CHAPTER 6

# Looping

In this chapter, you will:

- Learn about the loop structure
- Create `while` loops
- Use shortcut arithmetic operators
- Create `for` loops
- Create `do…while` loops
- Nest loops
- Improve loop performance

Unless noted otherwise, all images are © 2014 Cengage Learning

## Learning About the Loop Structure

If making decisions is what makes programs seem smart, looping is what makes programs seem powerful. A **loop** is a structure that allows repeated execution of a block of statements. Within a looping structure, a Boolean expression is evaluated. If it is `true`, a block of statements called the **loop body** executes and the Boolean expression is evaluated again. The loop body can be a single statement or a block of statements between curly braces. As long as the expression is `true`, the statements in the loop body continue to execute. When the Boolean evaluation is `false`, the loop ends. One execution of any loop is called an **iteration**. Figure 6-1 shows a diagram of the logic of a loop.

In Java, you can use several mechanisms to create loops. In this chapter, you learn to use three types of loops:

- A `while` loop, in which the loop-controlling Boolean expression is the first statement in the loop
- A `for` loop, which is usually used as a concise format in which to execute loops
- A `do...while` loop, in which the loop-controlling Boolean expression is the last statement in the loop

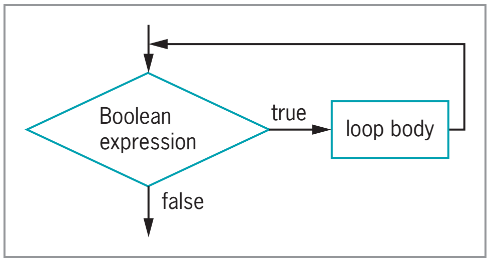


**Figure 6-1** Flowchart of a loop structure

**TWO TRUTHS & A LIE**

**Learning About the Loop Structure**

1. A loop is a structure that allows repeated execution of a block of statements as long as a tested expression is `true`.
2. If a loop's tested Boolean expression is `true`, a block of statements called the loop body executes before the Boolean expression is evaluated again.
3. When the Boolean evaluation tested in a loop becomes `false`, the loop body executes one last time.

The false statement is #3. When the Boolean evaluation tested in a loop is `false`, the loop ends.

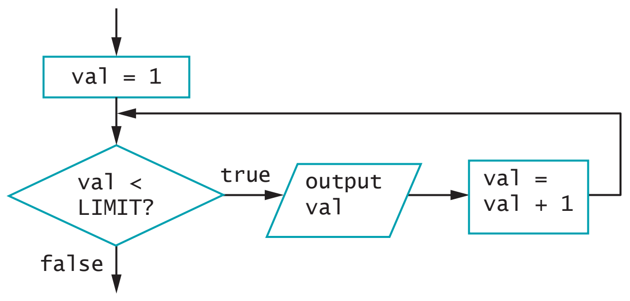
## Creating while Loops

You can use a **while loop** to execute a body of statements continually as long as the Boolean expression that controls entry into the loop continues to be `true`. In Java, a `while` loop consists of the keyword `while` followed by a Boolean expression within parentheses, followed by the body of the loop.

You can use a `while` loop when you need to perform a task either a predetermined or unpredictable number of times. A loop that executes a specific number of times is a **definite loop**, or a counted loop. On the other hand, the number of times the loop executes might not be determined until the program is running. Such a loop is an **indefinite loop** because you don't know how many times it will eventually loop.

### Writing a Definite while Loop

To write a definite loop, you initialize a **loop control variable**, a variable whose value determines whether loop execution continues. While the Boolean value that results from comparing the loop control variable and another value is `true`, the body of the `while` loop continues to execute. In the body of the loop, you must include a statement that alters the loop control variable. For example, the program segment shown in Figure 6-2 displays the series of integers 1 through 10. The variable `val` is the loop control variable—it starts the loop holding a value of 1, and while the value remains under 11, `val` continues to be output and increased.

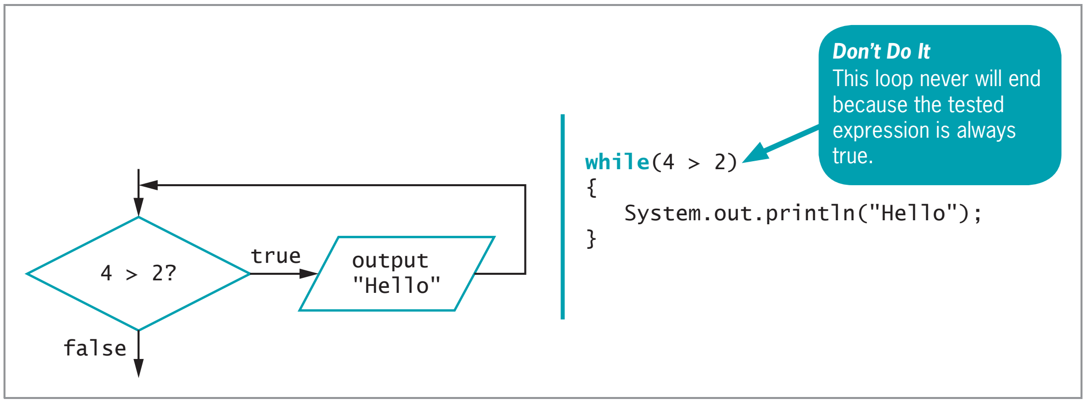


```
int val;
final int LIMIT = 11;

val = 1;
while(val < LIMIT)
{
   System.out.println(val);
   val = val + 1;
}
```

Figure 6-2 A while loop that displays the integers 1 through 10

When you write applications containing loops, it is easy to make mistakes. For example, executing the code shown in Figure 6-3 causes the message "Hello" to be displayed forever (theoretically) because there is no code to end the loop. A loop that never ends is called an **infinite loop**.


4 > 2?
true
output "Hello"
false
Don't Do It
This loop never will end because the tested expression is always true.

```
while(4 > 2)
{
   System.out.println("Hello");
}
```

**Figure 6-3** A loop that displays "Hello" infinitely

An infinite loop might not actually execute infinitely. Depending on the tasks the loop performs, eventually the computer memory might be exhausted (literally and figuratively) and execution might stop. Also, it's possible that the processor has a time-out feature that forces the loop to end. Either way, and depending on your system, quite a bit of time could pass before the loop stops running.

As an inside joke to programmers, the address of Apple Computer, Inc. is One Infinite Loop, Cupertino, California.

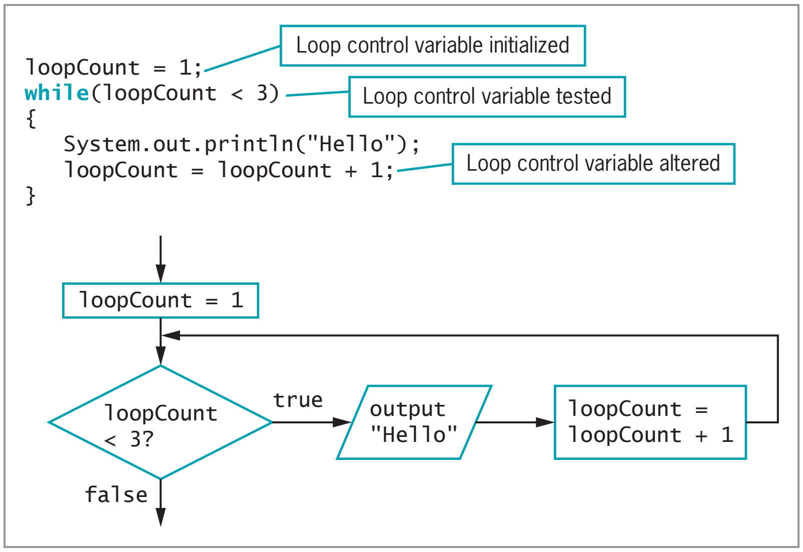
In Figure 6-3, the expression `4 > 2` evaluates to `true`. You obviously never need to make such an evaluation, but if you do so in this `while` loop, the body of the loop is entered and "Hello" is displayed. Next, the expression is evaluated again. The expression `4 > 2` is still `true`, so the body is entered again. "Hello" is displayed repeatedly; the loop never finishes because `4 > 2` is never `false`.

It is a bad idea to write an infinite loop intentionally. However, even experienced programmers write them by accident. So, before you start writing loops, it is good to know how to exit from an infinite loop. You might suspect an infinite loop if the same output is displayed repeatedly, or if the screen simply remains idle for an extended period of time without displaying expected output. If you think your application is in an infinite loop, you can press and hold Ctrl, and then press C or Break; the looping program should terminate. (On many keyboards, the Break key is also the Pause key.)

To prevent a `while` loop from executing infinitely, three separate actions must occur:

- A loop control variable is initialized to a starting value.
- The loop control variable is tested in the `while` statement.
- The loop control variable is altered within the body of the loop. The variable must be altered so that the test expression can eventually evaluate to `false` and the loop can end.

All of these conditions are met by the example in Figure 6-4. First, a loop control variable `loopCount` is named and set to a value of 1. Second, the statement `while(loopCount < 3)` is tested. Third, the loop body is executed because the loop control variable `loopCount` is less than 3. Note that the loop body shown in Figure 6-4 consists of two statements made into a block by their surrounding curly braces. The first statement displays "Hello," and then the second statement adds 1 to `loopCount`. The next time `loopCount` is evaluated, it is 2. It is still less than 3, so the loop body executes again. "Hello" is displayed a second time, and `loopCount` becomes 3. Finally, because the expression `loopCount < 3` now evaluates to `false`, the loop ends. Program execution then continues with any subsequent statements.



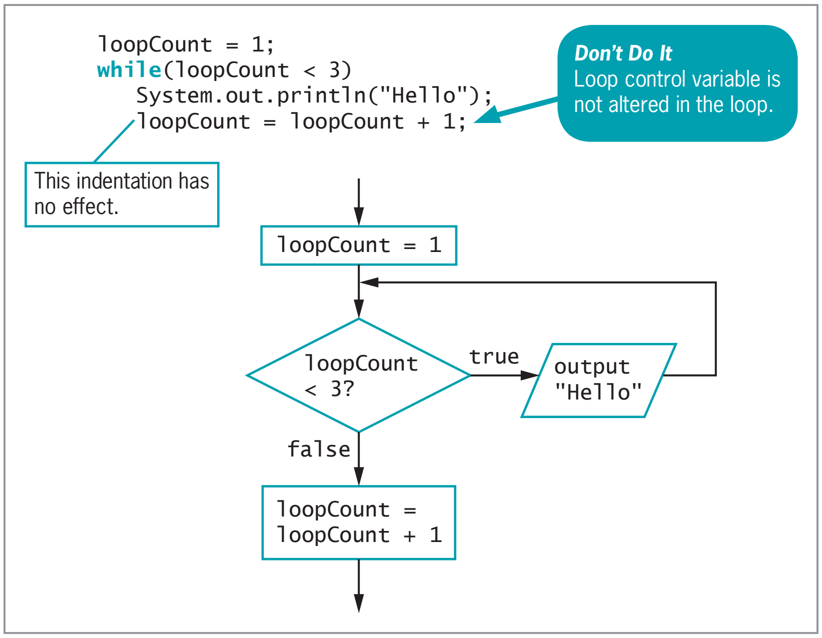
```
loopCount = 1;
while(loopCount < 3)
{
   System.out.println("Hello");
   loopCount = loopCount + 1;
}
```

Loop control variable initialized
Loop control variable tested
Loop control variable altered
loopCount = 1
loopCount < 3?
true
output "Hello"
loopCount = loopCount + 1
false

**Figure 6-4** A `while` loop that displays "Hello" twice

## Pitfall: Failing to Alter the Loop Control Variable Within the Loop Body

It is important that the loop control variable be altered within the body of the loop. Figure 6-5 shows the same code as in Figure 6-4, but the curly braces have been eliminated. In this case, the `while` loop body ends at the semicolon that appears at the end of the "Hello" statement. Adding 1 to the `loopCount` is no longer part of a block that contains the loop, so the value of `loopCount` never changes, and an infinite loop is created.



```
loopCount = 1;
while(loopCount < 3)
   System.out.println("Hello");
   loopCount = loopCount + 1;
```

Don't Do It
Loop control variable is not altered in the loop.
This indentation has no effect.
loopCount = 1
loopCount < 3?
true
output "Hello"
false
loopCount = loopCount + 1

