CHAPTER 5

Looping

In this chapter you will:

- Learn how to create loops using the `while` statement
- Learn how to create loops using the `for` statement
- Learn how to create loops using the `do` statement
- Use nested loops
- Accumulate totals
- Understand how to improve loop performance
- Learn about looping issues in GUI programs

In the previous chapter, you learned how computers make decisions by evaluating Boolean expressions. Looping allows a program to repeat tasks based on the value of a Boolean expression. For example, programs that produce thousands of paychecks or invoices rely on the ability to loop to repeat instructions. Likewise, programs that repeatedly prompt users for a valid credit card number or for the correct answer to a tutorial question run more efficiently with loop structures. In this chapter, you will learn to create loops in C# programs. Computer programs seem smart due to their ability to make decisions; looping makes programs seem powerful.

Creating Loops with the `while` Statement

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. As long as the expression is `true`, the statements in the loop body continue to execute and the loop-controlling Boolean expression continues to be reevaluated. When the Boolean evaluation is `false`, the loop ends. Figure 5-1 shows a diagram of the logic of a loop. One execution of any loop is called an **iteration**.

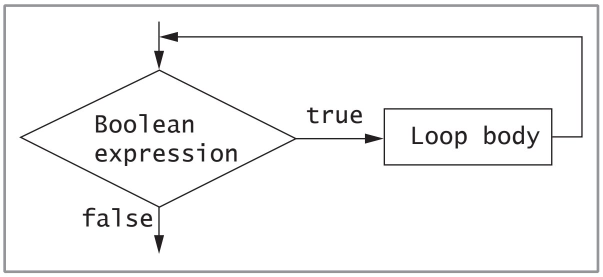


Figure 5-1 Flowchart of a loop structure

You can use a **`while` loop** to execute a body of statements continuously as long as the loop's test condition continues to be `true`. A `while` loop consists of the following:

- the keyword `while`
- a Boolean expression within parentheses
- the loop body

The evaluated Boolean expression in a `while` statement can be either a single Boolean expression or a compound expression that uses ANDs and ORs. The body can be a single statement or any number of statements surrounded by curly braces.

For example, the following code shows an integer declaration followed by a loop that causes the message *Hello* to display (theoretically) forever because there is no code to end the loop. A loop that never ends is called an **infinite loop**. An infinite loop might not actually execute infinitely. All programs run with the help of computer memory and hardware, both of which have finite capacities, so a program with an infinite loop might eventually fail and end. However, any loop that potentially runs forever is called infinite.

```
int number = 1;
while(number > 0)
   WriteLine("Hello");
```

In this loop, the expression `number > 0` evaluates as `true`, and *Hello* is displayed. The expression `number > 0` evaluates as `true` again, and *Hello* is displayed again. Because the value of `number` is never altered, the loop runs forever, evaluating the same Boolean expression and repeatedly displaying *Hello* as long as computer memory and hardware allow.

Writing an infinite loop is always a bad idea, although even experienced programmers write them by accident. If you ever find yourself in the midst of an infinite loop in a console application, you can break out by holding down the Ctrl key and pressing the C key or the Break (Pause) key. In a GUI program, you can simply close the `Frame` that is hosting the application.

To make a `while` loop end correctly, three separate actions should occur:

- A variable, the **loop control variable**, is initialized (before entering the loop).
- The loop control variable is tested in the `while` expression.
- The body of the loop must take some action that alters the value of the loop control variable (so that the `while` expression eventually evaluates as `false`).

For example, Figure 5-2 shows the logic for a loop that displays *Hello* four times. The variable `number` is initialized to 1, and a constant, `LIMIT`, is initialized to 5. The variable is less than `LIMIT`, and so the loop body executes. The loop body shown in Figure 5-2 contains two statements. The first displays *Hello*, and the second adds 1 to `number`. The next time `number` is evaluated, its value is 2, which is still less than `LIMIT`, so the loop body executes again. *Hello* displays a third time, and `number` becomes 4; then *Hello* displays a fourth time, and `number` becomes 5. Now when the expression `number < LIMIT` is evaluated, it is `false`, so the loop ends. If there were any subsequent statements following the `while` loop's closing curly brace, they would execute after the loop was finished.

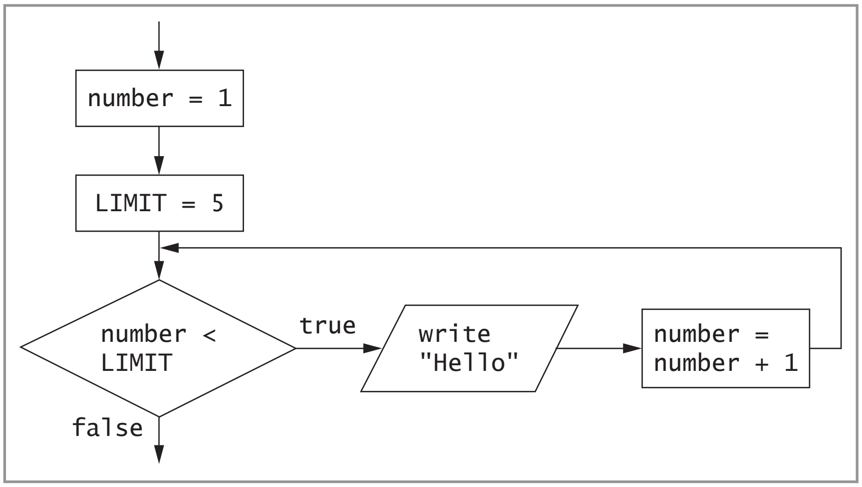
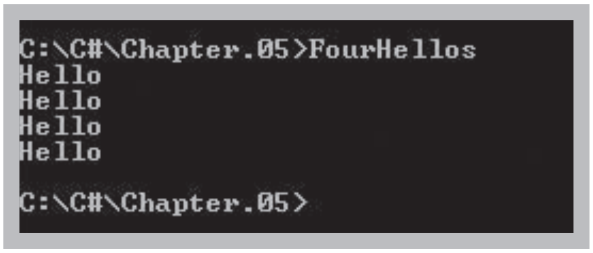


Figure 5-2 Flowchart for the logic of a `while` loop whose body executes four times

Figure 5-3 shows a C# program that uses the same logic as diagrammed in Figure 5-2. After the declarations, the shaded `while` expression compares `number` to `LIMIT`. The two statements that execute each time the Boolean expression is `true` are blocked using a pair of curly braces. Figure 5-4 shows the output.

```
using static System.Console;
class FourHellos
{
   static void Main()
   {
      int number = 1;
      const int LIMIT = 5;
      while(number < LIMIT)
      {
         WriteLine("Hello");
         number = number + 1;
      }
   }
}
```

Figure 5-3 A program that contains a `while` loop whose body executes four times



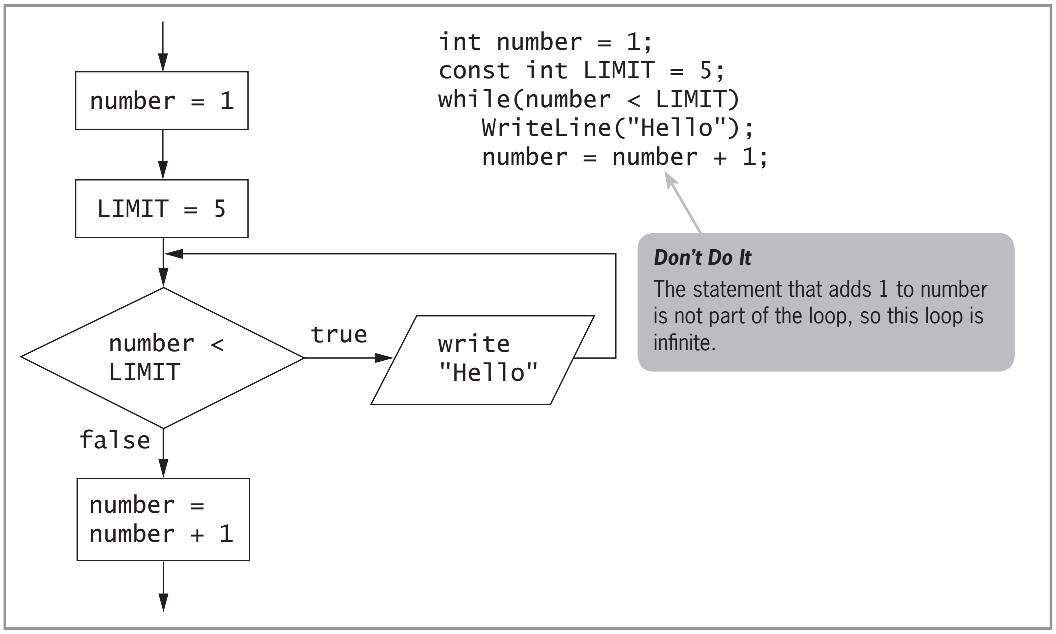
```
C:\C#\Chapter.05>FourHellos
Hello
Hello
Hello
Hello

C:\C#\Chapter.05>
```

Figure 5-4 Output of the `FourHellos` program

Recall from Chapter 2 that you also can use a shortcut operator to increase the value of a variable by 1. Instead of `number = number + 1`, you could achieve the same final result by writing `number++`, `++number`, or `number += 1`. Because counting is such a common feature of loops, you might want to review the difference between the prefix and postfix increment operators. A video called *Using Shortcut Arithmetic Operators* accompanies Chapter 2.

