set x to next drawing location
33             ball.setX( ball.getX( ) + ballDiameter + SPACER );
34          }
35       }
36    }
```

EXAMPLE 6.7 Roll a Ball, Version 1

In the *paint* method, we instantiate the ball (line 22). It will start with an x value of 0 (the left edge of the screen), y value of 50, diameter of 15, and a blue color. We use a *while* loop (lines 28–34) to repeatedly draw a ball and increment the x value. The sentinel value occurs when the x value of the next ball (x + *ballDiameter*) reaches the right edge of the applet window. We determine the value of the right edge of the window by calling the *getWidth* method of the *JApplet* class (line 26), which returns the width of the applet window.

We chose to increment the x value by the width of the ball (the diameter) plus 5 pixels so that each ball is 5 pixels apart (line 33). Let's take a closer look at that statement:

```
ball.setX( ball.getX( ) + ballDiameter + SPACER );
```

We call the *setX* method of the *Circle* class to set the x value of the next ball. In order to set the x value to a new value, we need to get the current x value of the *ball* object and the current diameter. We do that by calling the *getX* method and adding the *ballDiameter* value we got earlier from the *getDiameter* method (line 25). Thus, by adding the diameter of the ball to the current x value, we calculate the x value of the right side of the ball. Adding the constant *SPACER* to that result puts a space of 5 pixels between the last ball drawn and the next.

Figure 6.10 shows the output when the applet is run.

This is fine for a first effort, but it isn't the effect we want. The result is just a series of balls drawn from left to right. There's another problem with *RollABall1*, which you can appreciate only if you run the applet: all

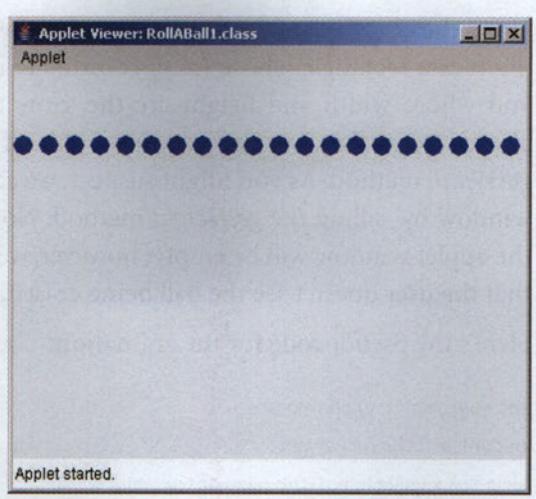


Figure 6.10
Roll a Ball, Version 1

the balls appear at once. There's no visual effect of the ball moving from left to right.

To get a rolling movement, we need to slow down the execution of the applet. To do that, we can use the *Pause* class provided by the authors and available in the directory on the CD-ROM containing this code. The *wait* method of the *Pause* class has the following API:

```
static void wait( double seconds )
```

Because the *wait* method is *static*, we invoke it using the *Pause* class name. For example, the following statement will pause the applet for approximately 3/100th of a second:

```
Pause.wait( .03 );
```

Also, we want to see only one ball at any time, and that should be the ball at the current (*x*, *y*) coordinate. To get this effect, we need to "erase" the previous ball before we draw the next ball in the new location. To erase the ball, we have two options: we can redraw the ball in the background color or we can clear the whole window by calling the *clearRect* method, which draws a rectangle in the background color. The default background color for the *JApplet* class is not one of the *Color* constants (*Color.BLUE*, etc.), so it's a difficult color to match. That being the case, we opt for the *clearRect* method, which has the following API:

```
void clearRect( int x, int y, int width, int height )
```

Using the *clearRect* method, we can erase the whole applet window by treating it as a rectangle whose (x, y) coordinate is the upper-left corner $(0, 0)$ and whose width and height are the same as the applet window. We've already seen that we can get the width of the window by calling the *getWidth* method. As you might suspect, we can get the height of the applet window by calling the *getHeight* method. Note that after we call *clearRect*, the applet window will be empty; however, we draw the next ball so quickly that the user doesn't see the ball being erased.

Here's the pseudocode for the animation:

```
set starting (x, y) coordinate
instantiate the ball object
while the x value is not the edge of the window
{
  draw the ball
  pause
  erase the ball
  set (x, y) coordinate to next drawing position
}
```

Example 6.8 shows the revised version of the rolling ball.

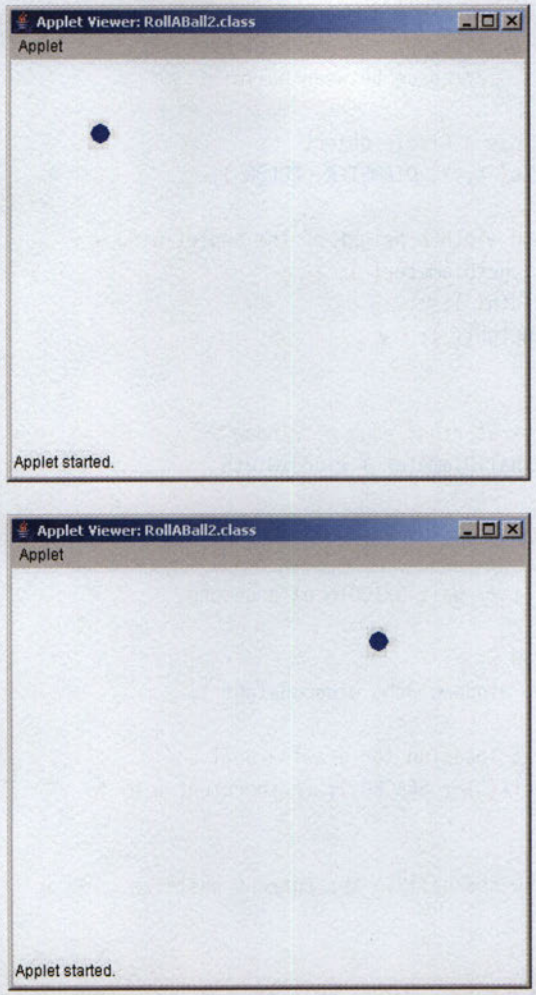
```
1 /* RollABall, version 2
2    Anderson, Franceschi
3 */
4
5 import java.awt.Graphics;
6 import java.awt.Color;
7 import javax.swing.JApplet;
8 import javax.swing.JOptionPane;
9
10 public class RollABall2 extends JApplet
11 {
12   public void paint( Graphics g )
13   {
14      super.paint( g );
15
16      final int X = 0;          // the x value of the ball
17      final int Y = 50;         // the y value of the ball
```

```
18      final int DIAMETER = 15;  // the diameter of the ball
19      final Color COLOR = Color.BLUE; // the color of the ball
20      final int SPACER = 5;     // space between balls
21
22      // instantiate the ball as a Circle object
23      Circle ball = new Circle( X, Y, DIAMETER, COLOR );
24
25      // get ball diameter and width & height of the applet window
26      int ballDiameter = ball.getDiameter( );
27      int windowWidth = getWidth( );
28      int windowHeight = getHeight( );
29
30      // rolling horizontally
31      // check whether ball is at right edge of window
32      while ( ball.getX( ) + ballDiameter < windowWidth )
33      {
34          ball.draw( g );  // draw the ball
35
36          Pause.wait( 0.03 ); // wait 3/100th of a second
37
38          // clear the window
39          g.clearRect( 0, 0, windowWidth, windowHeight );
40
41          // position to next location for drawing ball
42          ball.setX( ball.getX( ) + SPACER ); // increment x by 5
43      }
44
45      ball.draw( g );  // draw the ball in the current position
46  }
47 }
```

EXAMPLE 6.8 Roll a Ball, Version 2

In the *paint* method, we now draw a ball (line 34), pause for approximately 3/100ths of a second (line 36), then use the *clearRect* method to erase the window (line 39) before positioning *x* to the next location for drawing the ball (line 42). Now, only one ball is visible at any time, and the ball appears to roll across the screen, as shown in Figure 6.11. After the *while* loop completes, we draw the ball one more time (line 45) because when the *while* loop ends, the last ball drawn will have been erased.

Figure 6.11
Roll a Ball, Version 2



6.5 Type-Safe Input Using *Scanner*

One problem with reading input using *Scanner* is that if the next token does not match the data type we expect, an *InputMismatchException* is generated, which stops execution of the program. This could be caused by a simple typo on the user's part; for example, the user may type a letter or other nonnumeric character when our program prompts for an integer. To illustrate this problem, Example 6.9 shows a small program that prompts the user for an integer and calls the *nextInt* method of the *Scanner* class to read the integer, and Figure 6.12 shows the *InputMismatchException* generated when the user enters an *a* instead of an integer. Notice that the pro-

gram ends when the exception is generated; we never execute line 15, which echoes the age to the console.

```
1 /* Reading an integer from the user
2    Anderson, Franceschi
3 */
4 import java.util.Scanner;
5
6 public class ReadInteger
7 {
8    public static void main( String [ ] args )
9    {
10         Scanner scan = new Scanner( System.in );
11
12         System.out.print( "Enter your age as an integer > " );
13         int age = scan.nextInt( );
14
15         System.out.println( "Your age is " + age );
16   }
17 }
```

EXAMPLE 6.9 Reading an Integer

```
Enter your age as an integer > a
Exception in thread "main" java.util.InputMismatchException
        at java.util.Scanner.throwFor(Scanner.java:899)
        at java.util.Scanner.next(Scanner.java:1520)
        at java.util.Scanner.nextInt(Scanner.java:2150)
        at java.util.Scanner.nextInt(Scanner.java:2109)
        at ReadInteger.main(ReadInteger.java:13)
```

Figure 6.12
Input Failure

We can make our program more robust by checking, before we read, that the next token matches our expected input. The *Scanner* class provides *hasNext* methods for doing this, which are shown in Table 6.4. The *hasNext* methods return *true* if the next token can be read as the data type specified. For example, if we expect an integer, we can test whether the user has typed characters that can be interpreted as an integer by calling the *hasNextInt* method. If that method returns *true*, it is safe to read the value using the *nextInt* method. If the next token is not what we need, that is, if the *hasNextInt* method returns *false*, then reading that value as an *int* will

TABLE 6.4 *Scanner* Methods for Testing Tokens

Selected Input Stream Testing Methods of the *Scanner* Class	
Return value	**Method name and argument list**
`boolean`	`hasNext( )` returns *true* if there is another token in the input stream; *false*, otherwise
`boolean`	`hasNextByte( )` returns *true* if the token in the input stream can be read as a *byte*; *false*, otherwise
`boolean`	`hasNextShort( )` returns *true* if the token in the input stream can be read as a *short*; *false*, otherwise
`boolean`	`hasNextInt( )` returns *true* if the token in the input stream can be read as an *int*; *false*, otherwise
`boolean`	`hasNextLong( )` returns *true* if the token in the input stream can be read as a *long*; *false*, otherwise
`boolean`	`hasNextFloat( )` returns *true* if the token in the input stream can be read as a *float*; *false*, otherwise
`boolean`	`hasNextDouble( )` returns *true* if the token in the input stream can be read as a *double*; *false*, otherwise
`boolean`	`hasNextBoolean( )` returns *true* if the token in the input stream can be read as a *boolean*; *false*, otherwise
`String`	`nextLine( )` returns the remainder of the input line as a *String*

generate the *InputMismatchException*. In that case, we need to notify the user that the value typed is not valid and reprompt for new input. But first we need to clear the invalid input. We can flush the invalid input by calling the *nextLine* method of the *Scanner* class, which returns any remaining tokens on the input line as a *String*. Then we just ignore that *String*. Example 6.10 shows a revised version of Example 6.9 that is type-safe, meaning we guarantee we have an integer to read before reading it.

On line 14 of Example 6.10, we prompt for the integer. Then on line 15, the *while* loop condition checks whether the user has, indeed, typed an integer value. If not, we ignore whatever the user did type by calling the *nextLine* method (line 17). On line 18, we reprompt the user. The *while* loop continues executing until the user does enter an integer and the *hasNextInt* method returns *true*. At that point, we execute line 20, which reads the integer into the *age* variable. Figure 6.13 shows the output of this program when the user enters data other than integers, then finally enters an integer.