**Figure 6-5** A `while` loop that displays "Hello" infinitely because `loopCount` is not altered in the loop body

## Pitfall: Creating a Loop with an Empty Body

As with the decision-making `if` statement that you learned about in Chapter 5, placement of the statement-ending semicolon is important when you work with the `while` statement. If a semicolon is mistakenly placed at the end of the partial statement `while(loopCount < 3);`, as shown in Figure 6-6, the loop is also infinite. This loop has an **empty body**, or a body with no statements in it. So, the Boolean expression is evaluated, and because it is `true`, the loop body is entered. Because the loop body is empty, no action is taken, and the Boolean expression is evaluated again. Nothing has changed, so it is still `true`, the empty body is entered, and the infinite loop continues.

```
loopCount = 1;
while(loopCount < 3);
{
   System.out.println("Hello");
   loopCount = loopCount + 1;
}
```

***Don't Do It***
This semicolon causes the loop to have an empty body.

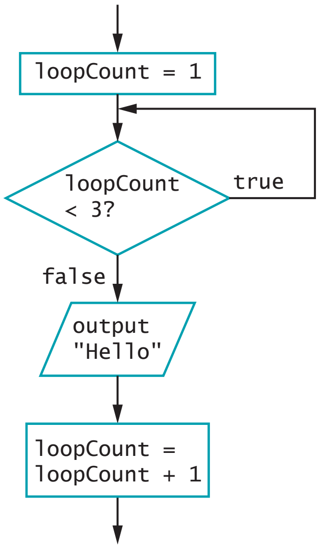


**Figure 6-6** A while loop that loops infinitely with no output because the loop body is empty

## Altering a Definite Loop's Control Variable

A definite loop is a **counter-controlled loop** because the loop control variable is changed by counting. It is very common to alter the value of a loop control variable by adding 1 to it, or **incrementing** the variable. However, not all loops are controlled by adding 1. The loop shown in Figure 6-7 displays "Hello" twice, just as the loop in Figure 6-4 does, but its loop is controlled by subtracting 1 from a loop control variable, or **decrementing** it.

```
loopCount = 3;
while(loopCount > 1)
{
   System.out.println("Hello");
   loopCount = loopCount - 1;
}
```

**Figure 6-7** A while loop that displays "Hello" twice, decrementing the loopCount variable in the loop body

In the program segment shown in Figure 6-7, the variable `loopCount` begins with a value of 3. The `loopCount` is greater than 1, so the loop body displays "Hello" and decrements `loopCount` to 2. The Boolean expression in the `while` loop is tested again. Because 2 is more than 1, "Hello" is displayed again, and `loopCount` becomes 1. Now `loopCount` is not greater than 1, so the loop ends. There are many ways to execute a loop two times. For example, you can initialize a loop control variable to 10 and continue while the value is greater than 8, decreasing the value by 1 each time you pass through the loop. Similarly, you can initialize the loop control variable to 12, continue while it is greater than 2, and decrease the value by 5 each time. In general, you should not use such unusual methods to count repetitions because they simply make a program confusing. To execute a loop a specific number of times, the clearest and best method is to start the loop control variable at 0 or 1, increment by 1 each time through the loop, and stop when the loop control variable reaches the appropriate limit.

When you first start programming, it seems reasonable to initialize counter values to 1, and that is a workable approach. However, many seasoned programmers start counter values at 0 because they are used to doing so when working with arrays. When you study arrays in the chapter *Introduction to Arrays*, you will learn that their elements are numbered beginning with 0.

Watch the video *Looping*.

## Writing an Indefinite `while` Loop

You are not required to alter a loop control variable by adding to it or subtracting from it. Often, the value of a loop control variable is not altered by arithmetic, but instead is altered by user input. Instead of being a counter-controlled loop, an indefinite loop is an **event-controlled loop**. That is, an event occurs that determines whether the loop continues. An event-controlled loop is a type of indefinite loop because you don't know how many times it will eventually repeat. For example, perhaps you want to continue performing some task as long as the user indicates a desire to continue by pressing a key. In this case, while you are writing the program, you do not know whether the loop eventually will be executed two times, 200 times, or at all.

Consider an application in which you ask the user for a bank balance and then ask whether the user wants to see the balance after interest has accumulated. Each time the user chooses to continue, an increased balance appears, reflecting one more year of accumulated interest. When the user finally chooses to exit, the program ends. The program appears in Figure 6-8.

```
import java.util.Scanner;
public class BankBalance
{
   public static void main(String[] args)
   {
      double balance;
      int response;
      int year = 1;
      final double INT_RATE = 0.03;
      Scanner keyboard = new Scanner(System.in);
      System.out.print("Enter initial bank balance > ");
      balance = keyboard.nextDouble();
      System.out.println("Do you want to see next year's balance?");
      System.out.print("Enter 1 for yes");
      System.out.print("   or any other number for no >> ");
      response = keyboard.nextInt();
      while(response == 1)
      {
         balance = balance + balance * INT_RATE;
         System.out.println("After year " + year + " at " +  INT_RATE +
             " interest rate, balance is $" + balance);
         year = year + 1;
         System.out.println("\nDo you want to see the balance " +
            "at the end of another year?");
         System.out.print("Enter 1 for yes");
         System.out.print("   or any other number for no >> ");
         response = keyboard.nextInt();
      }
   }
}
```

**Figure 6-8** The BankBalance application

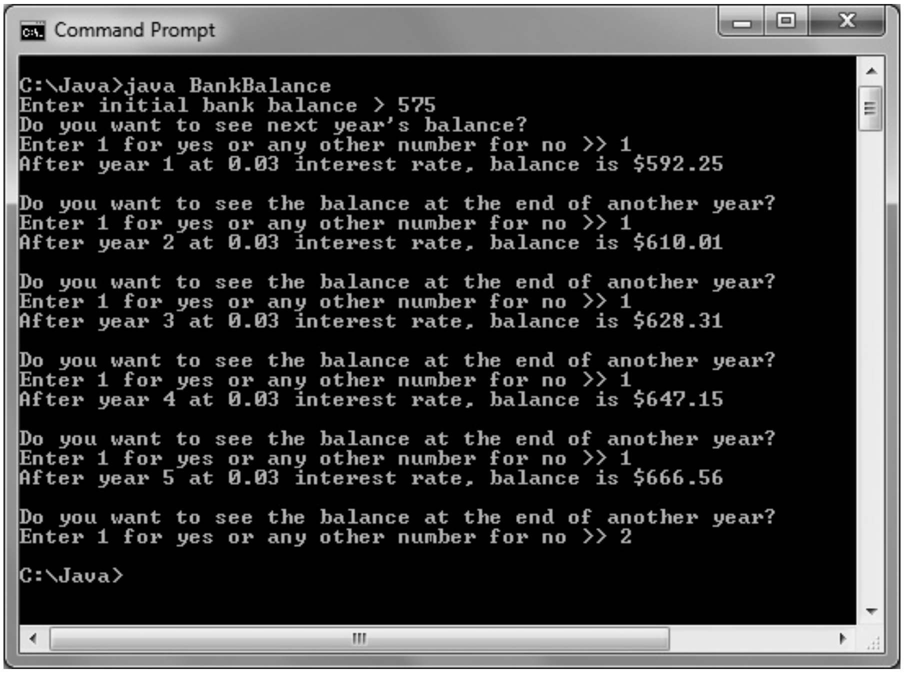
In the BankBalance program, as in any interactive program, the user must enter data that has the expected data types. If not, an error occurs and the program terminates. You will learn to manage user entry errors in the chapter *Exception Handling*.

The program shown in Figure 6-8 declares needed variables and a constant for a 3 percent interest rate, and then asks the user for a balance. The application then asks the user to enter a *1* if the user wants see the next year's balance. As long as the user wants to continue, the application continues to display increasing bank balances.

The loop in the application in Figure 6-8 begins with the line that contains:

```
while(response == 1)
```

If the user enters any integer value other than *1*, the loop body never executes; instead, the program ends. However, if the user enters *1*, all the statements within the loop body execute. The application increases the balance by the interest rate value, displays the new balance, adds 1 to `year`, and asks whether the user wants another balance. The last statement in the loop body accepts the user's response. After the loop body executes, control returns to the top of the loop, where the Boolean expression in the `while` loop is tested again. If the user's response is *1*, the loop is entered and the process begins again. Figure 6-9 shows the output of the `BankBalance` application after the user enters a $575.00 starting balance and responds with *1* five times to the prompt for increased interest payments before responding *2*.



```
C:\Java>java BankBalance
Enter initial bank balance > 575
Do you want to see next year's balance?
Enter 1 for yes or any other number for no >> 1
After year 1 at 0.03 interest rate, balance is $592.25

Do you want to see the balance at the end of another year?
Enter 1 for yes or any other number for no >> 1
After year 2 at 0.03 interest rate, balance is $610.01

Do you want to see the balance at the end of another year?
Enter 1 for yes or any other number for no >> 1
After year 3 at 0.03 interest rate, balance is $628.31

Do you want to see the balance at the end of another year?
Enter 1 for yes or any other number for no >> 1
After year 4 at 0.03 interest rate, balance is $647.15

Do you want to see the balance at the end of another year?
Enter 1 for yes or any other number for no >> 1
After year 5 at 0.03 interest rate, balance is $666.56

Do you want to see the balance at the end of another year?
Enter 1 for yes or any other number for no >> 2

C:\Java>
```

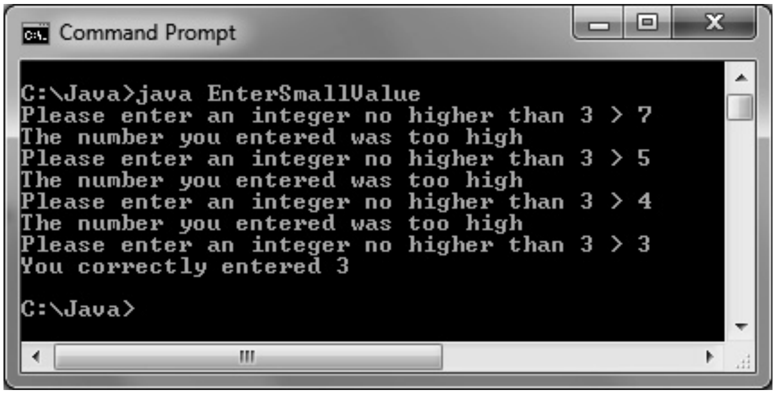
**Figure 6-9** Typical execution of the `BankBalance` application

## Validating Data

Programmers commonly use indefinite loops when validating input data. **Validating data** is the process of ensuring that a value falls within a specified range. For example, suppose you require a user to enter a value no greater than 3. Figure 6-10 shows an application that does not progress past the data entry loop until the user enters a correct value. If the user enters 3 or less at the first prompt, the shaded loop never executes. However, if the user enters a number greater than 3, the shaded loop executes, providing the user with another chance to enter a correct value. While the user continues to enter incorrect data, the loop repeats. Figure 6-11 shows a typical execution.

```
import java.util.Scanner;
public class EnterSmallValue
{
   public static void main(String[] args)
   {
      int userEntry;
      final int LIMIT = 3;
      Scanner input = new Scanner(System.in);
      System.out.print("Please enter an integer no higher than " +
         LIMIT + " > ");
      userEntry = input.nextInt();
      while(userEntry > LIMIT)
      {
         System.out.println("The number you entered was too high");
         System.out.print("Please enter an integer no higher than " +
            LIMIT + " > ");
         userEntry = input.nextInt();
      }
      System.out.println("You correctly entered " + userEntry);
   }
}
```

**Figure 6-10** The `EnterSmallValue` application



```
C:\Java>java EnterSmallValue
Please enter an integer no higher than 3 > 7
The number you entered was too high
Please enter an integer no higher than 3 > 5
The number you entered was too high
Please enter an integer no higher than 3 > 4
The number you entered was too high
Please enter an integer no higher than 3 > 3
You correctly entered 3

C:\Java>
```

**Figure 6-11** Typical execution of the `EnterSmallValue` program

Figure 6-10 illustrates an excellent method for validating input. Before the loop is entered, the first input is retrieved. This first input might be a value that prevents any executions of the loop. This first input statement prior to the loop is called a **priming read** or **priming input**. Within the loop, the last statement retrieves subsequent input values for the same variable that will be checked at the entrance to the loop.

Novice programmers often make the mistake of checking for invalid data using a decision instead of a loop. That is, they ask whether the data is invalid using an `if` statement; if the data is invalid, they reprompt the user. However, they forget that a user might enter incorrect data multiple times. Usually, a loop is the best structure to use when validating input data.

## TWO TRUTHS & A LIE

### Creating `while` Loops

1. A finite loop executes a specific number of times; an indefinite loop is one that never ends.
2. A well-written `while` loop contains an initialized loop control variable that is tested in the `while` expression and then altered in the loop body.
3. In an indefinite loop, you don't know how many times the loop will occur.