The curly braces surrounding the body of the `while` loop in Figure 5-3 are important. If they are omitted, the `while` loop ends at the end of the statement that displays *Hello*. Without the braces, adding 1 to `number` would no longer be part of the loop body, so an infinite loop would be created. Even if the statement that increases `number` was indented under the `while` statement, it would not be part of the loop without the surrounding curly braces. Figure 5-5 shows the incorrect logic that would result from omitting the curly braces. For clarity, many programmers recommend surrounding a loop body with curly braces even when there is only one statement in the body.



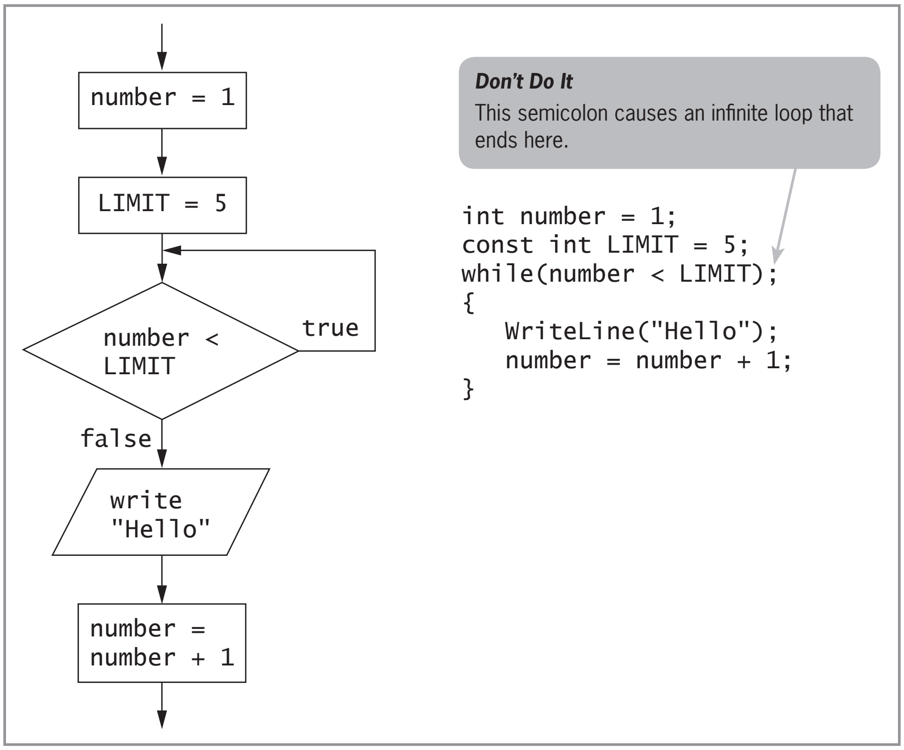
```
int number = 1;
const int LIMIT = 5;
while(number < LIMIT)
   WriteLine("Hello");
   number = number + 1;
```

number = 1
LIMIT = 5
number < LIMIT
true
write "Hello"
false
number = number + 1
Don't Do It
The statement that adds 1 to number is not part of the loop, so this loop is infinite.

Figure 5-5 Incorrect logic when curly braces are omitted from the loop in the `FourHellos` program

Because this code contains no curly braces, the `while` statement ends with the first semicolon, and an infinite loop is created.

Also, if a semicolon is mistakenly placed at the end of the partial statement, as in Figure 5-6, then the loop is also infinite. This loop has an **empty body**, or a body with no statements. In this case, `number` is initialized to 1, the Boolean expression `number < LIMIT` evaluates, and because it is `true`, the loop body is entered. Because the loop body is empty, ending at the semicolon, no action takes place, and the Boolean expression evaluates again. It is still `true` (nothing has changed), so the empty body is entered again, and the infinite loop continues. The program can never progress to either the statement that displays *Hello* or the statement that increases the value of `number`. The fact that these two statements are blocked using curly braces has no effect because of the incorrectly placed semicolon.



Don't Do It
This semicolon causes an infinite loop that ends here.

```
int number = 1;
const int LIMIT = 5;
while(number < LIMIT);
{
   WriteLine("Hello");
   number = number + 1;
}
```

Figure 5-6 Incorrect logic when an unwanted semicolon is mistakenly added to the loop in the `FourHellos` program

Because this code contains an unwanted semicolon, the loop has an empty body.

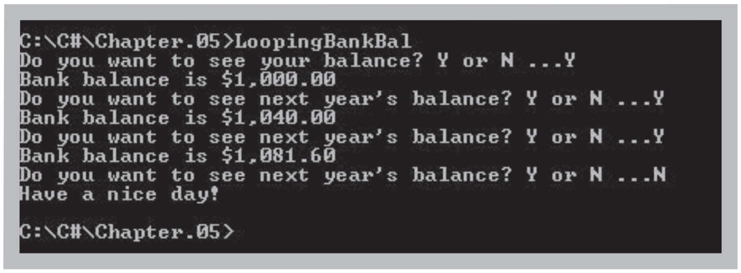
Within a correctly functioning loop's body, you can change the value of the loop control variable in a number of ways. Many loop control variable values are altered by **incrementing**, or adding to them, as in Figures 5-2 and 5-3. Other loops are controlled by reducing, or **decrementing**, a variable and testing whether the value remains greater than some benchmark value.

A loop for which the number of iterations is predetermined is called a **definite loop** or **counted loop**. Often, the value of a loop control variable is not altered by arithmetic but instead is altered by user input. For example, perhaps you want to continue performing some task while the user indicates a desire to continue. In that case, you do not know when you write the program whether the loop will be executed two times, 200 times, or not at all. This type of loop is an **indefinite loop**.

Consider a program that displays a bank balance and asks if the user wants to see what the balance will be after one year of interest has accumulated. Each time the user indicates she wants to continue, an increased balance appears. When the user finally indicates she has had enough, the program ends. The program appears in Figure 5-7, and a typical execution appears in Figure 5-8.

```
using System;
using static System.Console;
class LoopingBankBal
{
   static void Main()
   {
      double bankBal = 1000;
      const double INT_RATE = 0.04;
      string inputString;
      char response;
      Write("Do you want to see your balance? Y or N...");
      inputString = ReadLine();
      response = Convert.ToChar(inputString);
      while(response == 'Y')
      {
         WriteLine("Bank balance is {0}",
         bankBal.ToString("C"));
         bankBal = bankBal + bankBal * INT_RATE;
         Write("Do you want to see next year's balance? Y or N...");
         inputString = ReadLine();
         response = Convert.ToChar(inputString);
      }
      WriteLine("Have a nice day!");
   }
}
```

Figure 5-7 The LoopingBankBal program



```
C:\C#\Chapter.05>LoopingBankBal
Do you want to see your balance? Y or N ...Y
Bank balance is $1,000.00
Do you want to see next year's balance? Y or N ...Y
Bank balance is $1,040.00
Do you want to see next year's balance? Y or N ...Y
Bank balance is $1,081.60
Do you want to see next year's balance? Y or N ...N
Have a nice day!

C:\C#\Chapter.05>
```

Figure 5-8 Typical execution of the `LoopingBankBal` program

The program shown in Figure 5-7 continues to display bank balances while the response is *Y*. It could also be written to display while the response is not *N*, as in `while(response != 'N')...` . A value such as `'Y'` or `'N'` that a user must supply to stop a loop is a **sentinel value**.

In the program shown in Figure 5-7, the loop control variable is `response`. It is initialized by the first input and tested with the Boolean expression `response == 'Y'`. If the user types any character other than *Y*, then the loop body never executes; instead, the next statement to execute displays *Have a nice day!*. However, if the user enters *Y*, then all five statements within the loop body execute. The current balance is displayed, and the program increases the balance by the interest rate value; this value will not be displayed unless the user requests another loop repetition. Within the loop, the program prompts the user and reads in a new value for `response`. This input statement is the one that potentially alters the loop control variable. The loop ends with a closing curly brace, and program control returns to the top of the loop, where the Boolean expression is tested again. If the user typed *Y* at the last prompt, then the loop is reentered, and the increased `bankBal` value that was calculated during the last loop cycle is displayed.

In C#, character data is case sensitive. If a program tests `response == 'Y'`, a user response of *y* will result in a `false` evaluation. Beware of the pitfall of writing a loop similar to `while(response != 'Y' || response != 'y')...` to test both for uppercase and lowercase versions of the response. Every character is either not *Y* or not *y*, even *Y* and *y*. A correct loop might begin with the following:

```
while(response != 'Y' && response != 'y')…
```

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

TWO TRUTHS & A LIE

Creating Loops with the `while` Statement

1. To make a `while` loop that executes correctly, a loop control variable is initialized before entering the loop.
2. To make a `while` loop that executes correctly, the loop control variable is tested in the `while` expression.
3. To make a `while` loop that executes correctly, the body of the `while` statement must never alter the value of the loop control variable.