```
1  /* Type-Safe Input Using Scanner
2     Anderson, Franceschi
3  */
4
5  import java.util.Scanner;
6
7  public class TypeSafeReadInteger
8  {
9     public static void main( String [ ] args )
10    {
11       Scanner scan = new Scanner( System.in );
12       String garbage;
13
14       System.out.print( "Enter your age as an integer > " );
15       while ( ! scan.hasNextInt( ) )
16       {
17             garbage = scan.nextLine( );
18             System.out.print( "\nPlease enter an integer > " );
19       }
20       int age = scan.nextInt( );
21       System.out.println( "Your age is " + age );
22    }
23 }
```

EXAMPLE 6.10 Type-Safe Input

Figure 6.13
Reprompting Until the User Enters an Integer

```
Enter your age as an integer > asd

Please enter an integer > 12wg

Please enter an integer > 12.4

Please enter an integer > 23
Your age is 23
```

6.6 Constructing Loop Conditions

Constructing the correct loop condition may seem a little counterintuitive. The loop executes as long as the loop condition evaluates to *true*. Thus, if we want our loop to terminate when we read the sentinel value, then the loop condition should check that the input value is *not* the sentinel value. In other words, the loop continuation condition is the inverse of the loop termination condition. For a simple sentinel-controlled loop, the condition normally follows this pattern:

```
while ( inputValue != sentinel )
```

In fact, you can see that the loop conditions in many of the examples in this chapter use this form of *while* loop condition. Examples 6.7 and 6.8 use a similar pattern. We want to roll the ball as long as the ball is completely within the window. The loop termination condition is that the starting x value plus the diameter of the ball is greater than or equal to the x value of the right side of the window. The loop continuation condition, therefore, is that the x value of the ball plus the diameter is less than the x value of the right side of the window.

```
while ( ball.getX( ) + ballDiameter < windowWidth )
```

For some applications, there may be multiple sentinel values. For example, suppose we provide a menu for a user with each menu option being a single character. The user can repeatedly select options from the menu, with the sentinel value being *S* for stop. To allow case-insensitive input, we want to recognize the sentinel value as either *S* or *s*. To do this, we need a compound

loop condition, that is, a loop condition that uses a logical AND (&&) or logical OR (||) operator.

Our first inclination might be to form the condition this way, which is **incorrect**:

```
while ( option != 'S' || option != 's' )  // INCORRECT
```

With this condition, the loop will execute forever. Regardless of what the user enters, the loop condition will be *true*. If the user types *S*, the first expression (`option != 'S'`) is *false*, but the second expression (`option != 's'`) is *true*. Thus, the loop condition evaluates to *true* and the *while* loop body is executed. Similarly, if the user types *s*, the first expression (`option != 'S'`) is *true*, so the loop condition evaluates to *true* and the *while* loop body is executed.

An easy method for constructing a correct *while* loop condition consists of three steps:

1. Define the loop termination condition; that is, define the condition that will make the loop stop executing.
2. Create the loop continuation condition—the condition that will keep the loop executing—by applying the logical NOT operator (!) to the loop termination condition.
3. Simplify the loop continuation condition by applying DeMorgan's Laws, where possible.

Let's use these three steps to construct the correct loop condition for the menu program.

REFERENCE POINT

DeMorgan's Laws are explained in Chapter 5.

1. Define the loop termination condition:

   The loop will stop executing when the user enters an *S* or the user enters an *s*. Translating that into Java, we get

   ```
   ( option == 'S' || option == 's' )
   ```

2. Create the loop continuation condition by applying the ! operator:

   ```
   ! ( option == 'S' || option == 's' )
   ```

3. Simplify by applying DeMorgan's Laws:

   To apply DeMorgan's Laws, we change the == equality operators to !=

and change the logical OR operator (||) to the logical AND operator (&&), producing an equivalent, but simpler expression:

```
( option != 'S' && option != 's' )
```

We now have our loop condition.

To illustrate, let's write an application that calculates the cost of cell phone service. We'll provide a list of options, and the user will select options one at a time until the user enters *S* or *s* to stop. This is an accumulation operation because we are accumulating the total cost of the cell phone service. Example 6.11 shows the code for this application and Figure 6.14 shows the output of a sample run.

```
1 /* Calculate price for cell phone service
2    Anderson, Franceschi
3 */
4
5 import java.util.Scanner;
6 import java.text.DecimalFormat;
7
8 public class CellService
9 {
10    public static void main( String [ ] args )
11    {
12       String menu = "\nAvailable Options";
13       menu += "\n\tA  1,000 anytime minutes: $25.49";
14       menu += "\n\tU  Unlimited weekend minutes: $6.99";
15       menu += "\n\tN  Nationwide long distance: $12.99";
16       menu += "\n\tT  Text messaging: $5.99";
17
18       String optionS;
19       char option;
20       double cost = 10.99; // base cost
21
22       DecimalFormat money = new DecimalFormat( "$###.00" );
23       Scanner scan = new Scanner( System.in );
24
25       System.out.println( "Select the options "
26                           + "for your cell phone service: " );
27       System.out.println( "Base cost: " + money.format( cost ) );
28
29       System.out.println( menu ); // print the menu
30       System.out.print( "Enter an option, "
31                         + "or \"S\" to stop > " );
32       option = scan.next( ).charAt( 0 );
33
```

```
34        while ( option != 'S' && option != 's' )
35        {
36           switch ( option )
37           {
38              case 'a':
39              case 'A':
40                 System.out.println( "1,000 anytime minutes: "
41                                     + "$25.49" );
42                 cost += 25.49;
43                 break;
44              case 'u':
45              case 'U':
46                 System.out.println( "Unlimited weekend minutes: "
47                                     + "$6.99" );
48                 cost += 6.99;
49                 break;
50              case 'n':
51              case 'N':
52                 System.out.println( "Nationwide long distance: "
53                                     + "$12.99" );
54                 cost += 12.99;
55                 break;
56              case 't':
57              case 'T':
58                 System.out.println( "Text messaging: "
59                                     + "$5.99" );
60                 cost += 5.99;
61                 break;
62              default:
63                 System.out.println( "Unrecognized option" );
64           }
65
66           System.out.println( "Current cost: "
67                               + money.format( cost ) );
68
69           System.out.println( menu ); // print the menu
70           System.out.print( "Enter an option, "
71                             + "or \"S\" to stop > " );
72           option = scan.next( ).charAt( 0 );
73        }
74
75        System.out.println( "\nTotal cost of cell service is "
76                            + money.format( cost ) );
77     }
78  }
```

EXAMPLE 6.11 A Compound Loop Condition

Figure 6.14
Calculating Cell Phone Service

```
Select the options for your cell phone service:
Base cost: $10.99

Available Options
       A  1,000 anytime minutes; $25.49
       U  Unlimited weekend minutes: $6.99
       N  Nationwide long distance; $12.99
       T  Text messaging: $5.99
Enter an option, or "S" to stop > a
1,000 anytime minutes: $25.49
Current cost: $36.48

Available Options
       A  1,000 anytime minutes: $25.49
       U  Unlimited weekend minutes: $6.99
       N  Nationwide long distance: $12.99
       T  Text messaging: $5.99
Enter an option, or "S" to stop > U
Unlimited weekend minutes: $6.99
Current cost: $43.47

Available Options
       A  1,000 anytime minutes: $25.49
       U  Unlimited weekend minutes: $6.99
       N  Nationwide long distance: $12.99
       T  Text messaging: $5.99
Enter an option, or "S" to stop > s

Total cost of cell service is $43.47
```

In Example 6.11, we use the compound condition in the *while* loop (line 34). Then within the *while* loop, we use a *switch* statement (lines 36–64) to determine which menu option the user has chosen. We handle case-insensitive input of menu options by including *case* constants for both the lowercase and uppercase versions of each letter option.

COMMON ERROR TRAP

Do not check for the sentinel value inside a *while* loop. Let the *while* loop condition detect the sentinel value.

Note that we don't provide *case* statements for the sentinel values. Instead, we use the *while* loop condition to detect when the user enters the sentinel values.

Animation is another operation that may require a *while* loop with a compound condition. For example, suppose that instead of rolling our ball horizontally, we roll it diagonally down and to the right. To roll the ball diagonally down and to the right, we need to change both the x and the y values in the

while loop body. Thus, within the while loop, we increment both *x* and *y*. We continue as long as the ball has not rolled beyond the right edge of the window and the ball has also not rolled beyond the bottom of the window.

Let's develop the condition by applying our three steps:

1. The loop termination condition is that the ball has rolled beyond either the right edge of the window or the bottom edge of the window.

```
// the ball is out of bounds
( ball.getX( ) + ballDiameter > windowWidth
    || ball.getY( ) + ballDiameter > windowHeight )
```

2. The loop continuation condition is created by applying the logical NOT operator (!) to the loop termination condition:

```
// the ball is not out of bounds
!  ( ball.getX( ) + ballDiameter > windowWidth
       || ball.getY( ) + ballDiameter > windowHeight )
```

3. Simplifying the condition by applying DeMorgan's Law, we get:

```
// the ball is in bounds
( ball.getX( ) + ballDiameter <= windowWidth
    && ball.getY( ) + ballDiameter <= windowHeight )
```

Example 6.12 shows the *RollABall3* class, which uses four *while* loops to roll the ball diagonally down to the right (*x* is incremented, *y* is incremented), then diagonally down to the left (*x* is decremented, *y* is incremented), then diagonally up to the left (*x* is decremented, *y* is decremented), and finally diagonally up to the right (*x* is incremented, *y* is decremented). We set the starting *y* value to 10 and the starting *x* value two-thirds of the way across the window. When the program runs, the ball appears to bounce off the walls of the window, as shown in Figure 6.15.

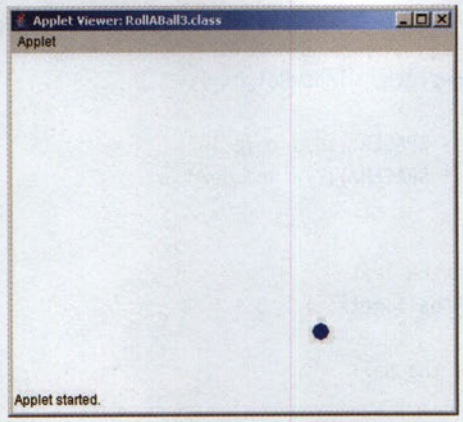
```
1  /* RollABall, Version 3
2     Rolls the ball diagonally
3     Anderson, Franceschi
4  */
5
6  import java.awt.Graphics;
7  import java.awt.Color;
8  import javax.swing.JApplet;
9
10 public class RollABall3 extends JApplet
```