The false statement is #1. A loop that executes a specific number of times is a definite loop or a counted loop; a loop that never ends is an infinite loop.

## You Do It

*Writing a Loop to Validate Data Entries*

In Chapter 5, you created an `AssignVolunteer4` application for Sacks Fifth Avenue, a nonprofit thrift shop. The application accepts a donation code and assigns the appropriate volunteer to price the item for sale. Now you add a loop to ensure that a valid code always is entered.

1. Open the **AssignVolunteer4.java** file that you created in Chapter 5. Change the class name to **`AssignVolunteer5`**, and immediately save the file as **AssignVolunteer5.java**.
2. After the input statement that gets a code from the user, but before the `switch` structure that assigns a volunteer, insert the following loop. The loop continues while the input `donationType` is less than the lowest valid code or higher than the highest valid code. (Recall that the values of `CLOTHING_CODE`, `FURNITURE_CODE`, `ELECTRONICS_CODE`, and `OTHER_CODE` are 1 through 4,

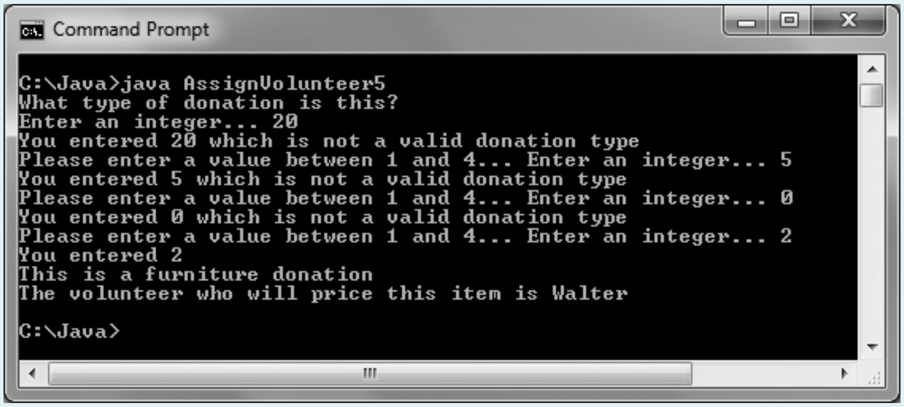
*(continues)*

*(continued)*

respectively.) Within the loop body, statements explain the error to the user and then get a new value for donationType.

```
while(donationType < CLOTHING_CODE || donationType > OTHER_CODE)
{
    System.out.println("You entered " + donationType +
       " which is not a valid donation type");
    System.out.print("Please enter a value between " +
       CLOTHING_CODE + " and " + OTHER_CODE + "... ");
    System.out.print("Enter an integer... ");
    donationType = input.nextInt();
}
```

3. Save the file, and compile and execute it. Figure 6-12 shows a typical execution in which a user enters an invalid code three times before entering a valid one.



```
C:\Java>java AssignVolunteer5
What type of donation is this?
Enter an integer... 20
You entered 20 which is not a valid donation type
Please enter a value between 1 and 4... Enter an integer... 5
You entered 5 which is not a valid donation type
Please enter a value between 1 and 4... Enter an integer... 0
You entered 0 which is not a valid donation type
Please enter a value between 1 and 4... Enter an integer... 2
You entered 2
This is a furniture donation
The volunteer who will price this item is Walter

C:\Java>
```

**Figure 6-12** Typical execution of the AssignVolunteer5 application

4. In the current program, the default case assigns "invalid" to the volunteer. At this point, some professionals would advise that you remove the default case from the case structure because it is no longer possible for the user to enter an invalid code. Others would argue that leaving the default case in place serves two purposes. First, it provides documentation that clarifies the course of action if the entered code does not match any of the listed cases. Second, the program requirements might change in the future. For example, perhaps one of the categories will be eliminated. Then, if you remove the case instructions for that category, a default block will already be in place to handle the new error.

## Using Shortcut Arithmetic Operators

Programmers commonly need to increase the value of a variable in a program. As you saw in the previous section, many loops are controlled by continually adding 1 to some variable, as in `count = count + 1;`. Incrementing a variable in a loop to keep track of the number of occurrences of some event is also known as **counting**. Similarly, in the looping bank balance program shown in Figure 6-8, the program not only incremented the `year` variable by adding 1, it also increased a bank balance by an interest amount with the statement `balance = balance + balance * INT_RATE;`. In other words, the bank balance became its old value *plus* a new interest amount; the process of repeatedly increasing a value by some amount is known as **accumulating**.

Because increasing a variable is so common, Java provides you with several shortcuts for incrementing and accumulating. The statement `count += 1;` is identical in meaning to `count = count + 1`. The += is the **add and assign operator**; it adds and assigns in one operation. Similarly, `balance += balance * INT_RATE;` increases a balance by the `INT_RATE` percentage. Besides using the shortcut operator +=, you can use the **subtract and assign operator** ( `-=` ), the **multiply and assign operator** ( `*=` ), the **divide and assign operator** ( `/=` ), and the **remainder and assign operator** ( %= ). Each of these operators is used to perform the operation and assign the result in one step. For example, `balanceDue -= payment` subtracts `payment` from `balanceDue` and assigns the result to `balanceDue`.

When you want to increase a variable's value by exactly 1, you can use two other shortcut operators—the **prefix ++**, also known as the **prefix increment operator**, and the **postfix ++**, also known as the **postfix increment operator**. To use a prefix ++, you type two plus signs before the variable name. The statement `someValue = 6;` followed by `++someValue;` results in `someValue` holding 7—one more than it held before you applied the ++. To use a postfix ++, you type two plus signs just after a variable name. The statements `anotherValue = 56; anotherValue++;` result in `anotherValue` containing 57. Figure 6-13 shows four ways you can increase a value by 1; each method produces the same result. You are never required to use shortcut operators; they are merely a convenience.

```
int value;
value = 24;
++value;  // Result: value is 25
value = 24;
value++;  // Result: value is 25
value = 24;
value = value + 1;   // Result: value is 25
value = 24;
value += 1;  // Result: value is 25
```

**Figure 6-13** Four ways to add 1 to a value

You can use the prefix ++ and postfix ++ with variables but not with constants. An expression such as `++84;` is illegal because an 84 must always remain an 84. However, you can create a variable named `val`, assign 84 to it, and then write `++val;` or `val++;` to increase the variable's value.

The prefix and postfix increment operators are unary operators because you use them with one value. As you learned in Chapter 2, most arithmetic operators, such as those used for addition and multiplication, are binary operators—they operate on two values. Other examples of unary operators include the cast operator, as well as ( + ) and ( – ) when used to indicate positive and negative values.

When you simply want to increase a variable's value by 1, there is no difference in the outcome, whether you use the prefix or postfix increment operator. For example, when `value` is set to 24 in Figure 6-13, both `++value` and `value++` result in `value` becoming 25; each operator results in increasing the variable by 1. However, when a prefix or postfix operator is used as part of a larger expression, it does make a difference which operator you use because they function differently in terms of what they *return*. When a prefix operator is used in an expression, the value after the calculation is used, but when a postfix operator is used in an expression, the value used is the one before the calculation takes place.

When you use the prefix ++, the result is calculated, and then its value is used. For example, consider the following statements:

```
b = 4;
c = ++b;
```

The result is that both `b` and `c` hold the value 5 because `b` is increased to 5 and then the value of the expression is assigned to `c`.

When you use the postfix ++, the value of the expression before the increase is stored. For example, consider these statements:

```
b = 4;
c = b++;
```

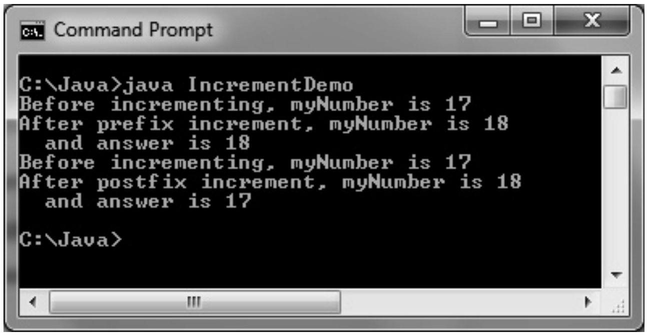
The result is still that `b` is 5, but `c` is only 4. Although `b` is increased, its original value is assigned to `c`.

In other words, if `b = 4`, the value of `b++` is also 4, but after the statement is completed, the value of `b` is 5.

Figure 6-14 shows an application that illustrates the difference between how the prefix and postfix increment operators work. Notice from the output in Figure 6-15 that when the prefix increment operator is used on `myNumber`, the value of `myNumber` increases from 17 to 18, and the result is stored in `answer`, which also becomes 18. After the value is reset to 17, the postfix increment operator is used; 17 is assigned to `answer`, and `myNumber` is incremented to 18.

```
public class IncrementDemo
{
   public static void main(String[] args)
   {
      int myNumber, answer;
      myNumber = 17;
      System.out.println("Before incrementing, myNumber is " +
         myNumber);
      answer = ++myNumber;
      System.out.println("After prefix increment, myNumber is " +
         myNumber);
      System.out.println("  and answer is " + answer);
      myNumber = 17;
      System.out.println("Before incrementing, myNumber is " +
         myNumber);
      answer = myNumber++;
      System.out.println("After postfix increment, myNumber is " +
         myNumber);
      System.out.println("  and answer is " + answer);
   }
}
```

**Figure 6-14** The `IncrementDemo` application



```
C:\Java>java IncrementDemo
Before incrementing, myNumber is 17
After prefix increment, myNumber is 18
  and answer is 18
Before incrementing, myNumber is 17
After postfix increment, myNumber is 18
  and answer is 17

C:\Java>
```

**Figure 6-15** Output of the `IncrementDemo` application

Later in this chapter, you will learn another reason why you might want to use a prefix operator instead of a postfix operator.

Choosing whether to use a prefix or postfix operator is important when one is part of a larger expression. For example, if `d` is 5, then `2 * ++d` is 12, but `2 * d++` is 10.

Similar logic can be applied when you use the **prefix and postfix decrement operators**. For example, if `b = 4` and `c = b--`, 4 is assigned to `c`, but `b` is decreased and takes the value 3. If `b = 4` and `c = --b`, `b` is decreased to 3 and 3 is assigned to `c`.

Watch the video *Using Shortcut Arithmetic Operators*.

**TWO TRUTHS & A LIE**

**Using Shortcut Arithmetic Operators**

1. Assume that `x = 4` and `y = 5`. The value of `++y + ++x` is 11.
2. Assume that `x = 4` and `y = 5`. The value of `y == x++` is `true`.
3. Assume that `x = 4` and `y = 5`. The value of `y += x` is 9.

The false statement is #2. If x is 4 and y is 5, then the value of x++ is 4, and so y is not equal to 4.

## You Do It

*Working with Prefix and Postfix Increment Operators*

Next, you write an application that demonstrates how prefix and postfix operators are used to increment variables and how incrementing affects the expressions that contain these operators.

1. Open a new text file, and begin a demonstration class named `DemoIncrement` by typing:

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

2. On a new line, add a variable `v`, and assign it a value of 4. Then declare a variable named `plusPlusV`, and assign it a value of `++v` by typing:

```
int v = 4;
int plusPlusV = ++v;
```

3. The last statement, `int plusPlusV = ++v;`, increases `v` to 5, so before declaring a `vPlusPlus` variable to which you assign `v++`, reset `v` to 4 by typing:

```
v = 4;
int vPlusPlus = v++;
```

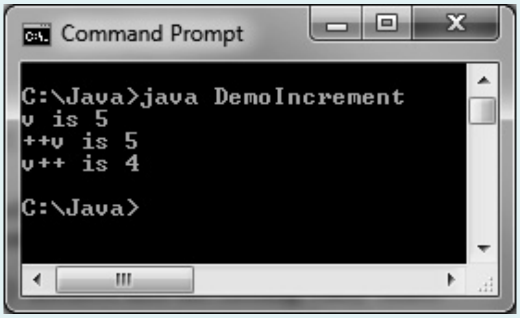
*(continues)*

*(continued)*

4. Add the following statements to display the three values:

```
System.out.println("v is " + v);
System.out.println("++v is " + plusPlusV);
System.out.println("v++ is " + vPlusPlus);
```

5. Add the closing curly brace for the `main()` method and the closing curly brace for the `DemoIncrement` class. Save the file as **DemoIncrement.java**, then compile and execute the program. Your output should look like Figure 6-16.



```
C:\Java>java DemoIncrement
v is 5
++v is 5
v++ is 4

C:\Java>
```

**Figure 6-16** Output of the `DemoIncrement` class

6. To illustrate how comparisons are made, add a few more variables to the `DemoIncrement` program. Change the class name to **`DemoIncrement2`**, and immediately save the file as **DemoIncrement2.java**.