The false statement is #3. To make a `while` loop that executes correctly, the body of the `while` statement must take some action that alters the value of the loop control variable.

You Do It

Using a `while` Loop

In the next steps, you write a program that continuously prompts the user for a valid ID number until the user enters one. For this application, assume that a valid ID number must be between 1000 and 9999, inclusive.

1. Open a new project named **ValidID**, and enter the beginning of the program by declaring variables for an ID number, the user's input, and constant values for the highest and lowest acceptable ID numbers.

```
using System;
using static System.Console;
class ValidID
{
   static void Main()
   {
      int idNum;
      string input;
      const int LOW = 1000;
      const int HIGH = 9999;
```

(continues)

(continued)

2. Add code to prompt the user for an ID number, and to then convert it to an integer.

```
Write("Enter an ID number: ");
input = ReadLine();
idNum = Convert.ToInt32(input);
```

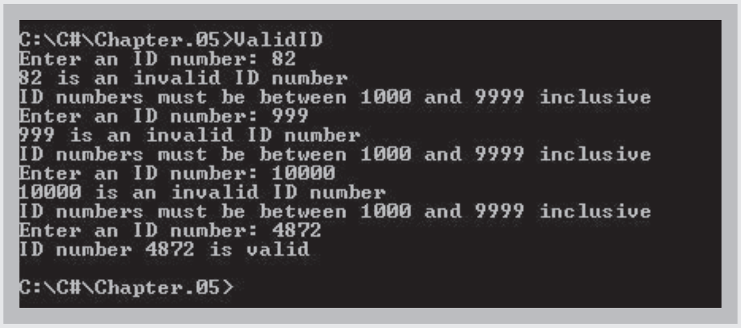
3. Create a loop that continues while the entered ID number is out of range. While the number is invalid, explain valid ID parameters and reprompt the user, converting the input to an integer.

```
while(idNum < LOW || idNum > HIGH)
{
   WriteLine("{0} is an invalid ID number", idNum);
   Write("ID numbers must be ");
   WriteLine("between {0} and {1} inclusive", LOW, HIGH);
   Write("Enter an ID number: ");
   input = ReadLine();
   idNum = Convert.ToInt32(input);
}
```

4. When the user eventually enters a valid ID number, the loop ends. Display a message, and add closing curly braces for the `Main()` method and for the class.

```
      WriteLine("ID number {0} is valid", idNum);
   }
}
```

5. Save the file, compile the program, and execute it. A typical execution during which the user makes several invalid entries is shown in Figure 5-9.



```
C:\C#\Chapter.05>ValidID
Enter an ID number: 82
82 is an invalid ID number
ID numbers must be between 1000 and 9999 inclusive
Enter an ID number: 999
999 is an invalid ID number
ID numbers must be between 1000 and 9999 inclusive
Enter an ID number: 10000
10000 is an invalid ID number
ID numbers must be between 1000 and 9999 inclusive
Enter an ID number: 4872
ID number 4872 is valid

C:\C#\Chapter.05>
```

Figure 5-9 Typical execution of the `ValidID` program

Creating Loops with the `for` Statement

The LoopingBankBal program in Figure 5-7 contains an indefinite loop because the number of executions is not predetermined; each time the program executes, the user might continue the loop a different number of times. You can use a `while` loop for either definite or indefinite loops. To write either type of `while` loop, you initialize a loop control variable, and as long as its test expression is `true`, you continue to execute the body of the `while` loop. To avoid an infinite loop, the body of the `while` loop must contain a statement that alters the loop control variable.

Because you need definite loops (ones with a predefined number of iterations) so frequently when you write programs, C# provides a shorthand way to create such loops. With 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` statement 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:

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

As with an `if` or a `while` statement, 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. The `while` and `for` statements shown in Figure 5-10 produce the same output—the integers 1 through 10.

```
// Declare loop control variable and limit
int x;
const int LIMIT = 10;

// Using a while loop to display 1 through 10
x = 1;
while(x <= LIMIT)
{
   WriteLine(x);
   ++x;
}

// Using a for loop to display 1 through 10
for(x = 1; x <= LIMIT; ++x)
   WriteLine(x);
```

Figure 5-10 Displaying integers 1 through 10 with `while` and `for` loops

The amount by which a loop control variable increases or decreases on each cycle through the loop is often called the **step value**. That's because in the BASIC programming language, and its descendent, the Visual Basic language, the keyword `STEP` is actually used in `for` loops.

Within the parentheses of the `for` statement shown in Figure 5-10, the initialization section prior to the first semicolon sets a variable named `x` to 1. The program executes this statement once, no matter how many times the body of the `for` loop eventually executes.

After the initialization expression executes, program control passes to the middle section, or test section, of the `for` statement. If the Boolean expression found there evaluates to `true`, then the body of the `for` loop is entered. In the program segment shown in Figure 5-10, `x` is initialized to 1, so when `x <= LIMIT` is tested, it evaluates to `true`, and the loop body outputs the value of `x`.

After the loop body executes, the final one-third of the `for` expression (the update section that follows the last semicolon) executes, and `x` increases to 2. Following the third section, program control returns to the second (test) section, where `x` is compared to `LIMIT` a second time. Because the value of `x` is 2, it is still less than or equal to `LIMIT`, so the body of the `for` loop executes, and the value of `x` is displayed. Then the third, altering portion of the `for` statement executes again. The variable `x` increases to 3, and the `for` loop continues.

Eventually, when `x` is *not* less than or equal to `LIMIT` (after 1 through 10 have been output), 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 a variable, testing it, and incrementing it, you can also perform other tasks:

- You can initialize more than one variable by placing commas between separate initializations in the first section of the `for` statement, as in the following:

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

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

```
for(int k = 0; k < 5; ++k)
```

In this example, `k` is declared to be an `int` and is initialized to 0. This technique is used frequently when the variable exists for no other purpose than to control the loop. When a variable is declared inside a loop, as `k` is in this example, it can be referenced only for the duration of the loop body; after that, it is **out of scope**, which means it is not usable because it has ceased to exist.

- You can perform more than one test in the middle section of the `for` statement by evaluating compound conditions, as in the following:

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

- You can perform tasks other than incrementing at the end of the loop's execution, as in the following code that decrements a variable:

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

- You can perform multiple tasks at the end of the loop's execution by separating the actions with commas, as in the following:

```
for(g = 0; g < 5; ++g, ++h)
```

- You can leave one or more portions of the `for` expression empty, although the two semicolons are still required as placeholders to separate the three sections. Usually, you want to use all three sections because the function of the loop is clearer to someone reading your program.

Generally, you should use the `for` loop in the standard manner, which is a shorthand way of initializing, testing, and altering a variable that controls a definite loop. You will learn about a similar loop, the `foreach` loop, when you study arrays in the next chapter.

Just as with a decision or a `while` loop, statements in a `for` loop body can be blocked. For example, the following loop displays *Hello* and *Goodbye* four times each:

```
const int TIMES = 4;
for(int x = 0; x < TIMES; ++x)
{
   WriteLine("Hello");
   WriteLine("Goodbye");
}
```

Without the curly braces in this code, the `for` loop would end after the statement that displays *Hello*. In other words, *Hello* would be displayed four times, but *Goodbye* would be displayed only once.

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

TWO TRUTHS & A LIE

Creating Loops with the `for` Statement

1. The following statement displays the numbers 3 through 6:

```
for(int x = 3; x <= 6; ++x)
   WriteLine(x);
```

2. The following statement displays the numbers 4 through 9:

```
for(int x = 3; x < 9; ++x)
   WriteLine(x + 1);
```

3. The following statement displays the numbers 5 through 12:

```
for(int x = 5; x < 12; ++x)
   WriteLine(x);
```

The false statement is #3. That loop displays only the numbers 5 through 11, because when x is 12, the loop body is not entered.

Creating Loops with the do Statement

With each of the loop statements you have learned about 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, part of which is shown in Figure 5-11. The loop begins at the shaded line by testing the value of `response`. If the user has not entered *Y*, the loop body never executes. The `while` loop checks a value at the "top" of the loop.

```
Write("Do you want to see your balance? Y or N...");
inputString = ReadLine();
response = Convert.ToChar(inputString);
while(response == 'Y')
{
   WriteLine("Bank balance is {0}", bankBal.ToString("C"));
   bankBal = bankBal + bankBal * INT_RATE;
   Write("Do you want to see next year's balance? Y or N...");
   inputString = ReadLine();
   response = Convert.ToChar(inputString);
}
```

Figure 5-11 Part of the bank balance program using a `while` loop

Sometimes you might need a loop body always to execute at least one time. If so, you can write a loop that checks at the "bottom" of the loop after the first iteration. The **do loop** (also called a **do...while loop**) checks the loop-controlling Boolean expression after one iteration. Figure 5-12 shows a diagram of the structure of a `do` loop.