```
11 {
12    public void paint( Graphics g )
13    {
14       super.paint( g );
15
16       final int Y = 10;          // the y value of the ball
17       final int DIAMETER = 15;   // the diameter of the ball
18       final Color COLOR = Color.BLUE; // the color of the ball
19       final int SPACER = 2;      // space between balls
20
21       // get width & height of the applet window
22       int windowWidth = getWidth( );
23       int windowHeight = getHeight( );
24
25       // start x 2/3 across the window
26       int x =  windowWidth * 2 / 3;
27
28       // instantiate the ball as a Circle object
29       Circle ball = new Circle( x, Y, DIAMETER, COLOR );
30
31       // get ball diameter
32       int ballDiameter = ball.getDiameter( );
33
34       // rolling diagonally down to the right
35       while ( ball.getX( ) + ballDiameter <= windowWidth
36                 && ball.getY( ) + ballDiameter <= windowHeight )
37       {
38
39          ball.draw( g );  // draw the ball
40
41          Pause.wait( 0.03 ); // pause for 3/100 of a second
42          // erase the ball
43          g.clearRect( 0, 0, windowWidth, windowHeight );
44
45          ball.setX( ball.getX( ) + SPACER ); // move right
46          ball.setY( ball.getY( ) + SPACER ); // and down
47       }
48
49       // rolling diagonally down to the left
50       while ( ball.getY( )  + ballDiameter < windowHeight
51                && ball.getX( ) > 0 )
52       {
53          ball.draw( g );  // draw the ball
54
```

```
55          Pause.wait( 0.03 ); // pause for 3/100 of a second
56          // erase the ball
57          g.clearRect( 0, 0, windowWidth, windowHeight );
58
59          ball.setX( ball.getX( ) - SPACER ); // move left
60          ball.setY( ball.getY( ) + SPACER ); // and down
61       }
62
63       // rolling diagonally up to the left
64       while ( ball.getY( ) > 0 && ball.getX( ) > 0 )
65       {
66          ball.draw( g );  // draw the ball
67
68          Pause.wait( 0.03 ); // pause for 3/100 of a second
69          // erase the ball
70          g.clearRect( 0, 0, windowWidth, windowHeight );
71
72          ball.setX( ball.getX( ) - SPACER ); // move left
73          ball.setY( ball.getY( ) - SPACER ); // and up
74       }
75
76       // rolling diagonally up to the right
77       while ( ball.getY( )  > 0
78                && ball.getX( ) + ballDiameter < windowWidth )
79       {
80          ball.draw( g );  // draw the ball
81
82          Pause.wait( 0.03 ); // pause for 3/100 of a second
83          // erase the ball
84          g.clearRect( 0, 0, windowWidth, windowHeight );
85
86          ball.setX( ball.getX( ) + SPACER ); // move right
87          ball.setY( ball.getY( ) - SPACER ); // and up
88       }
89
90       ball.draw( g );  // draw the ball
91    }
92 }
```

EXAMPLE 6.12 The *RollABall3* Class

Figure 6.15
Output from *RollABall3*



6.7 Testing Techniques for *while* Loops

It's a good feeling when your code compiles without errors. Getting a clean compile, however, is only part of the job for the programmer. The other part of the job is verifying that the code is correct; that is, that the program produces accurate results.

It usually isn't feasible to test a program with all possible input values, but we can get a reasonable level of confidence in the accuracy of the program by concentrating our testing in three areas:

1. Does the program produce correct results with a set of known input?
2. Does the program produce correct results if the sentinel value is the first and only input?
3. Does the program deal appropriately with invalid input?

Let's take a look at these three areas in more detail:

1. Does the program produce correct results with known input?

To test the program with known input, we select valid input values and determine what the results should be by performing the program's operation either by hand or by using a calculator. For example, to test whether a total or average is computed correctly, enter some values and compare the program's output to a total or average you calculate by entering those same values into a calculator.

It's especially important to select input values that represent boundary conditions, that is, values that are the lowest or highest expected values. For example, to test a program that determines whether a person is old enough to vote in a presidential election (that is, the person is 18 or older), we

should select test values of 17, 18, and 19. These values are the boundary conditions for `age >= 18`; the test values are one integer less, the same value, and one integer greater than the legal voting age. We then run the program with the three input values and verify that the program correctly identifies 17 as an illegal voting age and 18 and 19 as legal voting ages.

2. Does the program produce correct results if the sentinel value is the first and only input?

In our *while* loops, when we find the sentinel value, the flow of control skips the *while* loop body and picks up at the statement following the *while* loop. When the sentinel value is the first input value, our *while* loop body does not execute at all. We simply skip to the statement following the *while* loop. In cases like this, the highly respected computer scientist Donald Knuth recommends that we "do exactly nothing, gracefully."

In many programs that calculate a total or an average for the input values, when no value is input, your program should either report the total or average as 0 or output a message that no values were entered. Thus, it's important to write your program so that it tolerates no input except the sentinel value; therefore, we need to test our programs by entering the sentinel value first.

Let's revisit the earlier examples in this chapter to see how they handle the case when only the sentinel value is entered.

In the addition calculator (Example 6.3), we set the total to 0 before the *while* loop and simply report the value of total after the *while* loop. So we get the correct result (0) with only the sentinel value.

SOFTWARE ENGINEERING TIP

Expect that the user might enter the sentinel value first. Your program needs to handle this special case.

In Example 6.4 where we count the percentage of passing test scores, we handle the sole sentinel value by performing some additional checking after the *while* loop. If only the sentinel value is entered, the count will be 0. We check for this case and if we find a count of 0, we skip reporting the percentage so that we avoid dividing by 0. We use similar code in Example 6.5, where we calculate the average test score. If we detect a count of 0, we also skip the calculation of the average to avoid dividing by 0 and simply report the class average as 0.

3. Does the program deal appropriately with invalid input?

If the program expects a range of values or certain discrete values, then it should notify the user when the input doesn't fit the expected values.

In Example 6.11, we implemented a menu for calculating the cost of cell phone service. The user could enter *s*, *a*, *u*, *n*, or *t* (or the corresponding

capital letters) representing their desired service options. If the user enters a letter other than those expected values, we use the *default* clause of the *switch* statement to issue an error message, "*Unrecognized option.*"

In the next section, we explain how to validate that user input is within a range of values using a *do/while* loop.

6.8 Event-Controlled Loops Using *do/while*

Another form of loop that is especially useful for validating user input is the *do/while* loop. In the *do/while* loop, the loop condition is tested at the end of the loop (instead of at the beginning, as in the *while* loop). Thus the body of the *do/while* loop is executed at least once.

The syntax of the *do/while* loop is the following:

```
// initialize variables
do
{
   // body of loop
} while ( condition );
// process the results
```

Figure 6.16 shows the flow of control of a *do/while* loop.

To use the *do/while* loop to validate user input, we insert the prompt for the input inside the body of the loop, then use the loop condition to test the value of the input. Like the *while* loop, the body of the loop will be reexecuted if the condition is *true*. Thus, we need to form the condition so that it's *true* when the user enters invalid values.

Example 6.13 implements a *do/while* loop (lines 14–18) that prompts the user for an integer between 1 and 10. Figure 6.17 shows the output of the program. If the user enters a number outside the valid range, we reprompt the user until the input is between 1 and 10. Thus the condition for the *do/while* loop (line 18) is

```
while ( number < 1 || number > 10 )
```

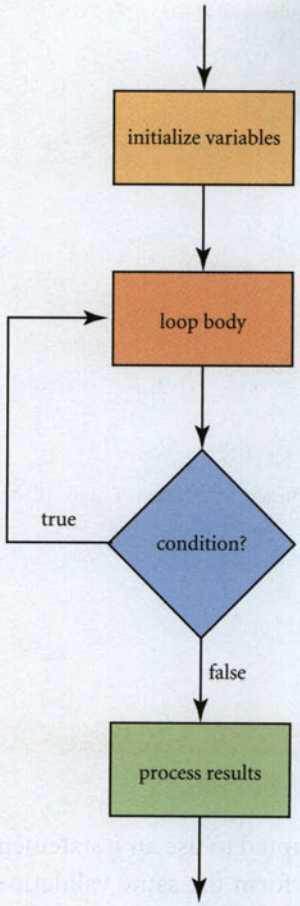


Figure 6.16
The Flow of Control of a *do/while* Statement

```
Enter a number between 1 and 10 > 20
Enter a number between 1 and 10 > -1
Enter a number between 1 and 10 > 0
Enter a number between 1 and 10 > 11
Enter a number between 1 and 10 > 5
Thank you!
```

Figure 6.17
Validating Input

```
1 /* Validate input is between 1 and 10
2    Anderson, Franceschi
3 */
4
5 import java.util.Scanner;
6
7 public class ValidateInput
8 {
9   public static void main( String [ ] args )
10  {
11    int number;  // input value
12    Scanner scan = new Scanner( System.in );
13
14    do
15    {
16       System.out.print( "Enter a number between 1 and 10 > "  );
17       number = scan.nextInt( );
18    } while ( number < 1 || number > 10 );
19
20    System.out.println( "Thank you!" );
21  }
22 }
```

EXAMPLE 6.13 Validating User Input

For validating input, you may be tempted to use an *if* statement rather than a *do/while* loop. For example, to perform the same validation as Example 6.13, you may try this **incorrect** code:

```
System.out.print( "Enter a number between 1 and 10 > " );
number = scan.nextInt( );

if ( number < 1 || number > 10 ) // INCORRECT!
{
    System.out.print( "Enter a number between 1 and 10 > " );
    number = scan.nextInt( );
}
```

COMMON ERROR TRAP

Do not use an *if* statement to validate input because it will catch invalid values entered the first time only. Use a *do/while* loop to reprompt the user until the user enters a valid value.

The problem with this approach is that the *if* statement will reprompt the user only once. If the user enters an invalid value a second time, the program will not catch it. A *do/while* loop, however, will continue to reprompt the user as many times as needed until the user enters a valid value.

Skill Practice
with these end-of-chapter questions

6.14.2 Reading and Understanding Code
Questions 9, 10

6.14.3 Fill In the Code
Question 28

6.14.4 Identifying Errors in Code
Question 34

6.14.5 Debugging Area
Question 36

6.14.6 Write a Short Program
Question 53

CODE IN ACTION

To see two step-by-step illustrations of *do/while* loops, look for the Flash movie in the Chapter 6 folder on the CD-ROM included with this book. Click on the link for Chapter 6 to start the movie.

6.9 Programming Activity 1: Using *while* Loops

In this activity, you will work with a sentinel-controlled *while* loop, performing this activity:

> Write a *while* loop to process the contents of a grocery cart and calculate the total price of the items. It is important to understand that, in this example, we do not know how many items are in the cart.

The framework will animate your code and display the current subtotal so that you can check the correctness of your code. The window will display the various *Item* objects moving down a conveyor belt toward a grocery bag. It will also display the unit price of the item and your current subtotal, as well as the correct subtotal.

For example, Figure 6.18 demonstrates the animation: We are currently scanning the first item, a milk carton, with a unit price of $2.00; thus, the correct subtotal is $2.00.

As the animation will show, *Item* objects could be milk, cereal, orange juice, or the divider bar. The number of *Item* objects in the cart is determined

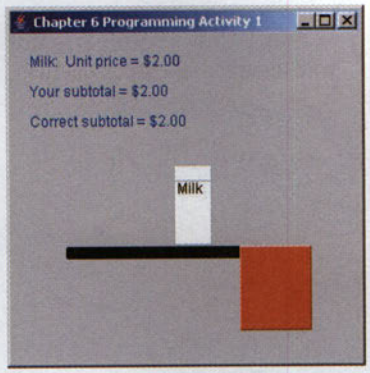


Figure 6.18
Animation of the *Cashier* Application

randomly; as you watch the animation, sometimes you will find that there are two items in the cart, sometimes six, sometimes three, and so forth. Scanning the divider bar signals the end of the items in the cart.

Task Instructions

Copy the files in the Chapter 6 Programming Activity 1 directory on the CD-ROM accompanying this book to a directory on your computer. Searching for five stars (*****) in the *Cashier.java* source code will show you where to add your code. You will add your code inside the *checkout* method of the *Cashier* class (the method header for the *checkout* method has already been coded for you). Example 6.14 shows a fragment of the *Cashier* class, where you will add your code:

```
public void checkout( )
{
 /* ***** Student writes the body of this method ***** */
 //
 //  Using a while loop, calculate the total price
 //  of the groceries.
 //
 // The getNext method (in this Cashier class) returns the
 // next item on the conveyor belt, which is an Item object
 // (we do not know which item and we do not know how many items
 // are in the cart - this is randomly generated).
 // getNext does not take any arguments. Its API is:
 //       Item getNext( )
 //
 // Right after you update the current subtotal,
 // you should call the animate method.
 // The animate method takes one parameter: a double,
 // which is your current subtotal.
 // For example, if the name of your variable representing
```

```
    // the current subtotal is total, your call to the animate
    // method should be:
    //    animate( total );
    //
    // The instance method getPrice of the Item class
    // returns the price of the Item object.
    // The method getPrice does not take any arguments.
    // Its API is:
    //         double getPrice( )
    //
    // The cart is empty when the getNext method returns
    // the divider Item.
    // You detect the divider Item because its price
    // is -0.99. So an Item with a price of -0.99
    // is the sentinel value for the loop.
    //
    // After you scan the divider, display the total
    // for the cart in a dialog box.