7. After the last `println()` statement, add three new integer variables and two new Boolean variables. The first Boolean variable compares `++w` to `y`; the second Boolean variable compares `x++` to `y`:

```
int w = 17, x = 17, y = 18;
boolean compare1 = (++w == y);
boolean compare2 = (x++ == y);
```

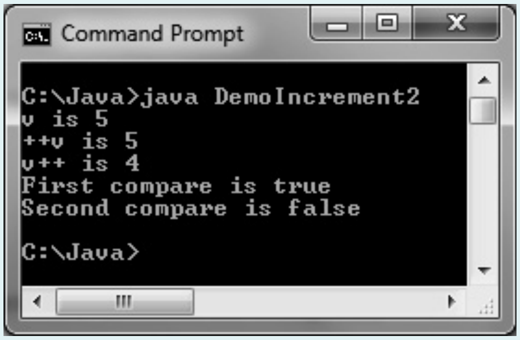
8. Add the following statements to display the values stored in the `compare` variables:

```
System.out.println("First compare is " + compare1);
System.out.println("Second compare is " + compare2);
```

9. Save, compile, and run the program. The output appears in Figure 6-17. Make certain you understand why each statement displays the values it does. Experiment by changing the values of the variables, and see if you can predict the output before recompiling and rerunning the program.

*(continues)*

*(continued)*



```
C:\Java>java DemoIncrement2
v is 5
++v is 5
v++ is 4
First compare is true
Second compare is false

C:\Java>
```

**Figure 6-17** Output of the `DemoIncrement2` application

## Creating a `for` Loop

A **`for` loop** is a special loop that is used when a definite number of loop iterations is required; it provides a convenient way to create a counter-controlled loop. Although a `while` loop can also be used to meet this requirement, the `for` loop provides you with a shorthand notation for this type of loop. When you use a `for` loop, you can indicate the starting value for the loop control variable, the test condition that controls loop entry, and the expression that alters the loop control variable—all in one convenient place.

You begin a `for` loop with the keyword `for` followed by a set of parentheses. Within the parentheses are three sections separated by exactly two semicolons. The three sections are usually used for the following:

- Initializing the loop control variable
- Testing the loop control variable
- Updating the loop control variable

The body of the `for` statement follows the parentheses. As with an `if` statement or a `while` loop, you can use a single statement as the body of a `for` loop, or you can use a block of statements enclosed in curly braces. Many programmers recommend that you always use a set of curly braces to surround the body of a `for` loop for clarity, even when the body contains only a single statement. You should use the conventions recommended by your organization.

Assuming that a variable named `val` has been declared as an integer, then the `for` statement shown in Figure 6-18 produces the same output as the `while` statement shown below it—both display the integers 1 through 10.

```
for(val = 1; val < 11; ++val)
{
   System.out.println(val);
}

val = 1;
while(val < 11)
{
   System.out.println(val);
   ++val;
}
```

**Figure 6-18** A `for` loop and a `while` loop that display the integers 1 through 10

Within the parentheses of the `for` statement shown in Figure 6-18, the first section prior to the first semicolon initializes `val` to 1. The program executes this statement once, no matter how many times the body of the `for` loop executes.

After initialization, program control passes to the middle, or test section, of the `for` statement that lies between the two semicolons. If the Boolean expression found there evaluates to `true`, the body of the `for` loop is entered. In the program segment shown in Figure 6-18, `val` is set to 1, so when `val < 11` is tested, it evaluates to `true`. The loop body displays `val`. In this example, the loop body is a single statement, so no curly braces are needed (although they could be added).

After the loop body executes, the final one-third of the `for` loop that follows the second semicolon executes, and `val` is increased to 2. Following the third section in the `for` statement, program control returns to the second section, where `val` is compared to 11 a second time. Because `val` is still less than 11, the body executes: `val` (now 2) is displayed, and then the third, altering portion of the `for` loop executes again. The variable `val` increases to 3, and the `for` loop continues.

Eventually, when `val` is not less than 11 (after 1 through 10 have been displayed), the `for` loop ends, and the program continues with any statements that follow the `for` loop. Although the three sections of the `for` loop are most commonly used for initializing, testing, and incrementing, you can also perform the following tasks:

- Initialization of more than one variable in the first section of the `for` statement by placing commas between the separate statements, as in the following:

  ```
  for(g = 0, h = 1; g < 6; ++g)
  ```

- Performance of more than one test using AND or OR operators in the second section, as in the following:

  ```
  for(g = 0; g < 3 && h > 1; ++g)
  ```

- Decrementation or performance of some other task in the third section, as in the following:

  ```
  for(g = 5; g >= 1; --g)
  ```

- Altering more than one variable in the third section, as in the following:

  ```
  for(g = 0; g < 10; ++g, ++h, sum += g)
  ```

- You can leave one or more portions of a `for` loop empty, although the two semicolons are still required as placeholders. For example, if `x` has been initialized in a previous program statement, you might write the following:

  ```
  for(; x < 10; ++x)
  ```

  However, to someone reading your program, leaving a section of a `for` statement empty is less clear than using all three sections.

In general, you should use the same loop control variable in all three parts of a `for` statement, although you might see some programs written by others in which this is not the case. You should also avoid altering the loop control variable in the body of the loop. If a variable is altered both within a `for` statement and within the block it controls, it can be very difficult to follow the program's logic. This technique can also produce program bugs that are hard to find. Usually, you should use the `for` loop for its intended purpose—as a shorthand way of programming a definite loop.

Occasionally, you will encounter a `for` loop that contains no body, such as the following:

```
for(x = 0; x < 100000; ++x);
```

This kind of loop exists simply to use time—that is, to occupy the central processing unit for thousands of processing cycles—when a brief pause is desired during program execution, for example. As with `if` and `while` statements, usually you do not want to place a semicolon at the end of the `for` statement before the body of the loop. Java also contains a built-in method to pause program execution. The `sleep()` method is part of the `Thread` class in the java.lang package, and you will learn how to use it as you continue to study Java.

You also can declare a variable within a `for` statement, as in the following:

```
for(int val = 1; val < 11; ++val)
```

Programmers often use this technique when the variable is not needed in any other part of the program. If you declare a variable within a `for` statement, the variable can be used only in the block that depends on the `for` statement; when the block ends, the variable goes out of scope.

Java also supports an enhanced `for` loop. You will learn about this loop in the chapter *Introduction to Arrays*.

Watch the video *Using the `for` Loop*.

## TWO TRUTHS & A LIE

### Creating a `for` Loop

1. A `for` loop always must contain two semicolons within its parentheses.
2. The body of a `for` loop might never execute.
3. Within the parentheses of a `for` loop, the last section must alter the loop control variable.

The false statement is #3. Frequently, the third section of a `for` loop is used to alter the loop control variable, but it is not required.

## You Do It

*Working with Definite Loops*

Suppose you want to find all the numbers that divide evenly into 100. You want to write a definite loop—one that executes exactly 100 times. In this section, you write a `for` loop that sets a variable to 1 and increments it to 100. Each of the 100 times through the loop, if 100 is evenly divisible by the variable, the application displays the number.

1. Open a new text file. Begin the application named `DivideEvenly` by typing the following code. Use a named constant for the 100 value and a variable named `var` that will hold, in turn, every value from 1 through 100:

```
public class DivideEvenly
{
   public static void main(String[] args)
   {
      final int LIMIT = 100;
      int var;
```

2. Type a statement that explains the purpose of the program:

```
System.out.print(LIMIT + " is evenly divisible by ");
```

3. Write the `for` loop that varies `var` from 1 through 100. With each iteration of the loop, test whether 100 % `var` is 0. If you divide 100 by a number and there is no remainder, the number goes into 100 evenly.

*(continues)*

*(continued)*

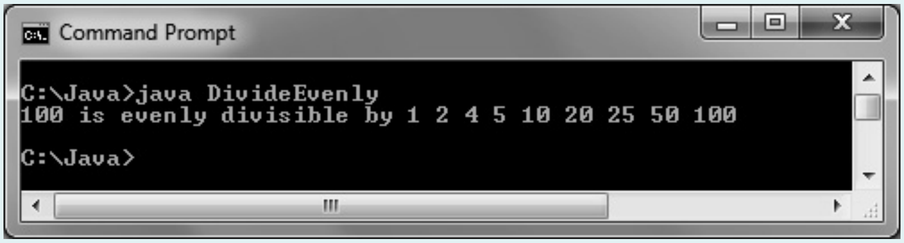
```
for(var = 1; var <= LIMIT; ++var)
    if(LIMIT % var == 0)
       System.out.print(var + " ");
       // Display the number and a space
```

4. Add an empty `println()` statement to advance the insertion point to the next line by typing the following:

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

5. Type the closing curly braces for the `main()` method and the `DivideEvenly` class.

6. Save the program as **DivideEvenly**. Compile and run the program. Figure 6-19 shows the output.



```
C:\Java>java DivideEvenly
100 is evenly divisible by 1 2 4 5 10 20 25 50 100

C:\Java>
```

**Figure 6-19** Output of the `DivideEvenly` application

7. By definition, no value that is greater than half of `LIMIT` can possibly go into `LIMIT` evenly. Therefore, the loop in the `DivideEvenly` program could be made to execute faster if the loop executes while `var` is less than or equal to half of `LIMIT`. If you decide to make this change to the program, remember that you must include the value of `LIMIT` in the output because it is evenly divisible into itself.

## Learning How and When to Use a do…while Loop

With all the loops you have written so far, the loop body might execute many times, but it is also possible that the loop will not execute at all. For example, recall the bank balance program that displays compound interest, which was shown in Figure 6-8. The program begins by asking whether the user wants to see next year's balance. If the user doesn't enter a *1* for *yes*, the loop body never executes.

Similarly, recall the `EnterSmallValue` application in Figure 6-10. The user is prompted to enter a value, and if the user enters a value that is 3 or less, the error-reporting loop body never executes.

In each of these cases, the loop control variable is evaluated at the "top" of the loop before the body has a chance to execute. Both `while` loops and `for` loops are **pretest loops**—ones in which the loop control variable is tested before the loop body executes.

Sometimes, you might need to ensure that a loop body executes at least one time. If so, you want to write a loop that checks at the "bottom" of the loop after the first iteration. The **do...`while` loop** is such a loop; it is a **posttest loop**—one in which the loop control variable is tested after the loop body executes.

Figure 6-20 shows the general structure of a `do...while` loop. Notice that the loop body executes before the loop-controlling question is asked even one time. Figure 6-21 shows a `BankBalance2` application that contains a `do...while` loop. The loop starts with the shaded keyword `do`. The body of the loop follows and is contained within curly braces. The first year's balance is output before the user has any option of responding. At the bottom of the loop, the user is prompted, "Do you want to see the balance at the end of another year?" Now the user has the option of seeing more balances, but viewing the first display was unavoidable. The user's response is checked in the shaded evaluation at the bottom of the loop; if it is *1* for *yes*, the loop repeats. Figure 6-22 shows a typical execution.