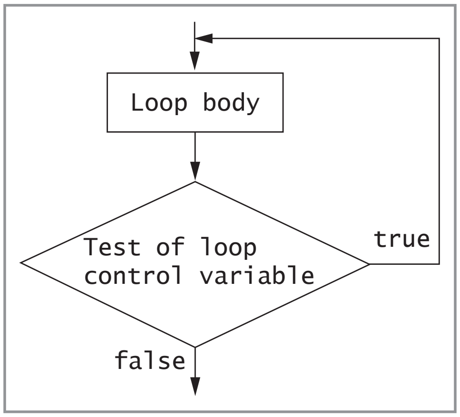
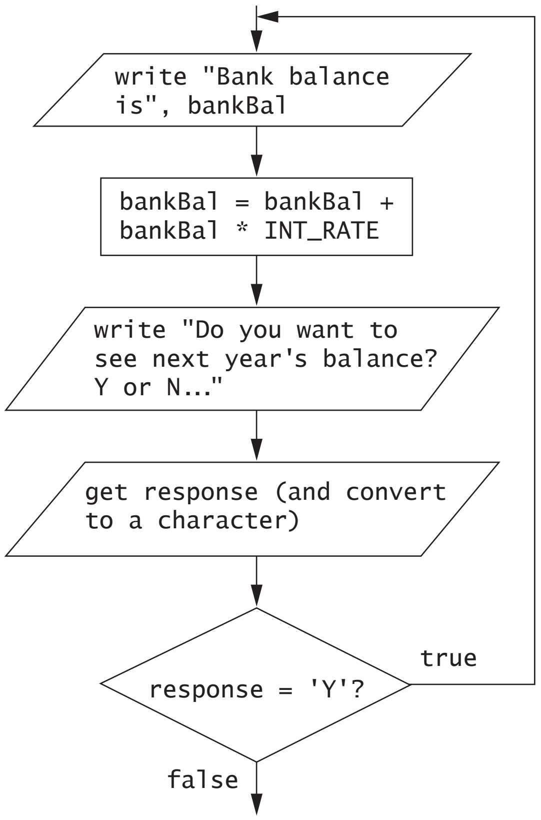


Figure 5-12 Flowchart of a `do` loop

Figure 5-13 shows the logic and the C# code for a `do` loop in a bank balance program. The loop starts with the keyword `do`. The body of the loop follows and is contained within curly braces. Within the loop, the next balance is calculated, and the user is prompted for a response. The Boolean expression that controls loop execution is written using a `while` statement, placed after the loop body. The `bankBal` variable is output the first time before the user has any option of responding. At the end of the loop, the user is prompted, *Do you want to see next year's balance? Y or N…*. Now the user has the option of seeing more balances, but the first view of the balance was unavoidable.


write "Bank balance is", bankBal
bankBal = bankBal + bankBal * INT_RATE
write "Do you want to see next year's balance? Y or N..."
get response (and convert to a character)
response = 'Y'?
true
false

```
do
{
   WriteLine("Bank balance is {0}", bankBal.ToString("C"));
   bankBal = bankBal + bankBal * INT_RATE;
   Write("Do you want to see next year's balance? Y or N ...");
   inputString = ReadLine();
   response = Convert.ToChar(inputString);
}  while(response == 'Y');
```

With the **do** loop, the loop control variable is tested after the loop body has executed one time.

Figure 5-13 Part of the bank balance program using a **do** loop

In a `do` loop, as a matter of style, many programmers prefer to align the `while` expression with the `do` keyword that starts the loop. Others feel that placing the `while` expression on its own line increases the chances that readers might misinterpret the line as the start of its own `while` statement instead of marking the end of a `do` statement.

You never are required to use a `do` loop. Within the bank balance example, you could achieve the same results by unconditionally displaying the bank balance once, prompting the user, and then starting a `while` loop that might not be entered. However, when a task must be performed at least one time, the `do` loop is convenient.

The `while` loop and `for` loop are **pretest loops**—ones in which the loop control variable is tested before the loop body executes. The `do` loop is a **posttest loop**—one in which the loop control variable is tested after the loop body executes.

TWO TRUTHS & A LIE

Creating Loops with the do Statement

1. The `do` loop checks the bottom of the loop after one iteration has occurred.
2. The Boolean expression that controls `do` loop execution is written using a `do` statement, placed after the loop body.
3. You never are required to use a `do` loop; you can always substitute one execution of the body statements followed by a `while` loop.

The false statement is #2. The Boolean expression that controls `do` loop execution is written using a `while` statement, placed after the loop body.

Using Nested Loops

Like `if` statements, loop statements also can be nested. 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 5-14 shows a diagram in which the shaded loop is nested within another loop; the shaded area is the inner loop as well as the body of the outer loop.

Suppose you want to display future bank balances for different years at a variety of interest rates. Figure 5-15 shows an application that contains an outer loop controlled by interest rates (starting with the first shaded statement in the figure) and an inner loop controlled by years (starting with the second shaded statement). The application displays annually compounded interest on $1000 at 4 percent, 6 percent, and 8 percent interest rates for one through five years. Figure 5-16 shows the output.

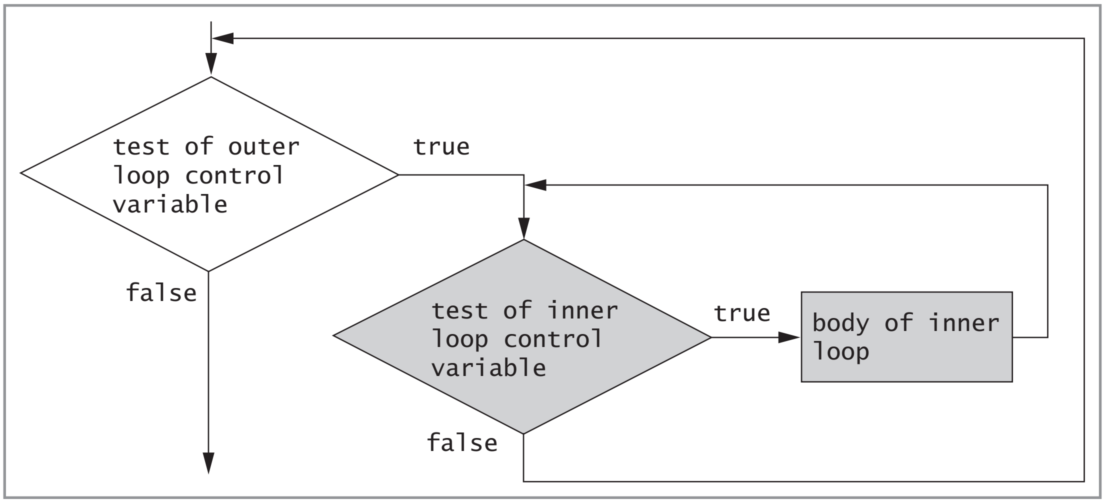
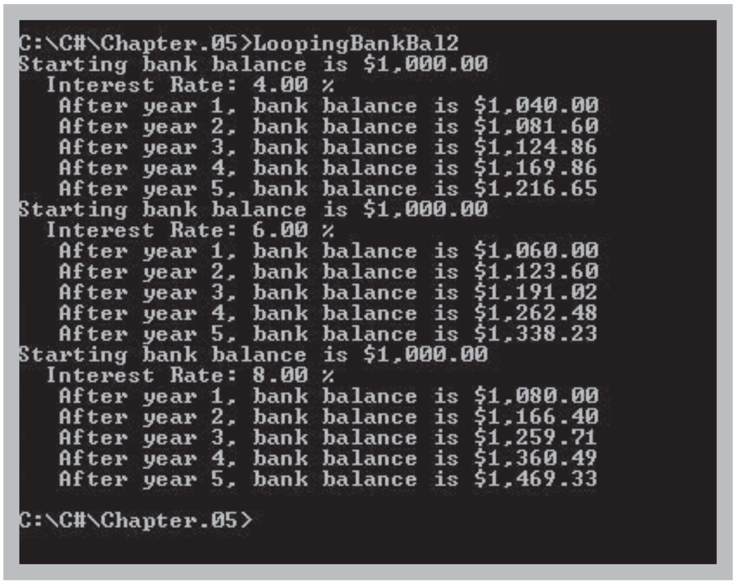

test of outer loop control variable
true
false
test of inner loop control variable
true
body of inner loop
false

Figure 5-14 Nested loops

```
using static System.Console;
class LoopingBankBal2
{
   static void Main()
   {
      double bankBal;
      double rate;
      int year;
      const double START_BAL = 1000;
      const double START_INT = 0.04;
      const double INT_INCREASE = 0.02;
      const double LAST_INT = 0.08;
      const int END_YEAR = 5;
      for(rate = START_INT; rate <= LAST_INT; rate += INT_INCREASE)
      {
         bankBal = START_BAL;
         WriteLine("Starting bank balance is {0}",
            bankBal.ToString("C"));
         WriteLine(" Interest Rate: {0}", rate.ToString("P"));
         for(year = 1; year <= END_YEAR; ++year)
         {
            bankBal = bankBal + bankBal * rate;
            WriteLine(" After year {0}, bank balance is {1}",
               year, bankBal.ToString("C"));
         }
      }
   }
}
```