    // End of student code
}
```

EXAMPLE 6.14 The *checkout* Method in *Cashier.java*

- You can access items in the cart by calling the *getNext* method of the *Cashier* class, which has the following API:

  ```
  Item getNext( )
  ```

  The *getNext* method returns an *Item* object, which represents an *Item* in the cart. As you can see, the *getNext* method does not take any arguments. Since we call the method *getNext* from inside the *Cashier* class, we call the method without an object reference. For example, a call to *getNext* could look like the following:

  ```
  Item newItem;

  newItem = getNext( );
  ```

  The *getNext* method is already written and contains code to generate the animation; it is written in such a way that the first *Item* object on the conveyor belt may or may not be the divider. (If the first *Item* is the divider, the cart is empty.)

- After you get a new *Item*, you can "scan" the item to get its price by calling the *getPrice* method of the *Item* class. The *getPrice* method has this API:

  ```
  double getPrice( )
  ```

Thus, you would get an item, then get its price using code like the following:

```
Item newItem;
double price;

newItem = getNext( );
price = newItem.getPrice( );
```

- After adding the price of an item to your subtotal, call the *animate* method of the *Cashier* class. This method will display both your subtotal and the correct subtotal so that you can verify that your code is correct.

  The animate method has the following API:

  ```
  void animate( double subtotal )
  ```

  Thus, if your variable representing the current total is *total*, you would call the animate method using the following code:

  ```
  animate( total );
  ```

- We want to exit the loop when the next *Item* is the divider. You will know that the *Item* is the divider because its price will be –0.99 (negative 0.99); thus, scanning an *Item* whose price is –0.99 should be your condition to exit the *while* loop.
- After you scan the divider, display the total for the cart in a dialog box. Verify that your total matches the correct subtotal displayed.
- To test your code, compile and run the application from the *Cashier* class.

Troubleshooting

If your method implementation does not animate or animates incorrectly, check these items:

- Verify that you have correctly coded the priming read.
- Verify that you have correctly coded the condition for exiting the loop.
- Verify that you have correctly coded the body of the loop.

DISCUSSION QUESTIONS ?

1. What is the sentinel value of your *while* loop?
2. Explain the purpose of the priming read.

6.10 Count-Controlled Loops Using *for*

Before the loop begins, if you know the number of times the loop body should execute, you can use a *count-controlled loop*. The *for* loop is designed for count-controlled loops, that is, when the number of iterations is determined before the loop begins.

6.10.1 Basic Structure of *for* Loops

The *for* loop has this syntax:

```
for ( initialization; loop condition; loop update )
{
      // loop body
}
```

Notice that the initialization, loop condition, and loop update in the *for* loop header are separated by semicolons (not commas). Notice also that there is no semicolon after the closing parenthesis in the *for* loop header. A semicolon here would indicate an empty *for* loop body. Although some advanced programs might correctly write a *for* loop with an empty loop body, the programs we write in this book will have at least one statement in the *for* loop body.

COMMON ERROR TRAP

Use semicolons, rather than commas, to separate the statements in a *for* loop header.

The scope of any variable declared within the *for* loop header or body extends from the point of declaration to the end of the *for* loop body.

The flow of control of the *for* loop is shown in Figure 6.19. When the *for* loop is encountered, the initialization statement is executed. Then the loop condition is evaluated. If the condition is true, the loop body is executed, then the loop update statement is executed, and the loop condition is reevaluated. Again, if the condition is true, the loop body is executed, followed by the loop update, then the reevaluation of the condition, and so on, until the condition is false.

COMMON ERROR TRAP

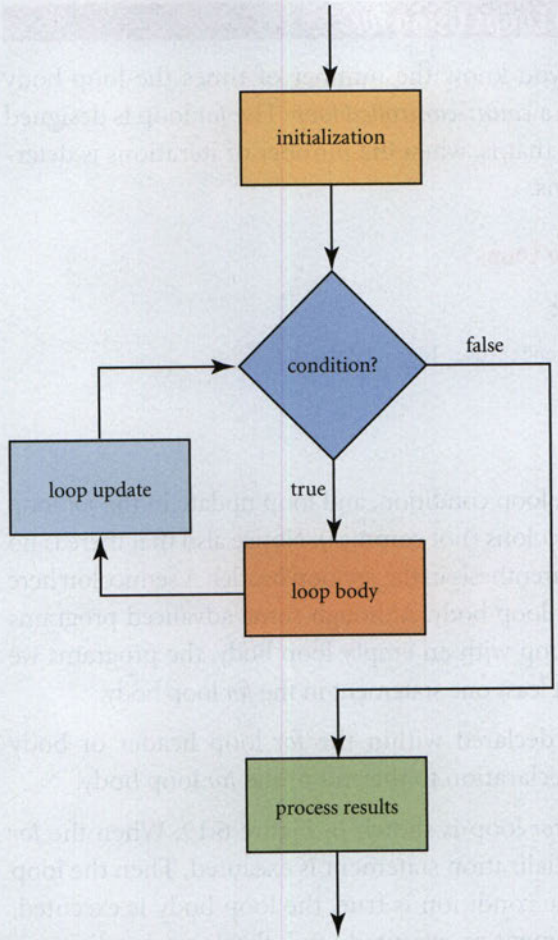
Adding a semicolon after the closing parenthesis in the *for* loop header indicates an empty loop body and will likely cause a logic error.

The *for* loop is equivalent to the following *while* loop:

```
// initialization
while ( loop condition )
{
   // loop body
   // loop update
}
```

As you can see, *while* loops can be used for either event-driven or count-controlled loops. A *for* loop is especially useful for count-controlled loops,

Figure 6.19
Flow of Control of the *for* Loop



however. Because all the loop control is contained in the *for* loop header, you can easily see what condition will stop the loop and how the condition will be updated after each iteration.

6.10.2 Constructing *for* Loops

Typically, we use a **loop control variable** in a *for* loop; that control variable is usually used for counting. We set its initial value in the initialization statement, increment or decrement its value in the loop update statement, and check its value in the loop condition.

For example, if we want to find the sum of five integers, we know the loop body should execute five times—once for each integer. We set our loop

control variable to 1 in the initialization statement, increment the loop control variable by 1 in the loop update statement, and check if its value is less than or equal to 5 in the loop condition. When the loop update statement increments the control variable's value to 6, we will have executed the loop body five times. The pseudocode for this program is the following:

```
set total to 0
for i = 1 to 5 by 1
{
  read integer
  add integer to total
}
print the total
```

With a *for* loop, we do not need to perform a priming read because the condition for exiting the loop is controlled by a counter, not by an input value.

Example 6.15 shows the *for* loop for calculating the sum of five integers.

```
1 /* Find the total of 5 numbers
2    Anderson, Franceschi
3 */
4
5 import java.util.Scanner;
6
7 public class Sum5Numbers
8 {
9   public static void main( String [ ] args )
10  {
11    int total = 0;  // stores the sum of the 5 numbers
12    int number;     // stores the current input
13
14    Scanner scan = new Scanner( System.in );
15
16    for ( int i = 1; i <= 5; i++ )
17    {
18      System.out.print( "Enter an integer > " );
19      number = scan.nextInt( );
20
21      total += number;  // add input to total
22    }
23
24    // process results by printing the total
```

```
25      System.out.println( "The total is " + total );
26    }
27 }
```

EXAMPLE 6.15 Finding the Sum of Five Numbers

In this example, which is a standard accumulation operation, the *for* loop initialization statement declares *i*, which will be our loop control variable. We start *i* at 1, and after each execution of the loop body, we increment *i* by 1 in the loop update statement. The loop condition checks if the value of *i* is less than or equal to 5; when *i* reaches 6, we have executed the loop body five times. Figure 6.20 shows the execution of this *for* loop.

Note that because we declare our loop counter variable *i* in the *for* loop header, we cannot reference *i* after the *for* loop ends. Thus, this code would generate a compiler error, because *i* is out of scope on line 24:

```
16      for ( int i = 1; i <= 5; i++ )
17      {
18        System.out.print( "Enter an integer > " );
19        number = scan.nextInt( );
20
21        total += number;  // add input to total
22      }
23
24      System.out.println( "The total for " + ( i - 1 )
25                          + " number is " + total );
```

Defining a new variable using the same name as a variable already in scope is invalid and generates a compiler error. However, a variable name can be reused when a previously defined variable with the same name is no longer in scope. In the code above, the scope of the variable *i* defined in line 16 is

Figure 6.20
Finding the Sum of Five Integers

```
Enter an integer > 12
Enter an integer > 10
Enter an integer > 5
Enter an integer > 7
Enter an integer > 3
The total is 37
```

limited to the *for* loop on lines 16–22. We cannot define another variable named *i* in that *for* loop; however, as shown here, we could reuse the name *i* in a subsequent *for* loop (lines 24–29), because the first *i* is no longer in scope. In fact, programmers often reuse the variable name *i* for the counter variable in their *for* loops.

```
16    for ( int i = 1; i <= 5; i++ )
17    {
18      System.out.print( "Enter an integer > " );
19      number = scan.nextInt( );
20
21      total += number;  // add input to total
22    }
23
24    for ( int i = 1; i <= 10; i++ )
25    {
26      System.out.print( "Enter integer " + i + " > " );
27      number = scan.nextInt( );
28    }
```

If you do want to refer to the loop variable after the loop ends, you can define the variable before the *for* loop, as shown in the following code:

```
15    int i;
16    for ( i = 1; i <= 5; i++ )
17    {
18      System.out.print( "Enter an integer > " );
19      number = scan.nextInt( );
20
21      total += number;  // add input to total
22    }
23
24    System.out.println( "The total for " + ( i - 1 )  // i is 6
25                        + " numbers is " + total );
```

We can also increment the loop control variable by values other than 1. Example 6.16 shows a *for* loop that increments the control variable by 2 to print the even numbers from 0 to 20.

The pseudocode for this program is the following:

```
set output to an empty String
for i = 0 to 20  by 2
{
   append i and a space to the output String
}
print the output String
```

We start with an empty *String* variable, *toPrint*, and with each iteration of the loop we append the next even number and a space. When the loop completes, we output *toPrint*, which prints all numbers on one line, as shown in Figure 6.21.