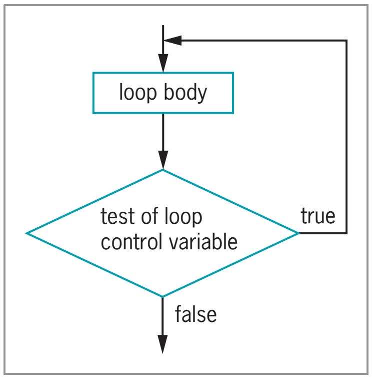
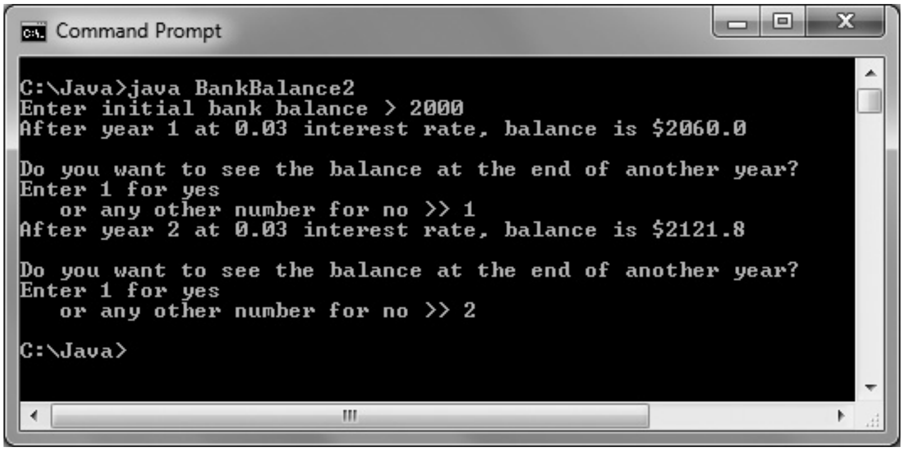


**Figure 6-20** General structure of a `do...while` loop

```
import java.util.Scanner;
public class BankBalance2
{
   public static void main(String[] args)
   {
      double balance;
      int response;
      int year = 1;
      final double INT_RATE = 0.03;
      Scanner keyboard = new Scanner(System.in);
      System.out.print("Enter initial bank balance > ");
      balance = keyboard.nextDouble();
      keyboard.nextLine();
      do
      {
         balance = balance + balance * INT_RATE;
         System.out.println("After year " + year + " at " + INT_RATE +
            " interest rate, balance is $" + balance);
         year = year + 1;
         System.out.println("\nDo you want to see the balance " +
            " at the end of another year?");
         System.out.println("Enter 1 for yes");
         System.out.print("   or any other number for no >> ");
         response = keyboard.nextInt();
      } while(response == 1);
   }
}
```

**Figure 6-21** A do...while loop for the BankBalance2 application



```
C:\Java>java BankBalance2
Enter initial bank balance > 2000
After year 1 at 0.03 interest rate, balance is $2060.0

Do you want to see the balance at the end of another year?
Enter 1 for yes
   or any other number for no >> 1
After year 2 at 0.03 interest rate, balance is $2121.8

Do you want to see the balance at the end of another year?
Enter 1 for yes
   or any other number for no >> 2

C:\Java>
```

**Figure 6-22** Typical execution of the BankBalance2 program

When the body of a do…while loop contains a single statement, you do not need to use curly braces to block the statement. For example, the following loop correctly adds numberValue to total while total remains less than 200:

```
do
   total += numberValue;
while(total < 200);
```

Even though curly braces are not required in this case, many programmers recommend using them. Doing so prevents the third line of code from looking like it should begin a new while loop instead of ending the previous do…while loop. Therefore, even though the result is the same, the following example that includes curly braces is less likely to be misunderstood by a reader:

```
do
{
   total += numberValue;
} while(total < 200);
```

You are never required to use a do…while loop. Within the bank balance example, you could achieve the same results as the logic shown in Figure 6-21 by unconditionally displaying the first year's bank balance once before starting the loop, prompting the user, and then starting a while loop that might not be entered. However, when you know you want to perform some task at least one time, the do…while loop is convenient.

**TWO TRUTHS & A LIE**

**Learning How and When to Use a do…while Loop**

1. The do…while loop checks the value of the loop control variable at the top of the loop prior to loop execution.
2. When the statements in a loop body must execute at least one time, it is convenient to use a do…while loop.
3. When the body of a do…while loop contains a single statement, you do not need to use curly braces to block the statement.

The false statement is #1. The do…while loop checks the value of the loop control variable at the bottom of the loop after one repetition has occurred.

## Learning About Nested Loops

Just as if statements can be nested, so can loops. You can place a while loop within a while loop, a for loop within a for loop, a while loop within a for loop, or any other combination. When loops are nested, each pair contains an **inner loop** and an **outer loop**. The inner loop must be entirely contained within the outer loop; loops can never overlap. Figure 6-23 shows a diagram in which the shaded loop is nested within another loop; the shaded area is the inner loop. You can nest virtually any number of loops; however, at some point, your machine will no longer be able to store all the necessary looping information.

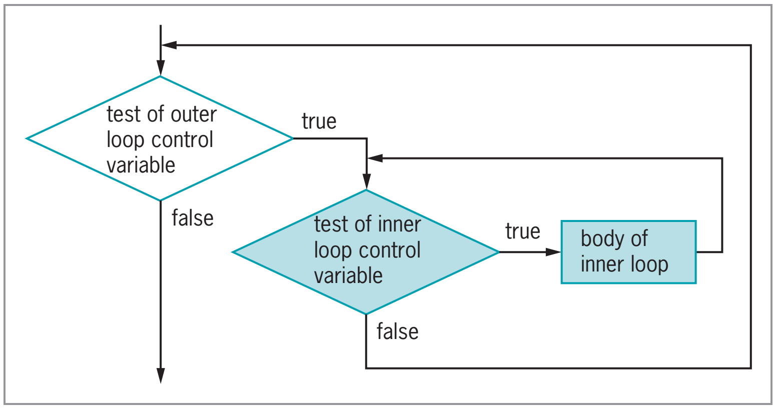

test of outer loop control variable
true
false
test of inner loop control variable
true
body of inner loop
false

Figure 6-23 Nested loops

Suppose you want to display future bank balances while varying both years and interest rates. Figure 6-24 shows an application that contains an outer loop that varies interest rates between specified limits. At the start of the outer loop, the working balance is reset to its initial value so that calculations are correct for each revised interest rate value. The shaded inner loop varies the number of years and displays each calculated balance. Figure 6-25 shows a typical execution.

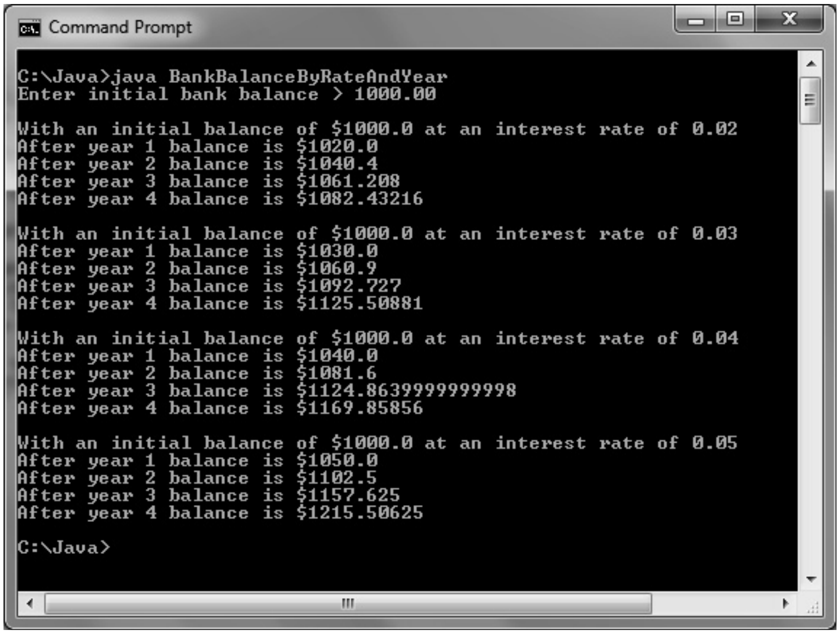
```
import java.util.Scanner;
public class BankBalanceByRateAndYear
{
   public static void main(String[] args)
   {
      double initialBalance;
      double balance;
      int year;
      double interest;
      final double LOW = 0.02;
      final double HIGH = 0.05;
      final double INCREMENT = 0.01;
      final int MAX_YEAR = 4;
      Scanner keyboard = new Scanner(System.in);
      System.out.print("Enter initial bank balance > ");
      initialBalance = keyboard.nextDouble();
      keyboard.nextLine();
```

Figure 6-24 The `BankBalanceByRateAndYear` class containing nested loops *(continues)*

*(continued)*

```
        for(interest = LOW; interest <= HIGH; interest += INCREMENT)
        {
           balance = initialBalance;
           System.out.println("\nWith an initial balance of $" +
              balance + " at an interest rate of " + interest);
           for(year = 1; year <= MAX_YEAR; ++ year)
           {
              balance = balance + balance * interest;
              System.out.println("After year " + year +
                 " balance is $" + balance);
           }
        }
     }
  }
```

**Figure 6-24** The `BankBalanceByRateAndYear` class containing nested loops



```
C:\Java>java BankBalanceByRateAndYear
Enter initial bank balance > 1000.00

With an initial balance of $1000.0 at an interest rate of 0.02
After year 1 balance is $1020.0
After year 2 balance is $1040.4
After year 3 balance is $1061.208
After year 4 balance is $1082.43216

With an initial balance of $1000.0 at an interest rate of 0.03
After year 1 balance is $1030.0
After year 2 balance is $1060.9
After year 3 balance is $1092.727
After year 4 balance is $1125.50881

With an initial balance of $1000.0 at an interest rate of 0.04
After year 1 balance is $1040.0
After year 2 balance is $1081.6
After year 3 balance is $1124.8639999999998
After year 4 balance is $1169.85856

With an initial balance of $1000.0 at an interest rate of 0.05
After year 1 balance is $1050.0
After year 2 balance is $1102.5
After year 3 balance is $1157.625
After year 4 balance is $1215.50625

C:\Java>
```

**Figure 6-25** Typical execution of the `BankBalanceByRateAndYear` program

In Figure 6-25, the floating-point calculations result in balances that contain fractions of pennies. If you wrote this program for a bank, you would have to ask whether interest should be compounded on fractions of a cent as it is here, or whether the amounts should be either rounded or truncated.

When you nest loops, sometimes it doesn't make any difference which variable controls the outer loop and which variable controls the inner one, but frequently it does make a difference. When you use a loop within a loop, you should always think of the outer loop as the all-encompassing loop. The variable in the outer loop changes more infrequently. For example, suppose a method named `outputLabel()` creates customer mailing labels in three different colors to use in successive promotional mailings. The following nested loop calls the method 60 times and produces three labels for the first customer, three labels for the second customer, and so on:

```
for(customer = 1; customer <= 20; ++customer)
   for(color = 1; color <= 3; ++color)
      outputLabel();
```

The following nested loop also calls `outputLabel()` 60 times, and it ultimately produces the same 60 labels, but it creates 20 labels in the first color, 20 labels in the second color, and then 20 labels in the third color.

```
for(color = 1; color <= 3; ++color)
   for(customer = 1; customer <= 20; ++customer)
      outputLabel();
```

If changing the ink color is a time-consuming process that occurs in the method, the second nested loop might execute much faster than the first one.

Watch the video *Nested Loops*.

## TWO TRUTHS & A LIE

### Learning About Nested Loops

1. You can place a `while` loop within a `while` loop or a `for` loop within a `for` loop, but you cannot mix loop types.
2. An inner nested loop must be entirely contained within its outer loop.
3. The body of the following loop executes 20 times:

```
for(int x = 0; x < 4; ++x)
   for(int y = 0; y < 5; ++y)
      System.out.println("Hi");
```

The false statement is #1. You can place a `while` loop within a `while` loop, a `for` loop within a `for` loop, a `while` loop within a `for` loop, or any other combination.

## You Do It

### *Working with Nested Loops*

Suppose you want to know not just what numbers go evenly into 100 but also what numbers go evenly into every positive number, up to and including 100. You can write 99 more loops—one that shows the numbers that divide evenly into 1, another that shows the numbers that divide evenly into 2, and so on—or you can place the current loop inside a different, outer loop, as you do next.

1. If necessary, open the file **DivideEvenly.java** in your text editor, change the class name to **`DivideEvenly2`**, and then save the class as **DivideEvenly2.java**.
2. Add a new variable declaration at the beginning of the file with the other variable declarations:

   ```
   int number;
   ```

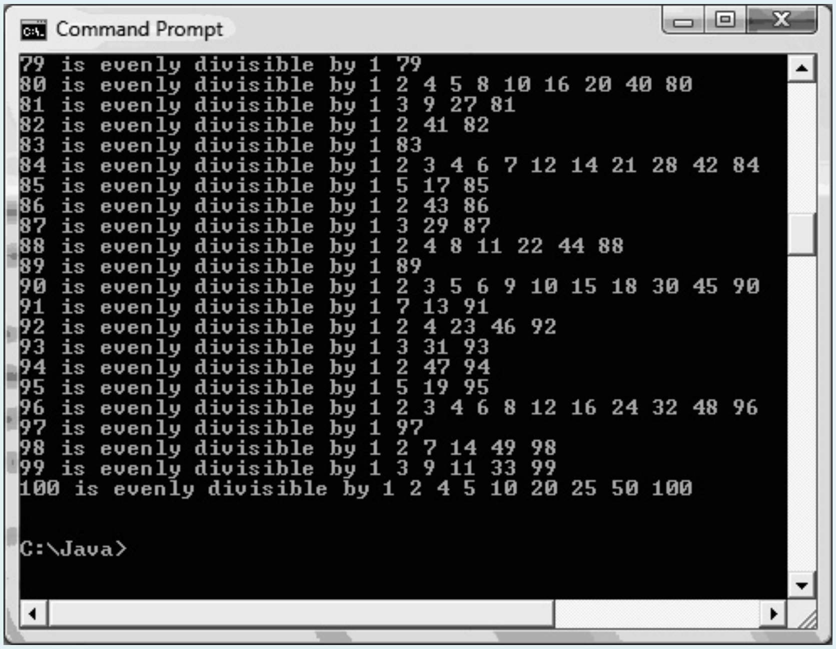
3. Replace the existing `for` loop with the following nested loop. The outer loop varies `number` from 1 to 100. For each number in the outer loop, the inner loop uses each positive integer from 1 up to the number and tests whether it divides evenly into the number:

   ```
   for(number = 1; number <= LIMIT; ++number)
   {
       System.out.print(number + " is evenly divisible by ");
       for(var = 1; var <= number; ++var)
          if(number % var == 0)
             System.out.print(var + " ");
             // Display the number and a space
       System.out.println();
   }
   ```

4. Make certain the file ends with three curly braces—one for the `for` outer loop that varies `number`, one for the `main()` method, and one for the class. The inner loop does not need curly braces because it contains a single output statement, although you could add a set of braces for the loop.
5. Save the file as **DivideEvenly2.java**, and then compile and execute the application. When the output stops scrolling, it should look similar to Figure 6-26.

