

Repetition

Summing Up

You have now seen that classes are collections of related methods and data. You have seen constructors that initialize fields, properties to access object state, and other methods that implement the complex behavior of objects. Additionally, you have seen two important methods of control—sequence and selection. Sequential control occurs when every statement executes one after another. Selection (implemented in C# with the `if/else` and `switch` statements) lets you write code that executes differently under different circumstances.

Coming Up

This chapter introduces the third major control structure—repetition. Repetition is discussed within the context of two general algorithmic patterns—the Determinate Loop and the Indeterminate Loop. Repetition control allows a block of code statements to be repeatedly executed. The repetition can occur either a predetermined number of times, or until some event or condition occurs to terminate it. After studying this chapter, you will be able to

- implement loops in C# with the `while`, `for`, `do/while`, and `foreach` statements.
- recognize and use the Determinate Loop pattern to execute a set of statements a predetermined number of times.
- recognize and use the Indeterminate Loop pattern to execute a set of statements until some event occurs to stop it.
- use Determinate and Indeterminate loops in your programs, as appropriate.
- design loops of your own to solve programming problems.

6.1 Repetition Concepts

Repetition refers to the repeated execution of a set of statements. It occurs naturally in non-computer algorithms such as these:

- For every student's name on the attendance roster, call the name. Write an "X" if the student is absent, or a checkmark if the student is present.
- Practice playing the piano piece until you can play it well.
- Add the flour $\frac{1}{4}$ cup at a time, whipping until smooth.
- Run five laps around the school track.

Similarly, you can use repetition in computer algorithms. If something can be done once, it can be done repeatedly. Computer-based uses of repetition include:

- Process many customers at an automated teller machine (ATM).
- Continuously accept reservations.
- While there are more fast-food items, sum the price of each item.
- Compute the course grade for every student in a class.
- Microwave the food until the timer reaches 0, the cancel button is pressed, or the oven door is opened.

This chapter examines repetitive algorithmic patterns and the C# statements that implement them. It begins with a statement that executes a collection of actions a fixed, predetermined number of times. A statement that is used to implement repetition is often called a *loop*. Loops are useful for many tasks: to process report cards *for every* student in a school, to keep prompting for input *until* a valid option is entered, or to keep processing data *while* there is more data to process.

Why Is Repetition Needed?

Many jobs formerly done by hand are now performed much more quickly by computers. Think of a payroll department that produces employee paychecks. With only a few employees, this task could certainly be done by hand. But with thousands of employees, a very large payroll department would be needed to compute and generate that many paychecks, by hand, in a timely fashion. Other situations that require repetition include finding an average, searching a collection of objects for a particular item, alphabetizing a list of names, and processing all the data in a file. Luckily, computers are great for quickly completing such repetitive tasks.

Consider the following code that finds the average of exactly three numbers. No repetitive control is used (this code will be improved shortly):

Code Sample: AverageThree Class

```
1 using System;
2
3 class AverageThree
4 {
5     static void Main()
6     {
7         double sum = 0.0;    // stores the sum of all inputs
8         double input;       // will be read many times below
9
10        // The following statements will be repeated several times.
11        Console.Write("Enter number: ");
12        input = double.Parse(Console.ReadLine());
13        sum += input;
14
15        Console.Write("Enter number: ");
16        input = double.Parse(Console.ReadLine());
17        sum += input;
18
19        Console.Write("Enter number: ");
20        input = double.Parse(Console.ReadLine());
21        sum += input;
22
23        double average = sum / 3.0;
24        Console.WriteLine("Average: {0}", average);
25    }
26 }
```

This program uses a *brute force* algorithm. This means that it uses a straightforward approach, doing each task one after the other, until all tasks are done. There is a drawback to this approach to repetition. This program is not general enough to handle input sets of various sizes. If it ever needs to average a larger set of inputs, it must be modified. As currently written, it repeats the same statements three times (lines 11–13, 15–17, and 19–21). Averaging 100 numbers would require 97 additional copies of these three statements. Also, in line 23, the constant 3.0 in `average = sum / 3.0` would have to be changed to 100.0.

A better algorithm would be adaptable to changes, such as needing to average an arbitrary number of terms. To solve this problem, we will use a form of control that can execute one copy of these three statements three times. We could do this writing a method that contained the repeated statements, and calling the method three times. However, if we wanted to call the method 100 times, we would still have to write 100 lines of method calls. This is wasteful, easy to miscount, and hard to modify later.

6.2 The while Loop

Without the selection control structures of the preceding chapter, computers are little more than nonprogrammable calculators. Selection control lets computers adapt to varying situations. But what really makes computers powerful is their ability to repeat the same actions quickly, accurately, and efficiently.

As mentioned previously, a statement that executes code multiple times is called a *loop*. The *while loop statement* is the simplest way to program a loop in C#. A `while` loop is a sequence of statements that keep executing *while* some Boolean condition is true. Here is its general form:

General Form: while loop statement

```
while (Boolean expression)  
{  
    statement(s) to execute;  
}
```

Notes:

The braces are optional if only one statement is used.