```
1 /* Print the even numbers from zero to twenty
2    Anderson, Franceschi
3 */
4
5 public class PrintEven
6 {
7   public static void main( String [ ] args )
8   {
9     String toPrint = "";  // initialize output String
10
11    for ( int i = 0; i <= 20; i += 2 )
12    {
13       toPrint += i + " "; // append current number and a space
14    }
15
16    System.out.println( toPrint ); // print results
17  }
18 }
```

EXAMPLE 6.16 Printing Even Numbers

```
0 2 4 6 8 10 12 14 16 18 20
```

Figure 6.21
Printing Even Numbers from 0 to 20

In this example, we initialize the loop control variable to 0, then increment *i* by 2 in the loop update statement (`i += 2`) to skip the odd numbers. Notice that we used the value of the loop control variable *i* inside the loop. The loop control variable can perform double duty such as this because the loop control variable is available to our code in the loop body.

The loop control variable also can be used in our prompt to the user. For example, in Example 6.15, we could have prompted the user for each integer using this statement:

```
System.out.print( "Enter integer " + i  + " > " );
```

```
Enter integer 1 > 23
Enter integer 2 > 12
Enter integer 3 > 10
Enter integer 4 > 11
Enter integer 5 > 15
The total is 71
```

Figure 6.22
Adding the Loop Control Variable to the Prompt

Then the user's prompt would look like that shown in Figure 6.22.

CODE IN ACTION

To see a step-by-step illustration of a *for* loop, look for the Flash movie in the Chapter 6 folder on the CD-ROM included with this book. Click on the link for Chapter 6 to start the movie.

We can also decrement the loop control variable. Example 6.17 shows an application that reads a sentence entered by the user and prints the sentence backward.

The pseudocode for this program is the following:

```
set backwards to an empty String
read a sentence

for i = ( length of sentence – 1 ) to 0  by –1
{
  get character at position i in sentence
  append character to backwards
}
print backwards
```

To print a sentence backward, we treat the sentence, a *String*, like a stream of characters; each iteration of the loop extracts and processes one character from the *String*, using the *charAt* method of the *String* class. Line 10 declares two *Strings*: *original*, to hold the sentence the user enters, and *backwards* (initialized as an empty *String*), to hold the reverse of the user's sentence. Lines

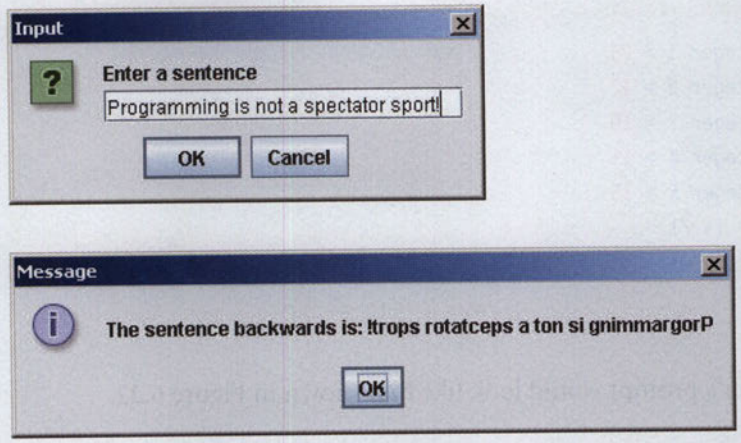


Figure 6.23
Printing a Sentence Backward

12–13 prompt the user for a sentence. Lines 15 through 18 make up the *for* loop, whose purpose is to copy the original sentence backward into the *String backwards*. We do this by starting the copying at the last character in the original *String* and moving backward in the *String* one character at a time until we have copied the first character in *original*. Thus, we initialize our loop variable to the position of the last character in *original* (`original.length( ) - 1`) and extract one character at a time, appending it to *backwards*. The loop update statement (`i--`) moves the loop variable backward by one position, and our loop condition (`i >= 0`) checks whether we have reached the beginning of the *String original*. Figure 6.23 shows the execution of the program with the user entering the sentence, "*Programming is not a spectator sport!*"

```
1 /* Print a sentence backward
2    Anderson, Franceschi
3 */
4 import javax.swing.JOptionPane;
5
6 public class Backwards
7 {
8   public static void main( String [ ] args )
9   {
10    String original, backwards = "";
11
12    original = JOptionPane.showInputDialog( null,
13                              "Enter a sentence" );
14
15    for ( int i = original.length( ) - 1; i >= 0; i-- )
16    {
17        backwards += original.charAt( i );
```

```
18     }
19
20     JOptionPane.showMessageDialog( null,
21                "The sentence backwards is: " + backwards );
22   }
23 }
```

EXAMPLE 6.17 Printing a Sentence Backward

We can display some interesting graphics using *for* loops. The applet in Example 6.18 draws the bull's-eye target shown in Figure 6.24. To make the bull's-eye target, we draw 10 concentric circles (circles that have the same center point), beginning with the largest circle and successively drawing a smaller circle on top of the circles already drawn. Thus, the bull's-eye target circles have the same center point, but different diameters. The pseudocode for this program is

```
for diameter = 200 to 20 by -20
{
   instantiate a circle
   draw the circle
   if color is black
      set color to red
   else
      set color to black
}
```

Figure 6.24
Drawing a Bull's-Eye Target

REFERENCE POINT

The API for the *Circle* class is given in Section 6.4.5

We can again use the *Circle* class introduced in Section 6.4.5, when we rolled a ball.

Translating the pseudocode into Java, we get the code shown in Example 6.18.

```
1 /* Bull's-eye target
2    Anderson, Franceschi
3 */
4
5 import javax.swing.JApplet;
6 import java.awt.Color;
7 import java.awt.Graphics;
8
9 public class Bullseye extends JApplet
10 {
11    // center of bullseye
12    private int centerX = 200, centerY = 150;
13    // color of first circle
14    private Color toggleColor = Color.BLACK;
15    // each circle will be a Circle object
16    private Circle circle;
17
18    public void paint( Graphics g )
19    {
20      super.paint( g );
21
22      for ( int diameter = 200; diameter >= 20; diameter -= 20 )
23      {
24        // instantiate circle with current diameter and color
25        circle = new Circle( centerX - diameter / 2,
26                             centerY - diameter / 2,
27                             diameter, toggleColor );
28
29        circle.draw( g ); // draw the circle
30
31        if ( toggleColor.equals( Color.BLACK ) )
32          toggleColor = Color.RED;   // if black, change to red
33        else
34          toggleColor = Color.BLACK; // if red, change to black
35      }
36    }
37 }
```

EXAMPLE 6.18 Drawing a Bull's-Eye Target

Our *for* loop initialization statement in line 22 sets up the diameter of the largest circle as 200 pixels and the loop update statement decreases the diameter of each circle by 20 pixels. The smallest circle we want to draw should have a diameter of 20 pixels, so we set the loop condition to check that the diameter is greater than or equal to 20. We need to start with the largest circle rather than the smallest circle so that new circles we draw don't hide the previously drawn circles.

Drawing the bull's-eye target circles illustrates two common programming techniques: conversion between units and a toggle variable.

We need to convert between units because the *Circle* class constructor takes as its arguments the upper-left (x, y) coordinate and the width and height of the circle's bounding rectangle (this is consistent with the *fillOval* method of the *Graphics* class). However, all our circles have the same center point, but not the same upper-left x and y coordinates. Given the diameter and the center point of the circle, however, we can calculate the (x, y) coordinate of the upper-left corner. Figure 6.25 shows how we make the conversion.

The difference between the center point and the upper-left corner of the bounding rectangle is the radius of the circle, which is half of the diameter

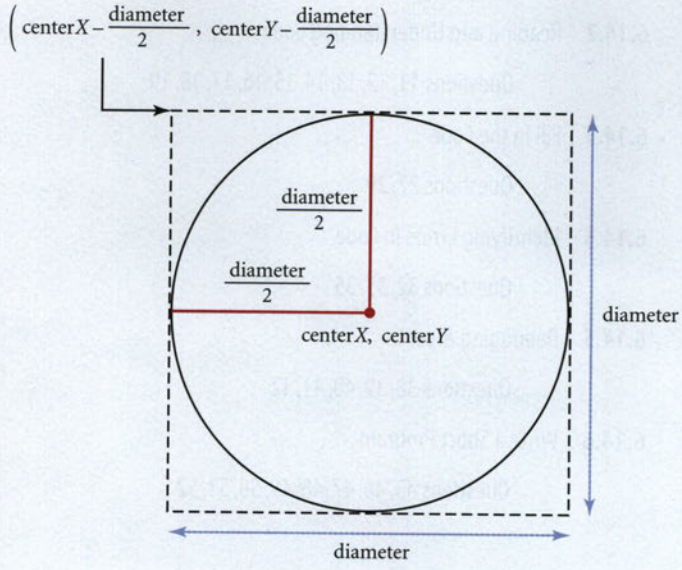


Figure 6.25
Converting Circle Coordinates

(`diameter / 2`). So, the upper-left *x* value is the *x* value of the center point minus half the diameter (`centerX - diameter / 2`). Similarly, the upper-left *y* value is the *y* value of the center point minus half the diameter (`centerY - diameter / 2`).

Thus, we instantiate each circle using the following statement:

```
circle = new Circle( centerX - diameter / 2,
                     centerY - diameter / 2,
                     diameter, toggleColor );
```

SOFTWARE ENGINEERING TIP

Use a toggle variable when you need to alternate between two values.

To alternate between red and black circles, we use a **toggle variable**, which is a variable whose value alternates between two values. We use a *Color* object for our toggle variable, *toggleColor*, and initialize it to *Color.BLACK*. After drawing each circle, we switch the color (lines 31–34). If the current color is black, we set it to red; otherwise, the color must be red, so we set the color to black.

Skill Practice
with these end-of-chapter questions

6.14.1 Multiple Choice Exercises

Questions 2, 3, 4

6.14.2 Reading and Understanding Code

Questions 11, 12, 13, 14, 15, 16, 17, 18, 19

6.14.3 Fill In the Code

Questions 27, 29

6.14.5 Identifying Errors in Code

Questions 32, 33, 35

6.14.5 Debugging Area

Questions 38, 39, 40, 41, 42

6.14.6 Write a Short Program

Questions 43, 46, 47, 48, 49, 50, 51, 52

6.10.3 Testing Techniques for *for* Loops

One of the most important tests for *for* loops is that the starting and ending values of the loop variable are set correctly. For example, to execute a *for* loop five times, we could set the initial value of the loop variable to 0 and use the condition (`i < 5`), or we could set the initial value of the loop variable to 1 and use the condition (`i <= 5`). Either of these *for* loop headers will cause the loop to execute five times.

```
for ( int i = 0; i < 5; i++ ) // executes 5 times
```

or

```
for ( int i = 1; i <= 5; i++ ) // executes 5 times
```

However, the following *for* loop header is incorrect; the loop will execute only four times.