Figure 5-15 The `LoopingBankBal2` program



```
C:\C#\Chapter.05>LoopingBankBal2
Starting bank balance is $1,000.00
  Interest Rate: 4.00 %
   After year 1, bank balance is $1,040.00
   After year 2, bank balance is $1,081.60
   After year 3, bank balance is $1,124.86
   After year 4, bank balance is $1,169.86
   After year 5, bank balance is $1,216.65
Starting bank balance is $1,000.00
  Interest Rate: 6.00 %
   After year 1, bank balance is $1,060.00
   After year 2, bank balance is $1,123.60
   After year 3, bank balance is $1,191.02
   After year 4, bank balance is $1,262.48
   After year 5, bank balance is $1,338.23
Starting bank balance is $1,000.00
  Interest Rate: 8.00 %
   After year 1, bank balance is $1,080.00
   After year 2, bank balance is $1,166.40
   After year 3, bank balance is $1,259.71
   After year 4, bank balance is $1,360.49
   After year 5, bank balance is $1,469.33

C:\C#\Chapter.05>
```

Figure 5-16 Output of the `LoopingBankBal2` program

When you use a loop within a loop, you should always think of the outer loop as the all-encompassing loop. When you describe the task at hand, you often use the word *each* to refer to the inner loop. For example, if you wanted to output balances for different interest rates each year for 10 years, you could appropriately initialize some constants as follows:

```
const double RATE1 = 0.03;
const double RATE2 = 0.07
const double RATE_INCREASE = 0.01
const int END_YEAR = 10;
```

Then you could use the following nested `for` loops:

```
for(rate = RATE1; rate <= RATE2; rate += RATE_INCREASE)
   for(year = 1; year <= END_YEAR; ++year)
      WriteLine(bankBal + bankBal * rate);
```

However, if you wanted to display balances for different years for each possible interest rate, you would use the following:

```
for(year = 1; year <= END_YEAR; ++year)
   for(rate = RATE1; rate <= RATE2; rate += RATE_INCREASE)
      WriteLine(bankBal + bankBal * rate);
```

In both of these examples, the same 50 values would be displayed—five different interest rates (from 0.03 through 0.07) for 10 years (1 through 10). However, in the first example, balances for years 1 through 10 would be displayed *within* each interest rate category, and in the second example, each balance for each interest rate would be displayed *within* each year category. In other words, in the first example, the first 10 amounts displayed would all use the first rate of 0.03 for 10 different years, and in the second example, the first five amounts displayed would use different interest values all in the first year.

Watch the video *Using Nested Loops*.

TWO TRUTHS & A LIE

Using Nested Loops

1. The body of the following loop executes six times:

```
for(a = 1; a < 4; ++a)
   for(b = 2; b < 3; ++b)
```

2. The body of the following loop executes four times:

```
for(c = 1; c < 3; ++c)
   for(d = 1; d < 3; ++d)
```

3. The body of the following loop executes 15 times:

```
for(e = 1; e <= 5; ++e)
   for(f = 2; f <= 4; ++f)
```

The false statement is #1. That loop executes only three times. The outer loop executes when a is 1, 2, and 3. The inner loop executes just once for each of those iterations.

You Do It

Using a Nested Loop

In the next steps, you write a program that creates a tipping table that restaurant patrons can use to approximate the correct tip for meals. Prices range from $10 to $100, and tipping percentage rates range from 10 percent to 25 percent. The program uses several loops.

1. Open a new file to start a program named **TippingTable**. It begins by declaring variables to use for the price of a dinner, a tip percentage rate, and the amount of the tip.

```
using static System.Console;
class TippingTable
{
   static void Main()
   {
      double dinnerPrice = 10.00;
      double tipRate;
      double tip;
```

(continues)

(continued)

2. Next, create some constants. Every tip from 10 percent through 25 percent will be computed in 5 percent intervals, so declare those values as `LOWRATE`, `MAXRATE`, and `TIPSTEP`. Tips will be calculated on dinner prices up to $100.00 in $10.00 intervals, so declare those constants, too.

```
const double LOWRATE = 0.10;
const double MAXRATE = 0.25;
const double TIPSTEP = 0.05;
const double MAXDINNER = 100.00;
const double DINNERSTEP = 10.00;
```

3. To create a heading for the table, display *Price*. (For alignment, insert three spaces after the quotes and before the *P* in *Price*.) On the same line, use a loop that displays every tip rate from `LOWRATE` through `MAXRATE` in increments of `TIPSTEP`. In other words, the tip rates are 0.10, 0.15, 0.20, and 0.25. Complete the heading for the table using a `WriteLine()` statement that advances the cursor to the next line of output and a `WriteLine()` statement that displays a dashed line.

```
Write("Price");
for(tipRate = LOWRATE; tipRate <= MAXRATE; tipRate += TIPSTEP)
   Write("{0, 8}", tipRate.ToString("F"));
WriteLine();
WriteLine("----------------------------------------");
```

Recall that within a `for` loop, the expression before the first semicolon executes once, the middle expression is tested, the loop body executes, and then the expression to the right of the second semicolon executes. In other words, `TIPSTEP` is not added to `tipRate` until after `tipRate` is displayed on each cycle through the loop.

As an alternative to typing 40 dashes in the `WriteLine()` statement, you could use the following loop to display a single dash 40 times. When the 40 dashes were completed, you could use `WriteLine()` to advance the cursor to a new line.

```
const int NUM_DASHES = 40;
for(int x = 0; x < NUM_DASHES; ++x)
   Write("-");
WriteLine();
```

4. Reset `tipRate` to 0.10. You must reset the rate because after the last loop, the rate will have been increased to greater than 0.25.

```
tipRate = LOWRATE;
```

5. Create a nested loop that continues while the `dinnerPrice` remains 100.00 (`MAXDINNER`) or less. Each iteration of this loop displays one row of the tip table. Within this loop, display the `dinnerPrice`, then loop to display four tips while the `tipRate` varies from 0.10 through 0.25. At the end of the

(continues)

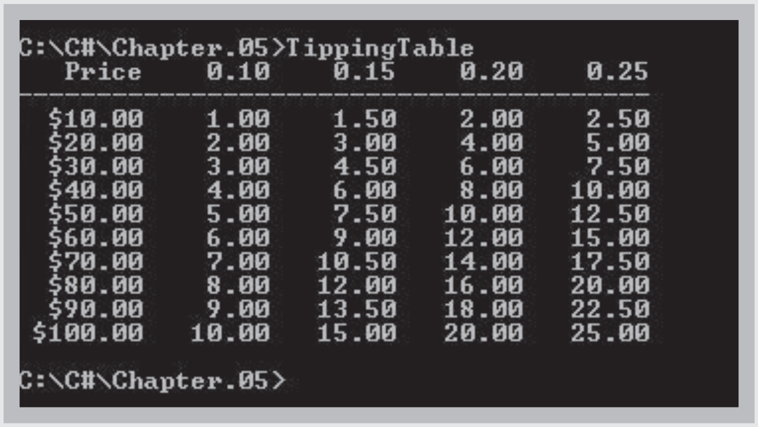
(continued)

loop, increase the `dinnerPrice` by 10.00, reset the `tipRate` to 0.10 so it is ready for the next row, and write a new line to advance the cursor.

```
while(dinnerPrice <= MAXDINNER)
{
   Write("{0, 8}", dinnerPrice.ToString("C"));
   while(tipRate <= MAXRATE)
   {
      tip = dinnerPrice * tipRate;
      Write("{0, 8}", tip.ToString("F"));
      tipRate += TIPSTEP;
   }
   dinnerPrice += DINNERSTEP;
   tipRate = LOWRATE;
   WriteLine();
}
```

Recall that the `{0, 8}` format string in the `Write()` statements displays the first argument in fields that are eight characters wide. You learned about format strings in Chapter 2.

6. Add two closing curly braces—one for the `Main()` method and one for the class.
7. Save the file, compile it, and execute the program. The output looks like Figure 5-17.



```
C:\C#\Chapter.05>TippingTable
   Price    0.10    0.15    0.20    0.25
----------------------------------------
  $10.00    1.00    1.50    2.00    2.50
  $20.00    2.00    3.00    4.00    5.00
  $30.00    3.00    4.50    6.00    7.50
  $40.00    4.00    6.00    8.00   10.00
  $50.00    5.00    7.50   10.00   12.50
  $60.00    6.00    9.00   12.00   15.00
  $70.00    7.00   10.50   14.00   17.50
  $80.00    8.00   12.00   16.00   20.00
  $90.00    9.00   13.50   18.00   22.50
 $100.00   10.00   15.00   20.00   25.00

C:\C#\Chapter.05>
```

Figure 5-17 Output of the `TippingTable` program

(continues)

(continued)

In the exercises at the end of this chapter, you will be instructed to make an interactive version of the TippingTable program in which many of the values are input by the user instead of being coded into the program as unnamed constants.

You Do It

Using a do Loop

In the next steps, you revise the TippingTable program to use a `do` loop in place of the `while` loop.

1. Open the **TippingTable** program, change the class name to **`TippingTable2`**, and save the file as **TippingTable2**.
2. Cut the `while` loop clause (`while(dinnerPrice <= MAXDINNER)`), and replace it with **`do`**.
3. Paste the `while` loop clause just after the closing curly brace for the loop and before the final two curly braces in the program, and then add a semicolon.
4. Save, compile, and execute the program. The output is identical to the output in Figure 5-17.