*(continues)*

*(continued)*



```
79 is evenly divisible by 1 79
80 is evenly divisible by 1 2 4 5 8 10 16 20 40 80
81 is evenly divisible by 1 3 9 27 81
82 is evenly divisible by 1 2 41 82
83 is evenly divisible by 1 83
84 is evenly divisible by 1 2 3 4 6 7 12 14 21 28 42 84
85 is evenly divisible by 1 5 17 85
86 is evenly divisible by 1 2 43 86
87 is evenly divisible by 1 3 29 87
88 is evenly divisible by 1 2 4 8 11 22 44 88
89 is evenly divisible by 1 89
90 is evenly divisible by 1 2 3 5 6 9 10 15 18 30 45 90
91 is evenly divisible by 1 7 13 91
92 is evenly divisible by 1 2 4 23 46 92
93 is evenly divisible by 1 3 31 93
94 is evenly divisible by 1 2 47 94
95 is evenly divisible by 1 5 19 95
96 is evenly divisible by 1 2 3 4 6 8 12 16 24 32 48 96
97 is evenly divisible by 1 97
98 is evenly divisible by 1 2 7 14 49 98
99 is evenly divisible by 1 3 9 11 33 99
100 is evenly divisible by 1 2 4 5 10 20 25 50 100

C:\Java>
```

**Figure 6-26** Output of the `DivideEvenly2` application when scrolling stops

## Improving Loop Performance

Whether you decide to use a `while`, `for`, or `do…while` loop in an application, you can improve loop performance by doing the following:

- Making sure the loop does not include unnecessary operations or statements
- Considering the order of evaluation for short-circuit operators
- Making a comparison to 0
- Employing loop fusion
- Using prefix incrementing rather than postfix incrementing

### Avoiding Unnecessary Operations

You can make loops more efficient by not using unnecessary operations or statements, either within a loop's tested expression or within the loop body. For example, suppose a loop should execute while `x` is less than the sum of two integers, `a` and `b`. The loop could be written as:

```
while (x < a + b)
   // loop body
```

**Don't Do It**
It might be inefficient to recalculate a + b for every loop iteration.

If this loop executes 1,000 times, then the expression `a + b` is calculated 1,000 times. Instead, if you use the following code, the results are the same, but the arithmetic is performed only once:

```
int sum = a + b;
while(x < sum)
   // loop body
```

Of course, if `a` or `b` is altered in the loop body, then a new sum must be calculated with every loop iteration. However, if the sum of `a` and `b` is fixed prior to the start of the loop, then writing the code the second way is far more efficient.

Similarly, try to avoid a method call within a loop if possible. For example, if the method `getNumberOfEmployees()` always returns the same value during a program's execution, then a loop that begins as follows might unnecessarily call the method many times:

```
while(count < getNumberOfEmployees())…
```

It is more efficient to call the method once, store the result in a variable, and use the variable in the repeated evaluations, as in this example:

```
numEmployees = getNumberOfEmployees();

while(count < numEmployees)…
```

## Considering the Order of Evaluation of Short-Circuit Operators

In Chapter 5 you learned that the expressions in each part of an AND or an OR expression use short-circuit evaluation; that is, they are evaluated only as much as necessary to determine whether the entire expression is `true` or `false`. When a loop might execute many times, it becomes increasingly important to consider the number of evaluations that take place.

For example, suppose a user can request any number of printed copies of a report from 0 to 15, and you want to validate the user's input before proceeding. If you believe that users are far more likely to enter a value that is too high than to enter a negative one, then you want to start a loop that reprompts the user with the following expression:

```
while(requestedNum > LIMIT || requestedNum < 0)…
```

Because you believe that the first Boolean expression is more likely to be `true` than the second one, you can eliminate testing the second one on more occasions. The order of the expressions is not very important in a single loop, but if this loop is nested within another loop, then the difference in the number of tests increases. Similarly, the order of the evaluations in `if` statements is more important when the `if` statements are nested within a loop.

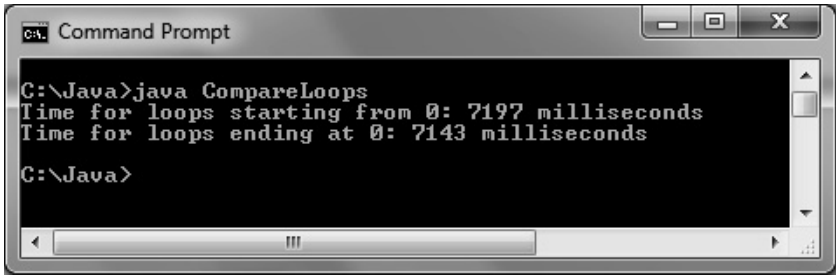
## Comparing to Zero

Making a comparison to 0 is faster than making a comparison to any other value. Therefore, if your application makes comparison to 0 feasible, you can improve loop performance by structuring your loops to compare the loop control variable to 0 instead of some other value. For example, a loop that performs based on a variable that varies from 0 up to 100,000 executes the same number of times as a loop based on a variable that varies from 100,000 down to 0. However, the second loop performs slightly faster. Comparing a value to 0 instead of other values is faster because in a compiled language, condition flags for the comparison are set once, no matter how many times the loop executes. Comparing a value to 0 is faster than comparing to other values, no matter which comparison operator you use—greater than, less than, equal to, and so on.

Figure 6-27 contains a program that tests the execution times of two nested do-nothing loops. (A **do-nothing loop** is one that performs no actions other than looping.) Before each loop, the `System` method `currentTimeMillis()` is called to retrieve the current time represented in milliseconds. After each nested loop repeats 100,000 times, the current time is retrieved again. Subtracting the two times computes the interval. As the execution in Figure 6-28 shows, there is a small difference in execution time between the two loops—about 1/100 of a second. The amount of time will vary on different machines, but the loop that uses the 0 comparison will never be slower than the other one. The difference would become more pronounced with additional repetitions or further nesting levels. If the loops used the loop control variable, for example, to display a count to the user, then they might need to increment the variable. However, if the purposes of the loops are just to count iterations, you might consider making the loop comparison use 0.

```java
public class CompareLoops
{
   public static void main(String[] args)
   {
      long startTime1, startTime2, endTime1, endTime2;
      final int REPEAT = 100000;
      startTime1 = System.currentTimeMillis();
      for(int x = 0; x <= REPEAT; ++x)
         for(int y = 0; y <= REPEAT; ++y);
      endTime1 = System.currentTimeMillis();
      System.out.println("Time for loops starting from 0: " +
         (endTime1 - startTime1) + " milliseconds");
      startTime2 = System.currentTimeMillis();
      for(int x = REPEAT; x >= 0; --x)
         for(int y = REPEAT; y >= 0; --y);
      endTime2 = System.currentTimeMillis();
      System.out.println("Time for loops ending at 0: " +
         (endTime2 - startTime2) + " milliseconds");
   }
}
```

**Figure 6-27** The `CompareLoops` application



```
C:\Java>java CompareLoops
Time for loops starting from 0: 7197 milliseconds
Time for loops ending at 0: 7143 milliseconds

C:\Java>
```

**Figure 6-28** Typical execution of the `CompareLoops` application

If you execute the program in Figure 6-27, you probably will see different results. With a fast operating system, you might not see the differences shown in Figure 6-28. If so, experiment with the program by increasing the value of `REPEAT` or by adding more nested levels to the loops.

## Employing Loop Fusion

**Loop fusion** is the technique of combining two loops into one. For example, suppose you want to call two methods 100 times each. You can set a constant named `TIMES` to 100 and use the following code:

```
for(int x = 0; x < TIMES; ++x)
   method1();
for(int x = 0; x < TIMES; ++x)
   method2();
```

However, you can also use the following code:

```
for(int x = 0; x < TIMES; ++x)
{
   method1();
   method2();
}
```

Fusing loops will not work in every situation; sometimes all the activities for `method1()` must be finished before those in `method2()` can begin. However, if the two methods do not depend on each other, fusing the loops can improve performance. On the other hand, if saving a few milliseconds ends up making your code harder to understand, you almost always should err in favor of slower but more readable programs.

## Using Prefix Incrementing Rather than Postfix Incrementing

Probably the most common action after the second semicolon in a `for` statement is to increment the loop control variable. In most textbooks and in many professional programs, the postfix increment operator is used for this purpose, as in the following:

```
for(int x = 0; x < LIMIT; x++)
```

Because incrementing x is a stand-alone statement in the for loop, the result is identical whether you use x++ or ++x. However, using the prefix increment operator produces a faster loop. Consider the program in Figure 6-29. It is similar to the CompareLoops application in Figure 6-27, but instead of comparing loops that count up and down, it compares loops that use prefix and postfix incrementing. The two timed, bodyless loops that repeat 1 billion times each are shaded in the figure.

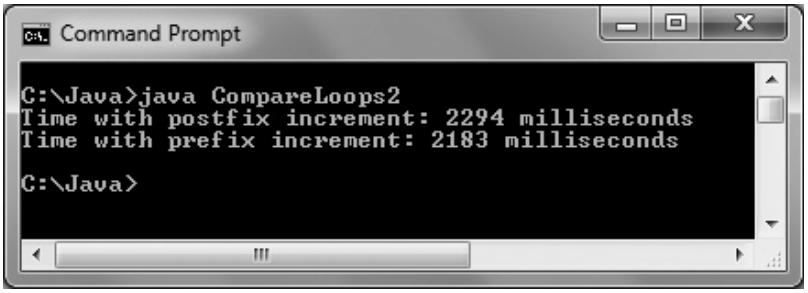
```
public class CompareLoops2
{
   public static void main(String[] args)
   {
      long startTime1, startTime2, endTime1, endTime2;
      final long REPEAT = 1000000000L;
      startTime1 = System.currentTimeMillis();
      for(int x = 0; x < REPEAT; x++);
      endTime1 = System.currentTimeMillis();
      System.out.println("Time with postfix increment: " +
         (endTime1 - startTime1)+ " milliseconds");
      startTime2 = System.currentTimeMillis();
      for(int x = 0; x < REPEAT; ++x);
      endTime2 = System.currentTimeMillis();
      System.out.println("Time with prefix increment: " +
         (endTime2 - startTime2)+ " milliseconds");
   }
}
```

**Figure 6-29** The CompareLoops2 program

Figure 6-30 shows the output of the program. The program that uses prefix incrementing runs slightly faster than the one that uses postfix incrementing because when the compiler uses postfix incrementing, it first makes a copy of the variable it uses as the expression's value, and then it increments the variable. In other words, the operators are methods that behave as follows:

- When you use the prefix incrementing method, as in ++x, the method receives a reference to x, increases it, and returns the increased value.
- When you use the postfix incrementing method, as in x++, the method receives a reference to x, makes a copy of the value and stores it, increments the value indicated by the reference, and returns the copy. The extra time required to make the copy is what causes postfix incrementing to take longer.