```
for ( int i = 1; i < 5; i++ ) // INCORRECT! executes only 4 times
```

Thus to test the *for* loop in Example 6.15 that prompts for five integers, we need to verify that the program outputs exactly five prompts. To test that we are prompting the user five times, we can enter the integers 1, 2, 3, 4, and 5 at the prompts. Another option, shown in Figure 6.26, is to append a number to the prompt, which does double duty. Besides keeping the user informed of the number of integers entered so far, it also helps to verify that we have the correct number of prompts.

Like *while* loops, the body of a *for* loop may not be executed at all. If the loop condition is *false* the first time it is tested, the body of the *for* loop is skipped. Thus, when testing, we want to simulate input that would cause the loop condition to be *false* when the *for* loop statement is first encountered. For example, in the *Backwards* class in Example 6.17, we need to test

```
Enter integer 1 > 23
Enter integer 2 > 12
Enter integer 3 > 10
Enter integer 4 > 11
Enter integer 5 > 15
The total is 71
```

Figure 6.26
Counting Five Prompts

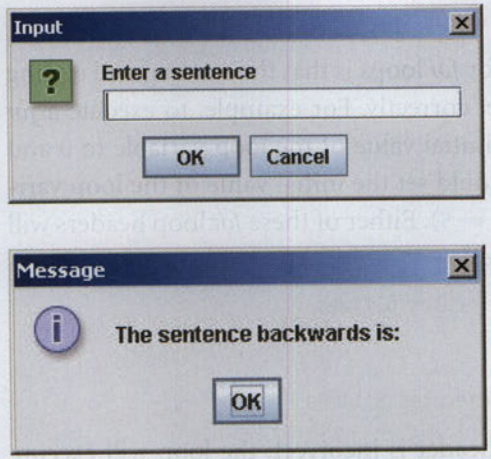


Figure 6.27
The Backwards Class with an Empty Sentence

the *for* loop with an empty sentence. In other words, when the prompt appears to enter a sentence, we simply press the *OK* button. If you try this, you will find that the application still works, as Figure 6.27 shows.

The program works correctly with an empty sentence because the *for* loop initialization statement is

```
int i = original.length( ) - 1;
```

Because the length of an empty *String* is 0, this statement sets *i* to –1. The loop condition (`i >= 0`) is *false*, so the loop body is never executed. The flow of control skips to the statement following the loop,

```
JOptionPane.showMessageDialog( null,
            "The sentence backwards is: " + backwards );
```

which prints an empty *String*. Although it would be more user-friendly to check whether the sentence is empty and print a message to that effect, the program does indeed do exactly nothing, gracefully.

6.11 Nested Loops

Loops can be nested inside other loops; that is, the body of one loop can contain another loop. For example, a *while* loop can be nested inside another *while* loop or a *for* loop can be nested inside another *for* loop. In fact, the nested loops do not need to be the same loop type; that is, a *for*

loop can be nested inside a *while* loop, and a *while* loop can be nested inside a *for* loop.

Nested loops may be useful if you are performing multiple operations, each of which has its own count or sentinel value. For example, we may be interested in processing data in a statistics table containing rows and columns. In order to process all the data, we could loop from the first row to the last row; inside that loop, we would process each row by looping from the first column to the last column of that row. A statistics table can be stored in what is called a two-dimensional array, a subject we discuss in Chapter 9.

REFERENCE POINT

Nested *for* loops are useful for processing data stored in two-dimensional arrays, which are discussed in Chapter 9. Simple *for* loops are useful to process data stored in standard arrays; standard arrays are discussed in Chapter 8.

Going back to Jane, our grocery cashier, her workday can be modeled using nested loops. In Programming Activity 1, we wrote the code for our cashier to calculate the total cost of the contents of one customer's grocery cart. But cashiers check out multiple customers, one after another. While the line of people in front of the cashier is not empty, she will help the next customer. For each customer, she will set the total order to $0.00 and start scanning items and add the prices to the total. While the current customer still has items in the cart, Jane will scan the next item. When Jane finishes processing a customer's cart, she will check to see if there is a customer waiting in line. If there is one, she will set the total to $0.00 and start scanning the next customer's items.

Thus, the cashier's job can be described using a *while* loop nested inside another *while* loop. The pseudocode for these nested loops is shown here:

```
look for a customer
while there is a customer in line
{
  set total to $0.00
  reach for first item
  while item is not the divider bar
  {
      add price to total
      reach for next item
  }
  // if we get here, the item is the divider bar
  output the total price

  look for another customer
}
```

The important point to understand with nested loops is that the inner (or nested) loop executes completely (executes all its iterations) for each single iteration of the outer loop.

Let's look at a simple example that uses nested *for* loops. Suppose we want to print five rows of numbers as shown here:

```
1
1 2
1 2 3
1 2 3 4
1 2 3 4 5
```

We can see a pattern here. In the first line, we print one number; in the second line, we print two numbers, and so on. In other words, the quantity of numbers we print and the line number are the same. The pseudocode for this pattern is the following:

```
for line = 1 to 5 by 1
{
  for number = 1 to line by 1
  {
    print number and a space
  }
  print a new line
}
```

Translating this pseudocode into nested *for* loops, we get the code shown in Example 6.19.

```
1 /*  Printing numbers using nested for loops
2     Anderson, Franceschi
3 */
4
5 public class NestedForLoops
6 {
7   public static void main( String [ ] args )
8   {
9     // outer for loop prints 5 lines
10    for ( int line = 1; line <= 5; line++ )
11    {
12      // inner for loop prints one line
13      for ( int number = 1; number <= line; number++ )
```

```
14        {
15          // print the number and a space
16          System.out.print( number + " " );
17        }
18
19        System.out.println( );    // print a newline
20      }
21    }
22 }
```

EXAMPLE 6.19 Nested *for* Loops

Notice that the inner *for* loop (lines 12–17) uses the value of *line*, which is set by the outer *for* loop (lines 9–20). Thus, for the first iteration of the outer loop, *line* equals 1, so the inner loop executes once, printing the number 1 and a space. Then we print a newline character because line 19 is part of the outer *for* loop. The outer loop then sets the value of *line* to 2, and the inner loop starts again at 1 and executes two times (until *number* equals the line number in the outer loop). Then we again print a newline. This operation continues until the *line* exceeds 5, when the outer loop terminates. The output from Example 6.19 is shown in Figure 6.28.

Note that we needed to use different names for our *for* loop control variables. The loop control variable *line* is in scope from lines 10 to 20, which includes the inner *for* loop.

Let's look at another example of a nested loop. We'll let the user enter positive integers, with a 0 being the sentinel value. For each number, we'll find all its factors; that is, we will find all the integers that are evenly divisible into the number, except 1 and the number itself.

If a number is evenly divisible by another, the remainder after division will be 0. The modulus operator (%) will be useful here, because it calculates the remainder after integer division. Thus, to find all the factors of a number,

```
1
1 2
1 2 3
1 2 3 4
1 2 3 4 5
```

Figure 6.28
Output from Example 6.19

we can test all integers from 1 up to the number to see if the remainder after division is 0. But let's think about whether that's a good approach. The number 1 will be a factor for every number, because every number is evenly divisible by 1. So we can test integers beginning at 2. Then, because 2 is the smallest factor, there's no need to test integers higher than *number / 2*. Thus, our range of integers to test will be from 2 to *number / 2*.

For this example, we'll use a *for* loop nested inside a *while* loop. The pseudocode for this example is

```
read first number  // priming read
while number is not 0
{
  print "The factors for number are "
  for factor = 2 to ( number / 2 ) by 1
  {
    if number % factor is 0
      print factor and a space
  }
  print a new line

  read next number  // update read
}
```

But what happens if we don't find any factors for a number? In that case, the number is a prime number. We can detect this condition by using a *boolean* variable called a **flag**. We set the flag to *false* before starting the *for* loop that checks for factors. Inside the *for* loop, we set the flag to *true* when we find a factor. In other words, we signal (or flag) the fact that we found a factor. Then after the *for* loop terminates, we check the value of the flag. If it is still *false*, we did not find any factors and the number is prime. Our pseudocode for this program now becomes

```
read first number  // priming read
while number is not 0
{
  print "The factors for number are "
  set flag to false
```

```
  for factor = 2  to ( number / 2 ) by 1
  {
    if number % factor is 0
    {
      print factor and a space
      set flag to true
    }
  }

  if  flag is false
    print "number is prime"

  print a new line

  read next number   // update read
}
```

Since we want to read positive numbers only, the lines "read first number" and "read next number" in the preceding pseudocode will actually be more complex than a simple statement. Indeed, we will prompt the user to enter a positive number until the user does so. In order to do that, we will use a *do/while* loop to validate the input from the user. Therefore, inside the *while* loop, we nest not only a *for* loop, but also a *do/while* loop. In the interest of keeping the pseudocode simple, we did not show that *do/while* loop. However, it is included in the code in Example 6.20 at lines 17–23 and 45–51.

Translating this pseudocode into Java, we get the code shown in Example 6.20; the output of a sample run of the program is shown in Figure 6.29.

```
 1 /* Factors of integers
 2    with checks for primes
 3    Anderson, Franceschi
 4 */
 5 import java.util.Scanner;
 6
 7 public class Factors
 8 {
 9   public static void main( String [ ] args )
10   {
```

```
11    int number;           // positive integer entered by user
12    final int SENTINEL = 0;
13    boolean factorsFound; // flag signals whether factors are found
14
15    Scanner scan = new Scanner( System.in );
16
17    // priming read
18    do
19    {
20      System.out.print( "Enter a positive integer "
21                        + "or 0 to exit > " );
22      number = scan.nextInt( );
23    } while ( number < 0 );
24
25    while ( number != SENTINEL )
26    {
27      System.out.print( "Factors of " + number + ":  " );
28      factorsFound = false;  // reset flag to no factors
29
30      for ( int factor = 2; factor <= number / 2; factor++ )
31      {
32        if ( number % factor == 0 )
33        {
34          System.out.print( factor + " " );
35          factorsFound = true;
36        }
37      } // end of for loop
38
39      if ( ! factorsFound )
40        System.out.print( "none, " + number + " is prime" );
41
42      System.out.println( );  // print a newline
43      System.out.println( );  // print a second newline
44
45      // read next number
46      do
47      {
48        System.out.print( "Enter a positive integer "
49                          + "or 0 to exit > " );
50        number = scan.nextInt( );
51      } while ( number < 0 );
52    } // end of while loop
53  }
54 }
```

EXAMPLE 6.20 Finding Factors

```
Enter a positive integer or 0 to exit > 100
Factors of 100:  2 4 5 10 20 25 50

Enter a positive integer or 0 to exit > 25
Factors of 25:  5

Enter a positive integer or 0 to exit > 21
Factors of 21:  3 7

Enter a positive integer or 0 to exit > 13
Factors of 13:  none, 13 is prime

Enter a positive integer or 0 to exit > 0
```

Figure 6.29
Output of Finding Factors

6.12 Programming Activity 2: Using *for* Loops

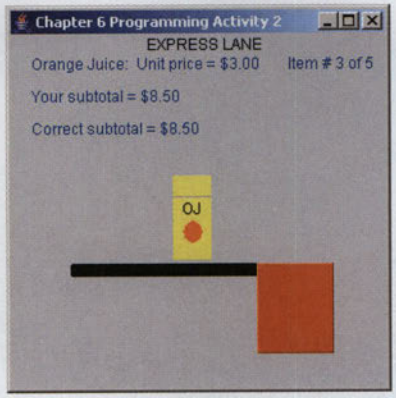
In this activity, you will write a *for* loop:

> For this Programming Activity, we will again calculate the total cost of the items in a grocery cart. This time, however, we will write the program for the Express Lane. In this lane, the customer is allowed up to 10 items. The user will be asked for the number of items in the grocery cart. Your job is to write a *for* loop to calculate the total cost of the items in the cart.

Like Programming Activity 1, the framework will animate your *for* loop, displaying the items in the cart moving down a conveyor belt toward a cashier station (a grocery bag). It will also display the unit price of the item, the correct subtotal, and your current subtotal. By comparing the correct subtotal to your subtotal, you will be able to check whether your code is calculating the correct value.

Figure 6.30 demonstrates the animation. The cart contains five items. The third item, a carton of orange juice, is being scanned at a unit price of $3.00, bringing the correct subtotal for the cart to $8.50.

Figure 6.30
Sample Animation



Instructions

Copy the files in the Chapter 6 Programming Activity 2 directory on the CD-ROM accompanying this book to a directory on your computer. Searching for five stars (*****) in the *Cashier.java* code will show you where to add your code. You will add your code inside the *checkout* method of the *Cashier* class (the method header for the *checkout* method has already been coded for you). Example 6.21 shows a fragment of the *Cashier* class, where you will add your code:

```
public void checkout( int numberOfItems )
{
   /* ***** Student writes the body of this method ***** */
   //
   //  The parameter of this method, numberOfItems,
   //  represents the number of items in the cart. The
   //  user will be prompted for this number.
   //
   //  Using a for loop, calculate the total price
   //  of the groceries for the cart.
   //
   //  The getNext method (in this Cashier class) returns the next
   //  item in the cart, which is an Item object (we do not
   //  know which item will be returned; this is randomly generated).
   //  getNext does not take any arguments. Its API is
   //      Item getNext( )
   //
```

```
    // As the last statement of the body of your for loop,
    // you should call the animate method.
    // The animate method takes one parameter:  a double,
    // which is your current subtotal.
    // For example, if the name of your variable representing
    // the current subtotal is total, your call to the animate
    // method should be:
    //     animate( total );
    //
    // The getPrice method of the Item class
    // returns the price of the Item object as a double.
    // The getPrice method does not take any arguments. Its API is
    //    double getPrice( )
    //
    // After you have processed all the items, display the total
    // for the cart in a dialog box.