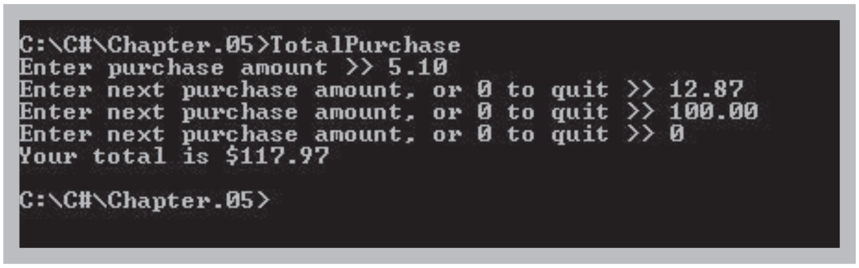
Accumulating Totals

Many computer programs display totals. When you receive a credit card or telephone service bill, you are usually provided with individual transaction details, but you are most interested in the total bill. Similarly, some programs total the number of credit hours generated by college students, the gross payroll for all employees of a company, or the total accounts receivable value for an organization. These totals are **accumulated**—that is, they are summed one at a time in a loop.

Figure 5-18 shows an example of an interactive program that accumulates the user's total purchases. The program prompts the user to enter a purchase price or *0* to quit. While the user continues to enter nonzero values, the amounts are added to a total. With each pass through the loop, the total is recalculated to include the new purchase amount. After the user enters the loop-terminating *0*, the accumulated total can be displayed. Figure 5-19 shows a typical program execution.

```
using System;
using static System.Console;
class TotalPurchase
{
   static void Main()
   {
      double purchase;
      double total = 0;
      string inputString;
      const double QUIT = 0;
      Write("Enter purchase amount >> ");
      inputString = ReadLine();
      purchase = Convert.ToDouble(inputString);
      while(purchase != QUIT)
      {
         total += purchase;
         Write("Enter next purchase amount, or " +
            QUIT + " to quit >> ");
         inputString = ReadLine();
         purchase = Convert.ToDouble(inputString);
      }
      WriteLine("Your total is {0}", total.ToString("C"));
   }
}
```

Figure 5-18 An application that accumulates total purchases entered by the user



```
C:\C#\Chapter.05>TotalPurchase
Enter purchase amount >> 5.10
Enter next purchase amount, or 0 to quit >> 12.87
Enter next purchase amount, or 0 to quit >> 100.00
Enter next purchase amount, or 0 to quit >> 0
Your total is $117.97

C:\C#\Chapter.05>
```

Figure 5-19 Typical execution of the `TotalPurchase` program

Recall from Chapter 2 that the expression `total += purchase` uses the shortcut add and assign operator, and that it is equivalent to `total = total + purchase`. The add and assign operator is frequently used when a running total is kept.

In the application in Figure 5-18, it is very important that the `total` variable used for accumulation is set to 0 before the loop begins. When it is not, the program will not compile. An unknown, unassigned value is known as **garbage**. The C# compiler prevents you from seeing an incorrect total by requiring you to provide a starting value; C# will not use the garbage value that happens to be stored at an uninitialized memory location.

In the application in Figure 5-18, the `total` variable must be initialized to 0, but the `purchase` variable is uninitialized. Many programmers would say it makes no sense to initialize this variable because no matter what starting value you provide, the value can be changed by the first input statement before the variable is ever used. As a matter of style, this book generally does not initialize variables if the initialization value is never used; doing so might mislead you into thinking the starting value had some purpose.

TWO TRUTHS & A LIE

Accumulating Totals

1. When totals are accumulated, 1 is added to a variable that represents the total.
2. A variable used to hold a total must be set to 0 before it is used to accumulate a total.
3. The C# compiler will not allow you to accumulate totals in an uninitialized variable.

The false statement is #1. When totals are accumulated, any value might be added to a variable that represents the total. For example, if you total 10 test scores, then each score is added. If you add only 1 to a total variable on each cycle through a loop, then you are counting rather than accumulating a total.

Improving Loop Performance

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

- Make sure the loop avoids unnecessary operations.
- Consider the order of evaluation for short-circuit operators.
- Employ loop fusion.
- Use prefix incrementing rather than postfix incrementing.

Avoiding Unnecessary Operations

You can improve loop performance by making sure the loop does not include unnecessary operations or statements. For example, suppose that a loop should execute while `x` is less than the sum of two integers `a` and `b`, and that neither `a` nor `b` is altered in the loop body. The loop could be written as:

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

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.

As another example, suppose you need a temporary variable within a loop to use for some calculation. The loop could be written as follows:

```
while(x < LIMIT)
{
   int tempTotal = a + b;
   // more statements here
}
```

When you declare a variable like `tempTotal` within a loop, it exists only for the duration of the loop; that is, it exists only until the loop's closing brace. Each time the loop executes, the variable is re-created. A more efficient solution is to declare the variable outside of the loop, as follows:

```
int tempTotal;
while(x < LIMIT)
{
   tempTotal = a + b;
   // more statements here
}
```

It is more efficient to declare this variable outside the loop than to redeclare it on every loop iteration.

Considering the Order of Evaluation of Short-Circuit Operators

In Chapter 4, you learned that the expressions in each part of an AND or 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 that a user can request any number of printed copies of a report from 1 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 value, then you want to start a loop that reprompts the user with the following expression:

```
while(requestedNum > 15 || requestedNum < 1)...
```

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 other loops, then the difference in the number of required tests increases. Similarly, the order of the evaluations in `if` statements is more important when the `if` statements are nested within a loop.

Employing Loop Fusion

Loop fusion is the technique of combining two loops into one. For example, suppose that 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. Performance issues can be crucial when you write programs for small mobile devices such as phones. 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 test program in Figure 5-20. It uses C#'s `Stopwatch` class. The statement `using System.Diagnostics;` is needed for the `Stopwatch` class. You will better understand how this class works after you read Chapter 9, "Using Classes and Objects," but you can use the class now by making statements similar to the following:

```
Stopwatch sw = Stopwatch.StartNew();
   // Place statements to be timed here
sw.Stop();
```

The first statement creates and starts a `Stopwatch` object for timing events. The object's name is `sw`—you can use any legal C# identifier. The last statement stops the `Stopwatch`.

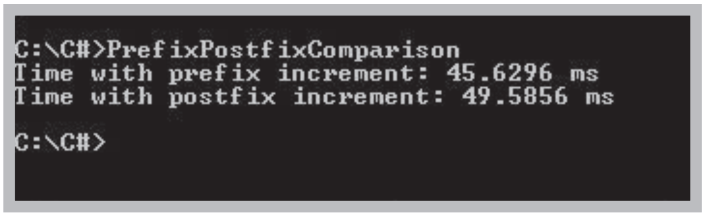
Subsequently, you can use a statement similar to the following to output the object's elapsed time in milliseconds:

```
WriteLine("Time used: {0} ms", sw.Elapsed.TotalMilliseconds);
```

Figure 5-20 contains a program that uses the `Stopwatch` class to compare loops that use prefix and postfix incrementing. Figure 5-21 shows the output of the program.

```
using System;
using static System.Console;
using System.Diagnostics;
class PrefixPostfixComparison
{
   static void Main()
   {
      int LOOPS = 10000000;
      Stopwatch sw = Stopwatch.StartNew();
      for(int x = 0; x < LOOPS; ++x);
      sw.Stop();
      Stopwatch sw2 = Stopwatch.StartNew();
      for(int x = 0; x < LOOPS; x++);
      sw2.Stop();
      WriteLine("Time with prefix increment: {0} ms",
         sw.Elapsed.TotalMilliseconds);
      WriteLine("Time with postfix increment: {0} ms",
         sw2.Elapsed.TotalMilliseconds);
   }
}
```

Figure 5-20 The `PrefixPostfixComparison` program



```
C:\C#>PrefixPostfixComparison
Time with prefix increment: 45.6296 ms
Time with postfix increment: 49.5856 ms

C:\C#>
```

Figure 5-21 Typical execution of the `PrefixPostfixComparison` program

The program that uses prefix incrementing runs slightly faster than the one that uses postfix incrementing. The difference in duration for the loops is very small, but it would increase if tasks were added to the loop bodies. If you run the PrefixPostfixComparison program multiple times, you will get different results, and occasionally the prefix operator loop might take longer than the postfix loop 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.

As you continue to study programming, you will discover many situations in which you can make your programs more efficient. You should always be on the lookout for ways to improve program performance without sacrificing readability.

TWO TRUTHS & A LIE

Improving Loop Performance

1. You can improve loop performance by making sure the loop does not include unnecessary operations or statements.
2. You can improve loop performance by declaring temporary variables outside of a loop instead of continuously redeclaring them.
3. You can improve loop performance by omitting the initialization of the loop control variable.

The false statement is #3. A loop control variable must be initialized for every loop.