As you can see in Figure 6-30, the difference in duration for the loops is very small. If you run the program multiple times, you will get different results, and sometimes the prefix operator loop will take longer because other programs are running concurrently on your computer. However, using the prefix operator typically saves a small amount of time. As a professional, you will encounter programmers who insist on using either postfix or prefix increments in their loops. You should follow the conventions established by your organization, but now you have the tools to prove that prefix incrementing is faster.



```
C:\Java>java CompareLoops2
Time with postfix increment: 2294 milliseconds
Time with prefix increment: 2183 milliseconds

C:\Java>
```

**Figure 6-30** Typical execution of the `CompareLoops2` application

In the previous sections, you have learned to improve loop performance by eliminating unnecessary operations, considering the order of evaluation for short-circuit operators, making comparisons to 0, employing loop fusion, and using prefix incrementing rather than postfix incrementing. As you become an experienced programmer, you will discover other ways to enhance the operation of the programs you write. You should always be on the lookout for ways to improve program performance.

**TWO TRUTHS & A LIE**

**Improving Loop Performance**

1. You can improve the performance of a loop by making sure the loop does not include unnecessary operations in the tested expression.
2. You can improve loop performance by making sure the loop does not include unnecessary operations in the body of the loop.
3. You can improve loop performance when two conditions must both be true by testing for the most likely occurrence first.

The false statement is #3. You can improve loop performance when two conditions must both be true by testing for the least likely occurrence first. That way, the second test will need to be performed less frequently.

## You Do It

### *Comparing Execution Times for Separate and Fused Loops*

In this section, you compare the execution times for accomplishing the same tasks using two loops or a single one.

1. Open a new file in your text editor, and start a class named `TimeFusedLoop`.

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

2. Create variables to hold starting and ending times for two operations, and create a constant that holds a number of times to repeat loops:

```
long startTime1, startTime2, endTime1, endTime2;
final int REPEAT = 10000;
```

3. Get a starting time. Then, in a loop that repeats 10,000 times, call a method named `method1()`, which accepts a parameter that controls loop repetitions within the method. When the iterations are complete, call a similar method, `method2()`, 10,000 times.

```
startTime1 = System.currentTimeMillis();
for(int x = REPEAT; x >= 0; --x)
    method1(REPEAT);
for(int x = REPEAT; x >= 0; --x)
    method2(REPEAT);
```

4. Get the ending time, and display the difference between the starting and ending times.

```
endTime1 = System.currentTimeMillis();
System.out.println("Time for two separate loops: " +
  (endTime1 - startTime1)+ " milliseconds");
```

5. Get a new starting time, and call `method1()` and `method2()` 10,000 times each in a single loop. Then get the ending time, and display the difference. Add a closing curly brace for the `main()` method.

```
startTime2 = System.currentTimeMillis();
for(int x = REPEAT; x >= 0; --x)
{
    method1(REPEAT);
    method2(REPEAT);
}
endTime2 = System.currentTimeMillis();
System.out.println("Time for fused loops: " +
    (endTime2 - startTime2)+ " milliseconds");
}
```

*(continues)*

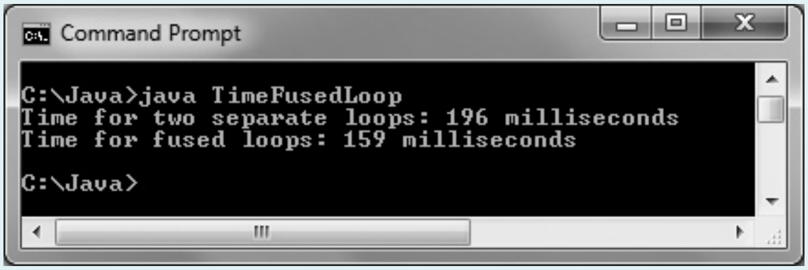
*(continued)*

6. Create the two methods named `method1()` and `method2()`. Each simply executes a do-nothing loop the number of times indicated by its parameter. Add a closing curly brace for the class.

```
    public static void method1(final int REPEAT)
    {
       for(int x = REPEAT; x >= 0; --x);
    }
    public static void method2(final int REPEAT)
    {
       for(int x = REPEAT; x >= 0; --x);
    }
}
```

7. Save the file as **TimeFusedLoop.java**, and then compile and execute it. Figure 6-31 shows the execution. The times might differ on your system, but you should be able to see that using a single loop significantly improves performance over using separate loops. If your system is very fast, you might have to increase the value of the `REPEAT` constant (perhaps changing it from an `int` to a `long`), or change the body of each method to contain a nested loop such as the following:

```
for(int x = REPEAT; x >= 0; --x);
   for(int y = REPEAT; y >= 0; --y);
```



```
C:\Java>java TimeFusedLoop
Time for two separate loops: 196 milliseconds
Time for fused loops: 159 milliseconds

C:\Java>
```

**Figure 6-31** Execution of the `TimeFusedLoop` program

8. Experiment with increasing and decreasing the value of `REPEAT`, and observe the effects. The time values you observe might also differ when you run the program at different times, depending on what other tasks are running on your system concurrently.

## Don’t Do It

- Don’t insert a semicolon at the end of a `while` clause before the loop body; doing so creates an empty loop body.
- Don’t forget to block multiple statements that should execute in a loop.
- Don’t make the mistake of checking for invalid data using a decision instead of a loop. Users might enter incorrect data multiple times, so a loop is the superior choice for input validation.
- Don’t ignore subtleties in the boundaries used to stop loop performance. For example, looping while interest rates are less than 8% is different than looping while interest rates are no more than 8%.
- Don’t repeat steps within a loop that could just as well be placed outside the loop; your program performance will improve.

## Key Terms

A **loop** is a structure that allows repeated execution of a block of statements.

A **loop body** is the block of statements that executes when the Boolean expression that controls the loop is `true`.

An **iteration** is one loop execution.

A **`while` loop** executes a body of statements continually as long as the Boolean expression that controls entry into the loop continues to be `true`.

A loop that executes a specific number of times is a **definite loop** or a counted loop.

An **indefinite loop** is one in which the final number of loops is unknown.

A **loop control variable** is a variable whose value determines whether loop execution continues.

An **infinite loop** is a loop that never ends.

An **empty body** is a block with no statements in it.

A **counter-controlled loop** is a definite loop.

**Incrementing** a variable adds 1 to its value.

**Decrementing** a variable reduces its value by 1.

An indefinite loop is an **event-controlled loop**.

**Validating data** is the process of ensuring that a value falls within a specified range.

A **priming read** or **priming input** is the first input statement prior to a loop that will execute subsequent input statements for the same variable.

**Counting** is the process of continually incrementing a variable to keep track of the number of occurrences of some event.

**Accumulating** is the process of repeatedly increasing a value by some amount to produce a total.

The **add and assign operator** ( += ) alters the value of the operand on the left by adding the operand on the right to it.

The **subtract and assign operator** ( −= ) alters the value of the operand on the left by subtracting the operand on the right from it.

The **multiply and assign operator** ( *= ) alters the value of the operand on the left by multiplying the operand on the right by it.

The **divide and assign operator** ( /= ) alters the value of the operand on the left by dividing the operand on the right into it.

The **remainder and assign operator** ( %= ) alters the value of the operand on the left by assigning the remainder when the left operand is divided by the right operand.

The **prefix ++**, also known as the **prefix increment operator**, adds 1 to a variable, then evaluates it.

The **postfix ++**, also known as the **postfix increment operator**, evaluates a variable, then adds 1 to it.

The **prefix and postfix decrement operators** subtract 1 from a variable.

A **for loop** is a special loop that can be used when a definite number of loop iterations is required.

A **pretest loop** is one in which the loop control variable is tested before the loop body executes.

The **do...while loop** executes a loop body at least one time; it checks the loop control variable at the bottom of the loop after one repetition has occurred.

A **posttest loop** is one in which the loop control variable is tested after the loop body executes.

An **inner loop** is contained entirely within another loop.

An **outer loop** contains another loop.

A **do-nothing loop** is one that performs no actions other than looping.

**Loop fusion** is the technique of combining two loops into one.

## Chapter Summary

- A loop is a structure that allows repeated execution of a block of statements. Within a looping structure, a Boolean expression is evaluated, and if it is `true`, a block of statements called the loop body executes; then the Boolean expression is evaluated again.
- You can use a `while` loop to execute a body of statements continually while some condition continues to be `true`. To execute a `while` loop, you initialize a loop control variable, test it in a `while` statement, and then alter the loop control variable in the body of the `while` structure.
- The add and assign operator ( `+=` ) adds and assigns in one operation. Similar versions are available for subtraction, multiplication, and division. The prefix and postfix increment operators increase a variable's value by 1. The prefix and postfix decrement operators reduce a variable's value by 1. An expression that uses a prefix operator uses the altered value, but an expression that uses a postfix operator uses the initial value before the increase.
- A `for` loop initializes, tests, and increments in one statement. There are three sections within the parentheses of a `for` loop that are separated by exactly two semicolons.
- The `do…while` loop tests a Boolean expression after one repetition has taken place, at the bottom of the loop.
- Loops can be nested, creating inner and outer loops.
- You can improve loop performance by making sure the loop does not include unnecessary operations or statements and by considering factors such as short-circuit evaluation, zero comparisons, loop fusion, and prefix incrementing.

## Review Questions

1. A structure that allows repeated execution of a block of statements is a ____________.

   a. cycle    c. ring

   b. loop    d. band

2. A loop that never ends is a(n) ____________ loop.

   a. iterative    c. structured

   b. infinite    d. illegal

3. To construct a loop that works correctly, you should initialize a loop control ____________.

   a. variable    c. structure

   b. constant    d. condition

4. What is the output of the following code?

```
b = 1;
while(b < 4)
   System.out.print(b + " ");
```

a. 1 c. 1 2 3 4
b. 1 2 3 d. 1 1 1 1 1 1...

5. What is the output of the following code?

```
b = 1;
while(b < 4)
{
   System.out.print(b + " ");
   b = b + 1;
}
```

a. 1 c. 1 2 3 4
b. 1 2 3 d. 1 1 1 1 1...

6. What is the output of the following code?

```
e = 1;
while(e < 4);
   System.out.print(e + " ");
```

a. Nothing c. 1 2 3 4
b. 1 1 1 1 1 1... d. 4 4 4 4 4 4...

7. If `total = 100` and `amt = 200`, then after the statement `total += amt`, ______________.

a. `total` is equal to 200 c. `amt` is equal to 100
b. `total` is equal to 300 d. `amt` is equal to 300

8. The prefix ++ is a ______________ operator.

a. unary c. tertiary
b. binary d. postfix

9. If `g = 5`, then after `h = ++g`, the value of `h` is ______________.

a. 4 c. 6
b. 5 d. 7

10. If `m = 9`, then after `n = m++`, the value of `m` is ______________.

a. 8 c. 10
b. 9 d. 11

11. If `m = 9`, then after `n = m++`, the value of `n` is ____________.

    a. 8 c. 10
    b. 9 d. 11

12. If `j = 5` and `k = 6`, then the value of `j++ == k` is ____________.

    a. 5 c. `true`
    b. 6 d. `false`

13. You must always include ____________ in a `for` loop's parentheses.

    a. two semicolons
    b. three semicolons
    c. two commas
    d. three commas

14. What does the following statement output?

```
for(a = 0; a < 5; ++a)
   System.out.print(a + " ");
```

    a. 0 0 0 0 0 c. 0 1 2 3 4 5
    b. 0 1 2 3 4 d. nothing

15. What does the following statement output?

```
for(b = 1; b > 3; ++b)
   System.out.print(b + " ");
```

    a. 1 1 1 c. 1 2 3 4
    b. 1 2 3 d. nothing

16. What does the following statement output?

```
for(f = 1, g = 4; f < g; ++f, --g)
   System.out.print(f + " " + g + " ");
```

    a. 1 4 2 5 3 6 4 7… c. 1 4 2 3
    b. 1 4 2 3 3 2 d. nothing

17. The loop that performs its conditional check at the bottom of the loop is a ____________ loop.

    a. `while` c. `for`
    b. `do…while` d. `for…while`

18. What does the following program segment output?

```
d = 0;
do
{
   System.out.print(d + " ");
   d++;
} while (d < 2);
```

a. 0
b. 0 1
c. 0 1 2
d. nothing

19. What does the following program segment output?

```
for(f = 0; f < 3; ++f)
   for(g = 0; g < 2; ++g)
      System.out.print(f + " " + g + " ");
```

a. 0 0 0 1 1 0 1 1 2 0 2 1
b. 0 1 0 2 0 3 1 1 1 2 1 3
c. 0 1 0 2 1 1 1 2
d. 0 0 0 1 0 2 1 0 1 1 1 2 2 0 2 1 2 2

20. What does the following program segment output?

```
for(m = 0; m < 4; ++m);
   for(n = 0; n < 2; ++n);
      System.out.print(m + " " + n + " ");
```

a. 0 0 0 1 1 0 1 1 2 0 2 1 3 0 3 1
b. 0 1 0 2 1 1 1 2 2 1 2 2
c. 4 2
d. 3 1

## Exercises

### *Programming Exercises*

1. a. Write an application that counts by five from 5 through 200 inclusive, and that starts a new line after every multiple of 50 (50, 100, 150, and 200). Save the file as **CountByFives.java**.

   b. Modify the `CountByFives` application so that the user enters the value to count by. Start each new line after 10 values have been displayed. Save the file as **CountByAnything.java**.