    //
    // End of student code
    //
}
```

EXAMPLE 6.21 The *checkout* Method in the *Cashier* Class

To write the body of your *for* loop, you can use the following methods:

- You can access items in the cart using the *getNext* method of the *Cashier* class, which has the following API:

  ```
  Item getNext( )
  ```

  The *getNext* method returns an *Item* object, which represents an *Item* in the cart. As you can see, the *getNext* method does not take any arguments. Since we call the method *getNext* from inside the *Cashier* class, we can simply call the method without an object reference. For example, a call to *getNext* could look like the following:

  ```
  Item newItem;

  newItem = getNext( );
  ```

- After you get a new *Item*, you can "scan" the item to get its price by calling the *getPrice* method of the *Item* class. The *getPrice* method has this API:

```
double getPrice( )
```

Thus, you would get the next item, then get its price using code like the following:

```
Item newItem;
double price;

newItem = getNext( );
price = newItem.getPrice( );
```

When you have finished writing the code for the *checkout* method, compile and run the application from the *Cashier* class. When the application finishes executing, verify that your code is correct by:

- checking that your subtotal matches the correct subtotal displayed
- checking that you have processed all the items in the cart by verifying that the current item number matches the total number of items. For example, if the cart has five items, check that the message in the upper-right corner of the screen displays: `Item # 5 of 5`.

Troubleshooting

If your method implementation does not animate or animates incorrectly, check these items:

- Verify that you have correctly coded the header of your *for* loop.
- Verify that you have correctly coded the body of the loop.

DISCUSSION QUESTIONS ?

1. Explain why a *for* loop is appropriate for this activity.
2. Explain how you set up your *for* loop; that is, what initialization statement did you use, what was your condition, and what was the loop update statement?

6.13 Chapter Summary

- Looping repeats a set of operations for each input item while a condition is *true*.
- The *while* loop is especially useful for event-controlled looping. The *while* loop executes a set of operations in the loop body as long as the loop condition is *true*. Each execution of the loop body is an iteration of the loop.
- If the loop condition evaluates to *false* the first time it is evaluated, the body of the *while* loop is never executed.
- If the loop condition never evaluates to *false*, the result is an infinite loop.
- In event-controlled looping, processing of items continues until the end of input is signaled either by a sentinel value or by reaching the end of the file.
- A sentinel value is a special input value that signals the end of the items to be processed. With a sentinel value, we perform a priming read before the *while* loop. The body of the loop processes the input, then performs an update read of the next data item.
- When reading data from an input file, we can test whether we have reached the end of the file by calling a *hasNext* method of the *Scanner* class.
- In the accumulation programming technique, we initialize a total variable to 0 before starting the loop. In the loop body, we add each input value to the total. When the loop completes, the current total is the total for all processed input values.
- In the counting programming technique, we initialize a count variable to 0 before starting the loop. In the loop body, we increment the count variable for each input value that meets our criteria. When the loop completes, the count variable contains the number of items that met our criteria.
- To find an average, we combine accumulation and counting. We add input values to the total and increment the count. When the

CHAPTER SUMMARY

loop completes, we calculate the average by dividing the total by the count. Before computing the average, however, we should verify that the divisor (that is, the count) is not 0.

- To find the maximum or minimum values in a set of input, we assign the first input to a running maximum or minimum. In the loop body, we compare each input value to our running maximum or minimum. If the input value is less than the running minimum, we assign the input value to the running minimum. Similarly, if the input value is greater than the running maximum, we assign the input value to the running maximum. When the loop completes, the running value is the maximum or minimum value of all the input values.
- To animate an image, the loop body draws the image, pauses for a short interval, erases the image, and changes the starting *x* or *y* values to the next location for drawing the image.
- To avoid generating exceptions when the user types characters other than the data type expected, use the *hasNext* methods of the *Scanner* class.
- To construct a loop condition, construct the inverse of the loop termination condition.
- When testing a program that contains a loop, test that the program produces correct results by inputting values and comparing the results with manual calculations. Also test that the results are correct if the *while* loop body never executes. Finally, test the results with input that is invalid.
- The *do/while* loop checks the loop condition after executing the loop body. Thus, the body of a *do/while* loop always executes at least once. This type of loop is useful for validating input.
- The *for* loop is useful for count-controlled loops, that is, loops for which the number of iterations is known when the loop begins.
- When the *for* loop is encountered, the initialization statement is executed. Then the loop condition is evaluated. If the condition is *true*, the loop body is executed. The loop update statement is then executed and the loop condition is reevaluated. Again, if the condition is *true*, the loop body is executed, followed by the loop update,

then the reevaluation of the condition, and so on, until the condition evaluates to *false*.

- Typically, we use a loop counting variable in a *for* loop. We set its initial value in the initialization statement, increment or decrement its value in the loop update statement, and check its value in the loop condition.
- The loop update statement can increment or decrement the loop variable by any value.
- In a *for* loop, it is important to test that the starting and ending values of the loop variable are correct. Also test with input for which the *for* loop body does not execute at all.

6.14 Exercises, Problems, and Projects

6.14.1 Multiple Choice Exercises

1. How do you discover that you have an infinite loop in your code?
   - ❑ The code does not compile.
   - ❑ The code compiles and runs but gives the wrong result.
   - ❑ The code runs forever.
   - ❑ The code compiles, but there is a runtime error.
2. If you want to execute a loop body at least once, what type of loop would you use?
   - ❑ *for* loop
   - ❑ *while* loop
   - ❑ *do/while* loop
   - ❑ none of the above
3. What best describes a *for* loop?
   - ❑ It is a count-controlled loop.
   - ❑ It is an event-controlled loop.
   - ❑ It is a sentinel-controlled loop.
4. You can simulate a *for* loop with a *while* loop.
   - ❑ true
   - ❑ false

6.14.2 Reading and Understanding Code

5. What is the output of this code sequence? (The user successively enters 3, 5, and –1.)

```
System.out.print( "Enter an int > " );
int i = scan.nextInt( );
while ( i != -1 )
{
   System.out.println( "Hello" );

   System.out.print( "Enter an int > " );
   i = scan.nextInt( );
}
```

6. What is the output of this code sequence? (The user successively enters 3, 5, and –1.)

```
int i = 0;
while ( i != -1 )
{
    System.out.println( "Hello" );
    System.out.print( "Enter an int > " );
    i = scan.nextInt( );
}
```

7. What is the output of this code sequence? (The user successively enters 3, 5, and –1.)

```
System.out.print( "Enter an int > " );
int i = scan.nextInt( );
while ( i != -1 )
{
   System.out.print( "Enter an int > " );
   i = scan.nextInt( );

   System.out.println( "Hello" );
}
```

8. What are the values of *i* and *sum* after this code sequence is executed?

```
int sum  = 0;
int i = 17;
while ( i % 10 != 0 )
{
  sum += i;
  i++;
}
```

9. What are the values of *i* and *product* after this code sequence is executed?

```
int i = 6;
int product = 1;
do
{
    product *= i;
    i++;
} while ( i < 9 );
```

10. What are the values of *i* and *product* after this code sequence is executed?

```
int i = 6;
int product = 1;
do
{
    product *= i;
    i++;
} while ( product < 9 );
```

11. What is the output of this code sequence?

```
for ( int i = 0; i < 3; i++ )
   System.out.println( "Hello" );
System.out.println( "Done" );
```

12. What is the output of this code sequence?

```
for ( int i = 0; i <= 2; i++ )
   System.out.println( "Hello" );
System.out.println( "Done" );
```

13. What is the value of *i* after this code sequence is executed?

```
int i = 0;
for ( i = 0; i <= 2; i++ )
   System.out.println( "Hello" );
```

14. What is the value of *i* after this code sequence is executed?

```
int i = 0;
for ( i = 0; i < 2034; i++ )
   System.out.println( "Hello" );
```

15. What are the values of *i* and *sum* after this code sequence is executed?

```
int i = 0;
int sum  = 0;
for ( i = 0; i < 5; i++ )
{
   sum += i;
}
```

16. What are the values of *i* and *sum* after this code sequence is executed?

```
int i = 0;
int sum  = 0;
for ( i = 0; i < 40; i++ )
{
   if ( i % 10 == 0 )
       sum += i;
}
```

17. What is the value of *sum* after this code sequence is executed?

```
int sum  = 0;
for ( int i = 1; i < 10; i++ )
{
    i++;
    sum += i;
}
```

18. What is the value of *sum* after this code sequence is executed?

```
int sum  = 0;
for ( int i = 10; i > 5; i-- )
{
    sum += i;
}
```

19. What is printed when this code sequence is executed?

```
for ( int i = 0; i < 5; i++ )
{
   System.out.println( Math.max( i, 3 ) );
}
```

20. What are the values of *i* and *sum* after this code sequence is executed?

```
int i = 0;
int sum  = 0;
while ( i != 7 )
{
   sum += i;
   i++;
}
```

6.14.3 Fill In the Code

21. This *while* loop generates random integers between 3 and 7 until a 5 is generated and prints them all out, excluding 5.

```
Random random = new Random( );
int i =  random.nextInt( 5 ) + 3;
```

22. This *while* loop takes an integer input from the user, then prompts for additional integers and prints all integers that are greater than or equal to the original input until the user enters 20, which is not printed.

```
System.out.print( "Enter a starting integer > " );

int start = scan.nextInt( );

// your code goes here
```

23. This *while* loop takes integer values as input from the user and finds the sum of those integers until the user types in the value –1 (which is not added).

```
System.out.print( "Enter an integer value, "
                  + "enter -1 to stop > " );
int value = scan.nextInt( );
// your code goes here
```

24. This loop calculates the sum of the first four positive multiples of 7 using a *while* loop (the sum will be equal to 7 + 14 + 21 + 28 = 70).

```
int sum = 0;
int countMultiplesOf7 = 0;
int count = 1;
// your code goes here
```

25. This loop takes words as input from the user and concatenates them until the user types in the word "end" (which is not concatenated). The code then outputs the concatenated *String*.

```
String sentence = "";
String word;
// your code goes here

while ( ! word.equals( "end" ) )
{
    // and your code goes here

}
System.out.println( "The sentence is " + sentence );
```

26. This loop reads integers from a file (already associated with the *Scanner* object reference *scan*) and computes the sum. We don't know how many integers are in the file.