Looping Issues in GUI Programs

Using a loop within a method in a GUI application is no different from using one in a console application; you can use `while`, `for`, and `do` statements in the same ways in both types of programs. For example, Figure 5-22 shows a GUI `Form` that prompts a user to enter a number and then displays *Hello* the corresponding number of times. The image on the left shows the `Form` when the program starts, and the image on the right shows the output after the user enters a value and clicks the button. Figure 5-23 shows the code in the `greetingsButton_Click()` method. When a user clicks the `greetingsButton`, an integer is extracted from the `TextBox` on the `Form`. Then a `for` loop appends *Hello* and a newline character to the `Text` property of the `outputLabel` the correct number of times.

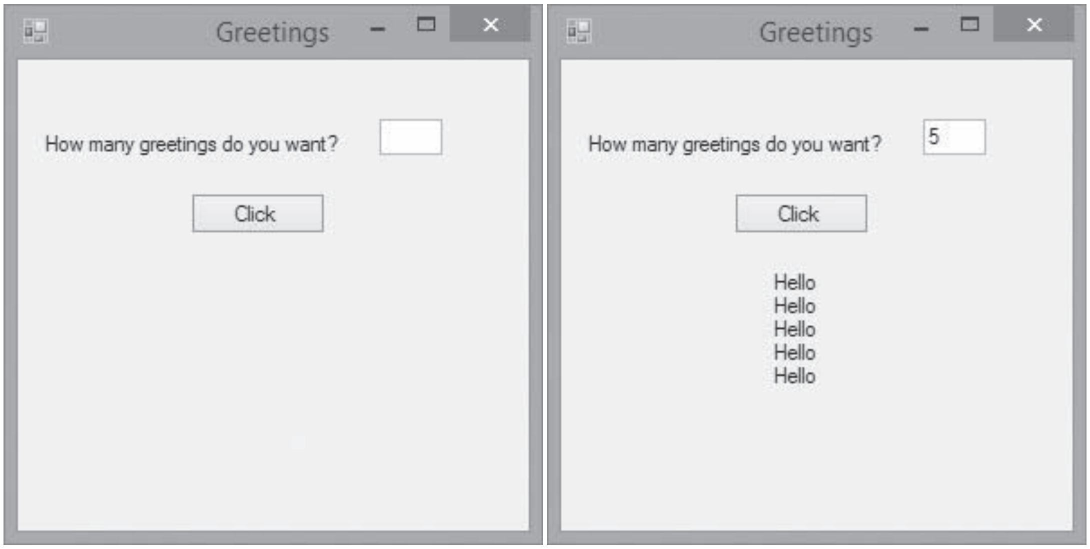

Greetings
How many greetings do you want?
Click
Greetings
How many greetings do you want?
5
Click
Hello
Hello
Hello
Hello
Hello

Figure 5-22 The Greetings `Form` when it starts and after the user enters a number and clicks the button

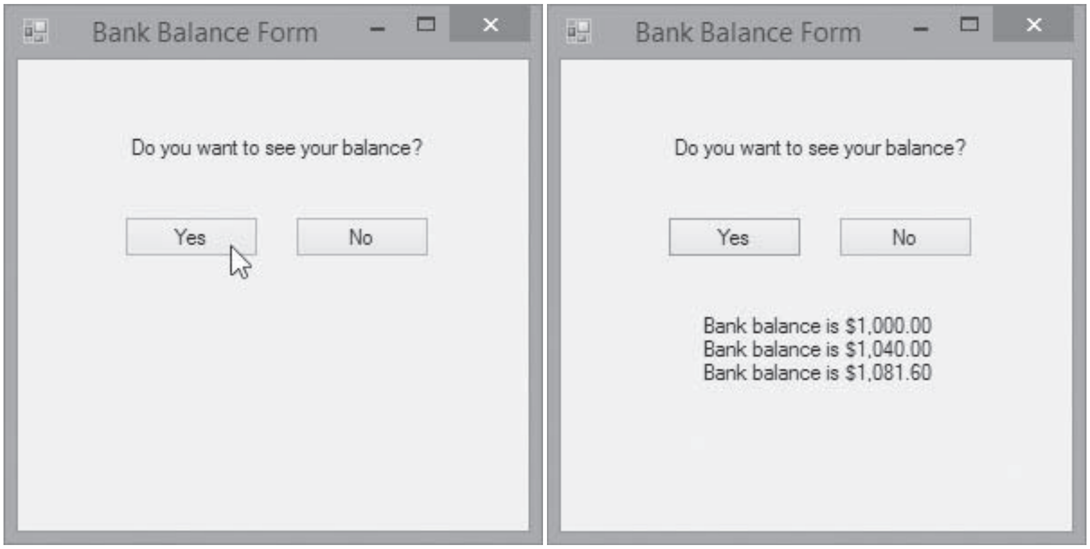
```
private void greetingsButton_Click(object sender, EventArgs e)
{
   int numGreetings = Convert.ToInt32(greetingsTextBox.Text);
   int count;
   for (count = 0; count < numGreetings; ++count)
      outputLabel.Text += "Hello\n";
}
```

Figure 5-23 The `greetingsButton_Click()` method in the `ManyHellosGUI` application

If the user clicked the button in the ManyHellosGUI program again, new instances of *Hello* would be added to the already-displayed ones. If your intention was to only show new *Hellos* then you could add the statement `outputLabel.Text = "";` to the `greetingsButtonClick()` method before the start of the loop.

Event-driven programs sometimes require fewer coded loops than their counterpart console applications, because in these programs some events are determined by the user's actions when the program is running, rather than by the programmer's coding beforehand. You can write an event-driven program so that an action continues as long as the user continues to make an appropriate selection. Depending on the application, the sentinel value for this sort of implicit loop might occur when the user either clicks a button that indicates *quit* or clicks the close button on the `Form` that hosts the application.

For example, Figure 5-24 shows a Form that controls a bank balance application similar to the one shown earlier in this chapter in Figure 5-7. The left side of the figure shows the Form when the application starts, and the right side shows the Form after the user has clicked the Yes button three times. Figure 5-25 shows the significant Form class code.


Bank Balance Form
Do you want to see your balance?
Yes
No
Bank Balance Form
Do you want to see your balance?
Yes
No

```
Bank balance is $1,000.00
Bank balance is $1,040.00
Bank balance is $1,081.60
```

Figure 5-24 The BankBalance Form when it starts and after the user has clicked *Yes* three times

```
namespace LoopingBankBalGUI
{
   partial class Form1 : Form
   {
      double bankBal = 1000;
      const double INT_RATE = 0.04;
      private void yesButton_Click(object sender, EventArgs e)
      {
         outputLabel.Text += String.Format("Bank balance is {0}\n",
            bankBal.ToString ("C"));
         bankBal = bankBal + bankBal * INT_RATE;
      }
      private void noButton_Click(object sender, EventArgs e)
      {
         outputLabel.Text = "Have a nice day!";
      }
   }
}
```

Figure 5-25 Code for the LoopingBankBalGUI program

In the LoopingBankBalGUI program, you might also choose to disable the *Yes* button or make it invisible so that the user cannot select it again after clicking *No*.

No loop is written in the code in Figure 5-25 because a loop body execution is caused automatically each time a user clicks the *Yes* button on the `Form`. Whenever `yesButton_Click()` executes, the `Text` of the `outputLabel` is appended to include a new line that displays the bank balance, and then the bank balance is increased by the interest rate. It is important that the `bankBal` variable is initialized outside the `yesButton_Click()` method; if it was initialized within the method, the balance would be reset to the original value of $1000 with each new button click. In the application in Figure 5-27, the balances in the `outputLabel` are replaced with *Have a nice day!* when the user indicates no more balances are needed.

Chapter Summary

- You can use a `while` loop to execute a body of statements continuously while a condition continues to be `true`. A `while` loop consists of the keyword `while`, a Boolean expression within parentheses, and the body of the loop, which can be a single statement or a block of statements surrounded by curly braces.
- A `for` statement includes loop control variable initialization, the test condition that controls loop entry, and the expression that alters the loop control variable. You begin a `for` statement with the keyword `for`, followed by a set of parentheses. Within the parentheses, the initialization, test, and update sections are separated by semicolons.
- The `do` loop checks the loop-controlling Boolean expression at the bottom of the loop after one repetition has occurred.
- You can nest any combination of loops.
- In computer programs, totals frequently are accumulated—that is, summed one at a time in a loop.
- You can improve loop performance by making sure the loop does not include unnecessary operations or statements, considering the order of evaluation for short-circuit operators, using comparisons to zero, employing loop fusion, and using prefix incrementing.
- You can use `while`, `for`, and `do` statements in the same ways in console and GUI programs. However, event-driven programs sometimes require fewer coded loops than console applications because some events are determined by the user's actions when the program is running, rather than by the programmer's coding beforehand.

Key Terms

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

A **loop body** is the block of statements executed in a loop.

An **iteration** is one execution of any loop.

A **`while` loop** executes a body of statements continuously while a test condition continues to be `true`; it uses the keyword `while`.

An **infinite loop** is one that (theoretically) never ends.

A **loop control variable** determines whether loop execution will continue.

An **empty body** has no statements in it.

Incrementing a variable means adding a value to it. (Specifically, the term often means to add 1 to a variable.)

Decrementing a variable means subtracting a value from it. (Specifically, the term often means to subtract 1 from a variable.)