2. Write an application that asks a user to type an even number to continue or to type 999 to stop. When the user types an even number, display the message "Good job!" and then ask for another input. When the user types an odd number, display an error message and then ask for another input. When the user types 999, end the program. Save the file as **EvenEntryLoop.java**.

3. Write an application that displays the factorial for every integer value from 1 to 10. A factorial of a number is the product of that number multiplied by each positive integer lower than it. For example, 4 factorial is 4 * 3 * 2 * 1, or 24. Save the file as **Factorials.java**.

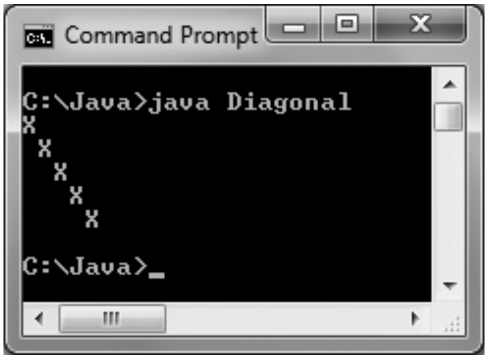
4. Write an application that prompts a user for two integers and displays every integer between them. Display a message if there are no integers between the entered values. Make sure the program works regardless of which entered value is larger. Save the file as **InBetween.java**.

5. A knot is a unit of speed equal to one nautical mile per hour. Write an application that displays every integer knot value from 15 through 30 and its kilometers per hour and miles per hour equivalents. One nautical mile is 1.852 kilometers or 1.151 miles. Save the file as **Knots.java**.

6. Write an application that shows the sum of 1 to *n* for every *n* from 1 to 50. That is, the program displays 1 (the sum of 1 alone), 3 (the sum of 1 and 2), 6 (the sum of 1, 2, and 3), 10 (the sum of 1, 2, 3, and 4), and so on. Save the file as **EverySum.java**.

7. Write an application that displays every perfect number from 1 through 1,000. A perfect number is one that equals the sum of all the numbers that divide evenly into it. For example, 6 is perfect because 1, 2, and 3 divide evenly into it, and their sum is 6; however, 12 is not a perfect number because 1, 2, 3, 4, and 6 divide evenly into it, and their sum is greater than 12. Save the file as **Perfect.java**.

8. Write an application that uses a loop to create the pattern of *X*s shown in Figure 6-32, in which each *X* is displayed one additional space to the right. Save the file as **Diagonal.java**.



```
C:\Java>java Diagonal
X
 X
  X
   X
    X

C:\Java>_
```

Figure 6-32 Output of the `Diagonal` application

9. Write an application that prompts a user for a child's current age and the estimated college costs for that child at age 18. Continue to reprompt a user who enters an age value greater than 18. Display the amount that the user needs to save each year until the child turns 18, assuming that no interest is earned on the money. For this application, you can assume that integer division provides a sufficient answer. Save the file as **CollegeCost.java**.

10. a. Write an application that prompts a user for the number of years the user has until retirement and the amount of money the user can save annually. If the user enters 0 or a negative number for either value, reprompt the user until valid entries are made. Assume that no interest is earned on the money. Display the amount of money the user will have at retirement. Save the file as **RetirementGoal.java**.

    b. Modify the `RetirementGoal` application to display the amount of money the user will have if the user earns 5% interest on the balance every year. Save the file as **RetirementGoal2.java**.

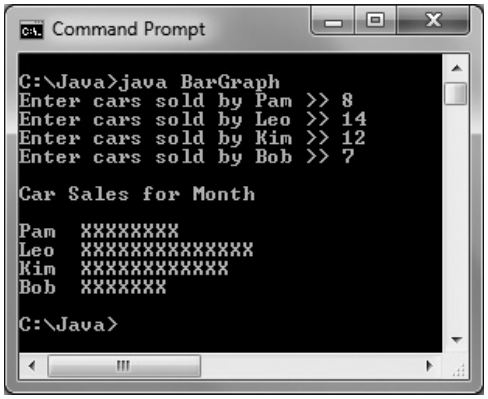
11. Pickering Manufacturing Company randomly selects one of its four factories to inspect each week. Write an application that determines which factory will be selected each week for the next 52 weeks. Use the `Math.random()` function explained in Appendix D to generate a factory number between 1 and 4; you use a statement similar to:

    ```
    factory = 1 + (int) (Math.random() * 4);
    ```

    After each selection, display the factory to inspect, and after the 52 selections are complete, display the percentage of inspections at each factory for the year. Run the application several times until you are confident that the factory selection is random. Save the file as **Inspections.java**.

12. Assume that the population of Mexico is 114 million and that the population increases 1.01 percent annually. Assume that the population of the United States is 312 million and that the population is reduced 0.15 percent annually. Write an application that displays the populations for the two countries every year until the population of Mexico exceeds that of the United States, and display the number of years it took. Save the file as **Population.java**.

13. Friendly Hal's Used Cars has four salespeople named Pam, Leo, Kim, and Bob. Accept values for the number of cars sold by each salesperson in a month, and create a bar chart that illustrates sales, similar to the one in Figure 6-33. Save the file as **BarGraph.java**.



```
C:\Java>java BarGraph
Enter cars sold by Pam >> 8
Enter cars sold by Leo >> 14
Enter cars sold by Kim >> 12
Enter cars sold by Bob >> 7

Car Sales for Month

Pam  XXXXXXXX
Leo  XXXXXXXXXXXXXX
Kim  XXXXXXXXXXXX
Bob  XXXXXXX

C:\Java>
```

**Figure 6-33** Typical execution of the `BarGraph` application

14. a. Create a class named `Purchase`. Each `Purchase` contains an invoice number, amount of sale, and amount of sales tax. Include set methods for the invoice number and sale amount. Within the `set()` method for the sale amount, calculate the sales tax as 5% of the sale amount. Also include a display method that displays a purchase's details. Save the file as **Purchase.java**.

    b. Create an application that declares a `Purchase` object and prompts the user for purchase details. When you prompt for an invoice number, do not let the user proceed until a number between 1,000 and 8,000 has been entered. When you prompt for a sale amount, do not proceed until the user has entered a nonnegative value. After a valid `Purchase` object has been created, display the object's invoice number, sale amount, and sales tax. Save the file as **CreatePurchase.java**.

15. Create a `BabysittingJob` class for Georgette's Babysitting Service. The class contains fields to hold the following:

    - A job number that contains six digits. The first two digits represent the year, and the last four digits represent a sequential number. For example, the first job in 2014 has a job number of 140001.
    - A code representing the employee assigned to the job. Assume that the code will always be 1, 2, or 3.
    - A name based on the babysitter code. Georgette has three babysitters: (1) Cindy, (2) Greg, and (3) Marcia.
    - The number of children to be watched. Assume that this number is always greater than zero.
    - The number of hours in the job. Assume that all hour values are whole numbers.
    - The fee for the job. Cindy is paid $7 per hour per child. Greg and Marcia are paid $9 an hour for the first child, and $4 per additional hour for each additional child. For example, if Greg watches three children for two hours, he makes $17 per hour for two hours, or $34.

    Create a constructor for the `BabysittingJob` class that accepts arguments for the job number, babysitter code, number of children, and hours. The constructor determines the babysitter name and fee for the job. Also include a method that displays every `BabysittingJob` object field. Save the file as **BabysittingJob.java**.

    Next, create an application that prompts the user for data for a babysitting job. Keep prompting the user for each of the following values until they are valid:

    - A four-digit year between 2013 and 2025 inclusive
    - A job number for the year between 1 and 9999 inclusive
    - A babysitter code of 1, 2, or 3

- A number of children for the job between 1 and 9 inclusive
- A number of hours between 1 and 12 inclusive

When all the data entries are valid, construct a job number from the last two digits of the year and a four-digit sequential number (which might require leading zeroes). Then, construct a `BabysittingJob` object, and display its values. Save the file as **CreateBabysittingJob.java**.

## *Debugging Exercises*

1. Each of the following files in the Chapter06 folder of your downloadable student files has syntax and/or logic errors. In each case, determine the problem and fix the program. After you correct the errors, save each file using the same filename preceded with *Fix*. For example, save DebugSix1.java as FixDebugSix1.java.

   a. DebugSix1.java
   b. DebugSix2.java
   c. DebugSix3.java
   d. DebugSix4.java

## *Game Zone*

1. a. Write an application that creates a quiz. The quiz should contain at least five questions about a hobby, popular music, astronomy, or any other personal interest. Each question should be a multiple-choice question with at least four answer options. When the user answers the question correctly, display a congratulatory message. If the user responds to a question incorrectly, display an appropriate message as well as the correct answer. At the end of the quiz, display the number of correct and incorrect answers and the percentage of correct answers. Save the file as **Quiz.java**.

   b. Modify the `Quiz` application so that the user is presented with each question continually until it is answered correctly. Remove the calculation for percentage of correct answers—all users will have 100 percent correct by the time they complete the application. Save the file as **Quiz2.java**.

2. In Chapter 1, you created a class called `RandomGuess`. In this game, players guess a number, the application generates a random number, and players determine whether they were correct. In Chapter 5, you improved the application to display a message indicating whether the player's guess was correct, too high, or too low. Now, add a loop that continuously prompts the user for the number, indicating whether the guess is high or low, until the user enters the correct value. After the user correctly guesses the number, display a count of the number of attempts it took. Save the file as **RandomGuess3.java**.

3. In Chapter 4, you created a `Die` class from which you could instantiate an object containing a random value from 1 through 6. Now use the class to create a simple dice game in which the user chooses a number between 2 (the lowest total possible from two dice) and 12 (the highest total possible). The user "rolls" two dice up to three times. If the number chosen by the user comes up, the user wins and the game ends. If the number does not come up within three rolls, the computer wins. Save the application as **TwoDice3.java**.

4. a. Using the `Die` class you created in Chapter 4, create a version of the dice game Pig that a user can play against the computer. The object of the game is to be the first to score 100 points. The user and computer take turns rolling a pair of dice following these rules:

   - On a turn, each player "rolls" two dice. If no *1* appears, the dice values are added to a running total, and the player can choose whether to roll again or pass the turn to the other player.
   - If a *1* appears on one of the dice, nothing more is added to the player's total, and it becomes the other player's turn.
   - If a *1* appears on both of the dice, not only is the player's turn over but the player's entire accumulated score is reset to 0.
   - In this version of the game, when the computer does not roll a *1* and can choose whether to roll again, generate a random value between 0 and 1. Have the computer continue to roll the dice when the value is 0.5 or more, and have the computer quit and pass the turn to the player when the value is not 0.5 or more.

   Save the game as **PigDiceGame.java**.

   b. Modify the `PigDiceGame` application so that if a player rolls a *1*, not only does the player's turn end but all the player's earned points during that round are eliminated. (Points from earlier rounds are not affected. That is, when either the player or computer rolls a 1, all the points accumulated since the other had a turn are subtracted.) Save the game as **PigDiceGame2.java**.

5. Two people play the game of Count 21 by taking turns entering a 1, 2, or 3, which is added to a running total. The player who adds the value that makes the total exceed 21 loses the game. Create a game of Count 21 in which a player competes against the computer, and program a strategy that always allows the computer to win. On any turn, if the player enters a value other than 1, 2, or 3, force the player to reenter the value. Save the game as **Count21.java**.

## Case Problems

1. Carly's Catering provides meals for parties and special events. In previous chapters, you developed a class that holds catering event information and an application that tests the methods using four objects of the class. Now modify the `EventDemo` class to do the following:

   - Continuously prompt for the number of guests for each `Event` until the value falls between 5 and 100 inclusive.
   - For one of the `Event` objects, create a loop that displays "Please come to my event!" as many times as there are guests for the `Event`.

   Save the modified file as **EventDemo.java**.

2. Sammy's Seashore Supplies rents beach equipment to tourists. In previous chapters, you developed a class that holds equipment rental information and an application that tests the methods using four objects of the class. Now modify the `RentalDemo` class to do the following:

   - Continuously prompt for the number of minutes of each `Rental` until the value falls between 60 and 7,200 inclusive.
   - For one of the `Rental` objects, create a loop that displays "Coupon good for 10 percent off next rental" as many times as there are full hours in the `Rental`.

   Save the modified file as **RentalDemo.java**.