```
int sum = 0;
// your code goes here
```

27. Here is a *while* loop; write the equivalent *for* loop.

```
int i = 0;
while ( i < 5 )
{
   System.out.println( "Hi there" );
   i++;
}

// your code goes here
```

28. This loop reads integers from the user until the user enters either 0 or 100. Then it prints the sum of the numbers entered (excluding the 0 or 100).

```
// your code goes here
```

29. This loop calculates the sum of the integers from 1 to 5 using a *for* loop.

```
int sum = 0;
// your code goes here
```

6.14.4 Identifying Errors in Code

30. Where is the problem with this code sequence (although this code sequence does compile)?

```
int i = 0;
while ( i < 3 )
   System.out.println( "Hello" );
```

31. Where is the error in this code sequence that is supposed to read and echo integers until the user enters –1?

```
int num;
while ( num != -1 )
{
   System.out.print( "Enter an integer > " );
   num = scan.nextInt( );
   System.out.println( num );
}
```

32. The following code sequence intends to print *Hello* three times; however, it does not. Where is the problem in this code sequence?

```
for ( int i = 0; i < 3; i++ );
   System.out.println( "Hello" );
```

33. Where is the error in this code sequence, which is intended to print *Hello* 10 times?

```
for ( int i = 10; i > 0; i++ )
   System.out.println( "Hello" );
```

34. Where is the problem with this code sequence? The code is intended to generate random numbers between 1 and 10 until the number is either a 7 or a 5.

```
Random random = new Random( );
int number  =  1 + random.nextInt( 10 );
while ( number != 5 || number != 7 )
{
    number  =  1 + random.nextInt( 10 );
}
System.out.println( "The number is " + number );
```

35. Where is the error with this code sequence?

```
int sum = 0;
for ( int i = 1; i < 6; i++ )
     sum += i;

System.out.println( "The value of i is " + i );
```

6.14.5 Debugging Area—Using Messages from the Java Compiler and Java JVM

36. You coded the following in the class *Test.java*:

```
int i = 0;
int sum = 0;
do
{
     sum += i;
     i++;
} while ( i < 3 )    // line 11
```

At compile time, you get the following error:

```
Test.java:11: ';' expected
while( i < 3 )   // line 10
              ^
1 error
```

Explain what the problem is and how to fix it.

37. You coded the following in the class *Test.java*:

```
int i = 0;
while ( i < 3 )
{
   System.out.println( "Hello" );
   i--;
}
```

The code compiles but never terminates.

Explain what the problem is and how to fix it.

38. You coded the following in the class *Test.java*:

```
for ( int i = 0; i++; i < 3 )      // line 5
   System.out.println( "Hello" );
```

At compile time, you get the following error:

```
Test.java:5: not a statement
for ( int i = 0; i++; i < 3 )      // line 5
                      ^
1 error
```

Explain what the problem is and how to fix it.

39. You coded the following in the class *Test.java*:

```
for ( int i = 1; i < 3; i++ )          // line 5
   System.out.println( "Hello" );
```

The code compiles and runs, but only prints *Hello* twice, whereas we expected to print *Hello* three times.

Explain what the problem is and how to fix it.

40. You coded the following in the class *Test.java*:

```
int product = 1;
for ( int i = 1, i < 5, i++ )          // line 8
   product *= i;
System.out.println( "Product is " + product ); // line 10
```

At compile time, you get the following errors:

```
Test.java:8: ';' expected
  for ( int i = 1, i < 5, i++ )        // line 8
                  ^
Test.java:8: illegal start of type
  for ( int i = 1, i < 5, i++ )        // line 8
                   ^
Test.java:8: illegal start of expression
    for( int i = 1, i < 5, i++ )       // line 8
                    ^
Test.java:8: ';' expected
    for( int i = 1, i < 5, i++ )       // line 8
                      ^
Test.java:8: illegal start of expression
    for( int i = 1, i < 5, i++ )       // line 8
                       ^
5 errors
```

Explain what the problem is and how to fix it.

41. You coded the following in the class *Test.java*:

```
for ( int i = 0; i < 3; i++ )
   System.out.println( "Hello" );
System.out.println( "i = " + i );   // line 8
```

At compile time, you get the following error:

```
Test.java:8: cannot find symbol
      System.out.println( "i = " + i ); // line 8
                                  ^
symbol  : variable i
location: class Test
1 error
```

Explain what the problem is and how to fix it.

42. You coded the following in the class *Test.java*:

```
int i = 0;
for ( int i = 0; i < 3; i++ ) // line 6
   System.out.println( "Hello" );
```

At compile time, you get the following error:

```
Test.java:6: i is already defined in main( java.lang.String[] )
for( int i = 0; i < 3; i++ )       // line 6
       ^
1 error
```

Explain what the problem is and how to fix it.

EXERCISES, PROBLEMS, AND PROJECTS

6.14.6 Write a Short Program

43. Write a program that prompts the user for a value greater than 10 as an input (you should loop until the user enters a valid value) and finds the square root of that number and the square root of the result, and continues to find the square root of the result until we reach a number that is smaller than 1.01. The program should output how many times the square root operation was performed.

44. Write a program that expects a word containing the @ character as an input. If the word does not contain an @ character, then your program should keep prompting the user for a word. When the user types in a word containing an @ character, the program should simply print the word and terminate.

45. Write a program that reads *double* values from a file named *input.txt* and outputs the average.

46. Write a program that uses a *for* loop to output the sum of all the integers between 10 and 20, inclusive, that is, $10 + 11 + 12 + \ldots + 19 + 20$.

47. Write a program that uses a *for* loop to output the product of all the integers between 3 and 7, inclusive, that is, $3 * 4 * 5 * 6 * 7$.

48. Write a program that uses a *for* loop to count how many multiples of 7 are between 33 and 97, inclusive.

49. Write a program that reads a value (say n) from the user and outputs *Hello World* n times. Verify that the user has entered an integer. If the input is 3, the output will be *Hello World* printed three times.

50. Write a program that takes a word as an input from the keyboard and outputs each character in the word, separated by a space.

51. Write a program that takes a value as an input from the keyboard and outputs the factorial of that number; the factorial of an integer n is $n * (n-1) * (n-2) * \ldots * 3 * 2 * 1$. For instance, the factorial of 4 is $4 * 3 * 2 * 1$, or 24.

52. Using a loop, write a program that takes 10 integer values from the keyboard and outputs the minimum value of all the values entered.

53. Write an applet that displays a rectangle moving horizontally from the right side of the window to the left side of the window.

6.14.7 Programming Projects

54. Write a program that inputs a word representing a binary number (0s and 1s). First, your program should verify that it is indeed a binary number, that is, the number contains only 0s and 1s. If that is not the case, your program should print a message that the number is not a valid binary number. Then, your program should count how many 1s are in that word and output the count.

55. Perform the same operations as Question 54, with the following modification: If the word does not represent a valid binary number, the program should keep prompting the user for a new word until a word representing a valid binary number is input by the user.

56. Write a program that inputs a word representing a binary number (0s and 1s). First, your program should check that it is indeed a binary number, that is, the number contains only 0s and 1s. If that is not the case, your program should output that the number is not a valid binary number. If that word contains exactly two 1s, your program should output that that word is "accepted," otherwise that it is "rejected."

57. Perform the same operations as Question 56, with the following modification: If the word does not represent a valid binary number, the program should keep prompting the user for a new word until a word representing a valid binary number is input by the user.

58. Write a program that inputs a word representing a binary number (0s and 1s). First, your program should check that it is indeed a binary number, that is, that it contains only 0s and 1s. If that is not the case, your program should output that the number is not a valid binary number. If that word contains at least three consecutive 1s, your program should output that that word is "accepted," otherwise that it is "rejected."

59. Perform the same operations as Question 58 with the following modification: If the word does not represent a valid binary number, the program should keep prompting the user for a new word until a word representing a valid binary number is input by the user.

60. Write a program that takes website names as keyboard input until the user types the word *stop* and counts how many of the website names

are commercial website names (i.e., end with *.com*), then outputs that count.

61. Using a loop, write a program that takes 10 values representing exam grades (between 0 and 100) from the keyboard and outputs the minimum value, maximum value, and average value of all the values entered. Your program should not accept values less than 0 or greater than 100.

62. Write a program that takes an email address as an input from the keyboard and, using a loop, steps through every character looking for an @ sign. If the email address has exactly one @ character, then print a message that the email address is valid; otherwise, print a message that it is invalid.

63. Write a program that takes a user ID as an input from the keyboard and steps through every character, counting how many digits are in the user ID; if there are exactly two digits, output that the user ID is valid, otherwise that it is invalid.

64. Write a program that takes an integer value as an input and converts that value to its binary representation; for instance, if the user inputs 17, then the output will be 10001.

65. Write a program that takes a word representing a binary number (0s and 1s) as an input and converts it to its decimal representation; for instance, if the user inputs 101, then the output will be 5; you can assume that the *String* is guaranteed to contain only 0s and 1s.

66. Write a program that simulates an XOR operation. The input should be a word representing a binary number (0s and 1s). Your program should XOR all the digits from left to right and output the results as "True" or "False." In an XOR operation, *a* XOR *b* is *true* if *a* or *b* is *true* but not both; otherwise, it is *false*. In this program, we will consider the character "1" to represent true and a "0" to represent false. For instance, if the input is 1011, then the output will be 1 (1 XOR 0 is 1, then 1 XOR 1 is 0, then 0 XOR 1 is 1, which causes the output to be "True"). You can assume that the input word is guaranteed to contain only 0s and 1s.

67. Write a program that takes a sentence as an input (using a dialog box) and checks whether that sentence is a palindrome. A palindrome is a

word, phrase, or sentence that is symmetrical; that is, it is spelled the same forward and backward. Examples are "otto," "mom," and "Able was I ere I saw Elba." Your program should be case-insensitive; that is, "Otto" should also be counted as a palindrome.

68. Write a program that takes an HTML-like sequence as an input (using a dialog box) and checks whether that sequence has the same number of opening brackets (<) and closing brackets (>).

69. Write an applet that shows a small circle getting bigger and bigger. Your applet should allow the user to input the starting radius and the ending radius (and also verify that the starting radius is smaller than the ending radius).

6.14.8 Technical Writing

70. In programming, a programmer can make syntax errors that lead to a compiler error; these errors can then be corrected. Other errors can lead to a runtime error; these errors can also be corrected. Logic errors, however, can lead to an incorrect result or no result at all. Discuss examples of logic errors that can be made when coding loops and the consequences of these logic errors.

71. Discuss how you would detect whether you have an infinite loop in your code.

6.14.9 Group Project (for a group of 1, 2, or 3 students)

72. Often on a web page, the user is asked to supply personal information, such as a telephone number. Your program should take an input from the keyboard representing a telephone number. We will consider that the input is a valid telephone number if it contains exactly 10 digits and any number of dash (-) and whitespace characters. Keep prompting the user for a telephone number until the user gives you a valid one. Once you have a valid telephone number, you should assume that the digits (only the digits, not the hyphen[s] nor the whitespace) in the telephone number may have been encrypted by shifting each number by a constant value. For instance, if the shift is 2, a 0 becomes a 2, a 1 becomes a 3, a 2 becomes a 4, . . . , an 8 becomes a 0, and a 9 becomes a 1. However, we know that the user is from New York where the decrypted area code (after the shift is applied), represented by the first three digits of the input, is 212. Your

program needs to decrypt the telephone number and output the decrypted telephone number with the format 212-xxx-xxxx, as well as the shift value of the encryption. If there was an error in the input and the area code cannot be decrypted to 212, you should output that information.