In a **definite loop**, the number of iterations is predetermined.

A **counted loop** is a definite loop.

An **indefinite loop** is one in which the number of iterations is not predetermined.

A **sentinel value** is one that a user must supply to stop a loop.

A **`for` loop** contains 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 statement.

A **step value** is the amount by which a loop control variable is altered, especially in a `for` loop.

Out of scope describes a program component that is not usable because it has ceased to exist.

The **do loop** (also called a **`do...while` loop**) checks the loop-controlling Boolean expression at the bottom of the loop after one repetition has occurred.

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

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

An **inner loop** is the loop in a pair of nested loops that is entirely contained within another loop.

An **outer loop** is the loop in a pair of nested loops that contains another loop.

Accumulated totals are computed by adding values one at a time in a loop.

Garbage is a term used to describe an unknown memory value.

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

Review Questions

1. A structure that allows repeated execution of a block of statements is a(n) ______________________ .
   - a. sequence
   - b. selection
   - c. loop
   - d. array
2. The body of a `while` loop can consist of ______________________ .
   - a. a single statement
   - b. a block of statements within curly braces
   - c. either a or b
   - d. neither a nor b
3. A loop that never ends is called a(n) ______________________ loop.
   - a. `while`
   - b. `for`
   - c. counted
   - d. infinite
4. Which of the following is not required of a loop control variable in a correctly working loop?
   - a. It is reset to its initial value before the loop ends.
   - b. It is initialized before the loop starts.
   - c. It is tested.
   - d. It is altered in the loop body.
5. A `while` loop with an empty body contains no ______________________ .
   - a. loop control variable
   - b. statements
   - c. curly braces
   - d. test within the parentheses of the `while` statement
6. A loop for which you do not know the number of iterations when you write it is a(n) ______________________ .
   - a. definite loop
   - b. indefinite loop
   - c. counted loop
   - d. `for` loop

7. What is the major advantage of using a `for` loop instead of a `while` loop?
    a. With a `for` loop, it is impossible to create an infinite loop.
    b. It is the only way to achieve an indefinite loop.
    c. Unlike with a `while` loop, the execution of multiple statements can depend on the test condition.
    d. The loop control variable is initialized, tested, and altered all in one place.

8. A `for` loop statement must contain ______________________ .
    a. two semicolons
    b. three commas
    c. four dots
    d. five pipes

9. In a `for` statement, the section before the first semicolon executes ______________________ .
    a. once
    b. once prior to each loop iteration
    c. once after each loop iteration
    d. one less time than the initial loop control variable value

10. The three sections of the `for` loop are most commonly used for ______________________ the loop control variable.
    a. testing, outputting, and incrementing
    b. initializing, testing, and incrementing
    c. incrementing, selecting, and testing
    d. initializing, converting, and outputting

11. Which loop is most convenient to use if the loop body must always execute at least once?
    a. a `do` loop
    b. a `while` loop
    c. a `for` loop
    d. an `if` loop

12. The loop control variable is checked at the bottom of which kind of loop?
    a. a `while` loop
    b. a `do` loop
    c. a `for` loop
    d. all of the above

13. A `for` loop is an example of a(n) ______________________ loop.
    a. untested
    b. pretest
    c. posttest
    d. infinite

14. A `while` loop is an example of a(n) ________________ loop.
    a. untested
    b. pretest
    c. posttest
    d. infinite

15. When a loop is placed within another loop, the loops are said to be ________________ .
    a. infinite
    b. bubbled
    c. nested
    d. overlapping

16. What does the following code segment display?

```
a = 1;
while (a < 5);
{
   Write("{0} ", a);
   ++a;
}
```

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

17. What is the output of the following code segment?

```
s=1;
while(s < 4)
   ++s;
   Write("{0} ", s);
```

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

18. What is the output of the following code segment?

```
j = 5;
while(j > 0)
{
   Write("{0} ", j);
   j--;
}
```

    a. 0
    b. 5
    c. 5 4 3 2 1
    d. 5 4 3 2 1 0

19. What does the following code segment display?

```
for(f = 0; f < 3; ++f);
   Write("{0} ", f);
```

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

20. What does the following code segment display?

```
for(t = 0; t < 3; ++t)
   Write("{0} ", t);
```

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

Exercises

Programming Exercises

1. Write an application named **SumFourInts** that allows the user to enter four integers and displays their sum.
2. Write an application named **SumInts** that allows the user to enter any number of integers continuously until the user enters *999*. Display the sum of the values entered, not including 999.
3. Write an application named **EnterLowercaseLetters** that asks the user to type a lowercase letter from the keyboard. If the character entered is a lowercase letter, display *OK*; if it is not a lowercase letter, display an error message. The program continues until the user types an exclamation point.
4. Write an application named **TestScores** that continuously prompts a user for test scores until the user enters a sentinel value. A valid score ranges from 0 through 100. When the user enters a valid score, add it to a total; when the user enters an invalid score, display an error message. Before the program ends, display the number of scores entered and their average.
5. Danielle, Edward, and Francis are three salespeople at Holiday Homes. Write an application named **HomeSales** that prompts the user for a salesperson initial (*D*, *E*, or *F*). Either uppercase or lowercase initials are valid. While the user does not type *Z*, continue by prompting for the amount of a sale. Issue an error message for any invalid initials entered. Keep a running total of the amounts sold by each salesperson. After the user types *Z* or *z* for an initial, display each salesperson's total, a grand total for all sales, and the name of the salesperson with the highest total.
6. Write an application named **DisplayMultiplicationTable** that displays a table of the products of every combination of two integers from 1 through 10.
7. Write an application named **OddNums** that displays all the odd numbers from 1 through 99.
8. Write an application named **MultiplicationTable** that prompts the user for an integer value, for example *7*. Then display the product of every integer from 1 through 10 when multiplied by the entered value. For example, the first three lines of the table might read *1 X 7 = 7*, *2 X 7 = 14*, and *3 X 7 = 21*.

9. Write an application named **Sum500** that sums the integers from 1 through 500.
10. Write an application named **Perfect** that displays every perfect number from 1 through 1000. A number is perfect if it equals the sum of all the smaller positive integers that divide evenly into it. For example, 6 is perfect because 1, 2, and 3 divide evenly into it and their sum is 6.
11. In a "You Do It" section of this chapter, you created a tipping table for patrons to use when analyzing their restaurant bills. Now, create a modified program named **TippingTable3** in which each of the following values is obtained from user input:
    - The lowest tipping percentage
    - The highest tipping percentage
    - The lowest possible restaurant bill
    - The highest restaurant bill
12. Write a program named **WebAddress** that asks a user for a business name. Suggest a good Web address by adding *www.* to the front of the name, removing all spaces from the name, and adding *.com* to the end of the name. For example, a good Web address for Acme Plumbing and Supply is *www.AcmePlumbingandSupply.com.*
13. Write a program named **CountVowels** that accepts a phrase from the user and counts the number of vowels in the phrase. For this exercise, count both uppercase and lowercase vowels, but do not consider *y* to be a vowel.
14. In Chapter 4, you created a program that generates a random number, allows a user to guess it, and displays a message indicating whether the guess is too low, too high, or correct. Now, create a modified program called **GuessingGame2** in which the user can continue to enter values until the correct guess is made. After the user guesses correctly, display the number of guesses made.

Recall that you can generate a random number whose value is at least `min` and less than `max` using the following statements:

```
Random ranNumber = new Random();
int randomNumber;
randomNumber = ranNumber.Next(min, max);
```

15. Modify the GuessingGame2 program to create a program called **GuessingGame3** in which the player is criticized for making a "dumb" guess. For example, if the player guesses that the random number is 4 and is told that the guess is too low, and then the player subsequently makes a guess lower than 4, display a message that the user should have known not to make such a low guess.

Debugging Exercises

1. Each of the following files in the Chapter.05 folder of your downloadable student files has syntax and/or logical 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 *Fixed*. For example, save DebugFive1.cs as **FixedDebugFive1.cs**.

   a. DebugFive1.cs

   b. DebugFive2.cs

   c. DebugFive3.cs

   d. DebugFive4.cs

Case Problems

1. In Chapter 4, you created an interactive application named **GreenvilleRevenue** that prompts a user for the number of contestants entered in this year's and last year's Greenville Idol competition and displays the revenue expected for this year's competition if each contestant pays a $25 entrance fee. The program also displays one of three appropriate statements specified in the case problem in Chapter 4, based on a comparison between the number of contestants this year and last year. Now, modify the program so that the user must enter a number between 0 and 30, inclusive, for the number of contestants each year. If the user enters an incorrect number, the program prompts for a valid value.
2. In Chapter 4, you created an interactive application named **MarshallsRevenue** that prompts a user for the number of interior and exterior murals scheduled to be painted during a month and computes the expected revenue for each type of mural. The program also prompts the user for the month number and modifies the pricing based on requirements listed in Chapter 4. Now, modify the program so that the user must enter a month value from 1 through 12. If the user enters an incorrect number, the program prompts for a valid value. Also, the user must enter a number between 0 and 30 inclusive for the number of murals of each type; otherwise, the program prompts the user again.