Examples:

```
int i = 0;  
int max = 10;  
while (i < max)  
{  
    Console.WriteLine(i);  
    i++;  
}  
  
string input = null;  
while (input != "OK")  
{  
    input = Console.ReadLine();  
}
```

The *Boolean expression* above is a `bool` expression that evaluates to either `true` or `false`. As with the bodies of `if` statements (discussed in the preceding chapter), the *statement(s) to execute* may be a block of C# statements, or a single statement.

Most of the examples in this textbook use a block enclosed in `{` and `}`, but the following is also legal, as long as only one statement is executed as a result of the `while` loop:

```
while (input != "OK")           // no {} braces...
    input = Console.ReadLine(); // legal syntax, but discouraged
```

When a `while` loop is encountered, the *Boolean expression* is evaluated to either `true` or `false`. If the expression evaluates to `false`, the loop is skipped, and the statement(s) inside it is not executed. However, if the expression is `true`, the statements in the body execute once, after which the *Boolean expression* is evaluated again. This process continues as long as the expression is `true`. In this way, the statements to execute (the “body” of the loop) can be executed many times, depending on the *Boolean expression* used as the loop’s condition.

The following figure shows the flow of execution of a `while` loop:

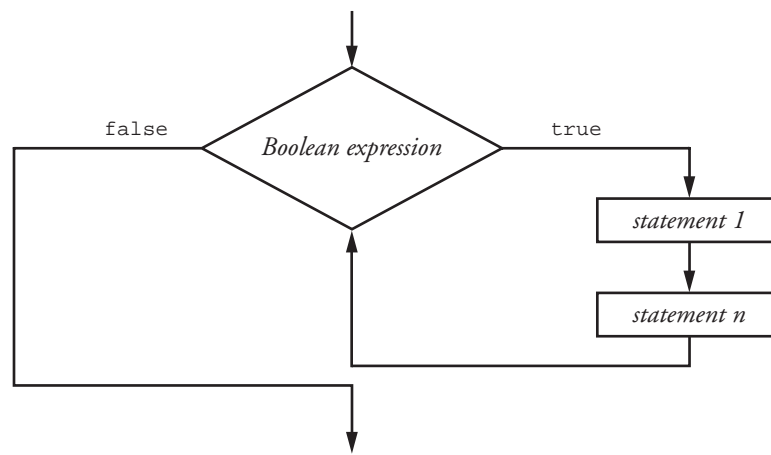


Figure 6.1: while Loop Execution Flow Chart

The statements in the body of a `while` loop might execute zero times, once, or many times. Each time that the statements in a loop’s body execute is called one *iteration* of the loop. A loop that repeats ten times, therefore, has undergone ten iterations.

The `AverageThree` class from the preceding section can be rewritten using a `while` loop with three iterations, as follows:

Code Sample: AverageThree Class with while loop

```
1 using System;
2
3 class AverageThree
4 {
5     static void Main()
6     {
7         double sum = 0.0;    // stores the sum of all inputs
8         double input;        // will be read many times below
9
10        int numIterations = 3;
11        int i = 0;
12
13        while (i < numIterations)
14        {
15            // The following statements will be repeated 3 times.
16            Console.WriteLine("Enter number: ");
17            input = double.Parse(Console.ReadLine());
18            sum += input;
19
20            i++;
21        }
22
23        double average = sum / numIterations;
24        Console.WriteLine("Average: {0}", average);
25    }
26 }
```

Looking at the above code, it may not be immediately obvious that it is equivalent to the original program. Walking through the code execution visually is helpful in proving that the code behaves as desired:

```
int numIterations = 3;
int i = 0;

while (i < numIterations)
```

Is `i < numIterations` true? Currently `i` has the value 0 and `numIterations` has the value 3; so yes, this is true. So, the statements in the loop execute once.

```

Console.Write("Enter number: ");
input = double.Parse(Console.ReadLine());
sum += input;

i++;

```

One input is read from the console, and the value of `i` is increased by 1. Now we start the loop over, by re-evaluating its Boolean expression:

```
while (i < numIterations)
```

Is `i < numIterations` true? Now `i` has the value 1 and `numIterations` has the value 3; so yes, this is still true. The statements in the loop execute a second time.

```

Console.Write("Enter number: ");
input = double.Parse(Console.ReadLine());
sum += input;

i++;

```

A second input is read from the console, and the value of `i` is again increased by 1. Now we start the loop over, re-evaluating its Boolean expression again:

```
while (i < numIterations)
```

Is `i < numIterations` true? Now `i` has the value 2 and `numIterations` has the value 3; so yes, this is still true. Therefore, the statements in the loop execute a third time.

```

Console.Write("Enter number: ");
input = double.Parse(Console.ReadLine());
sum += input;

i++;

```

A third input is read from the console, and the value of `i` is again increased by 1. Now we start the loop over, re-evaluating its Boolean expression again:

```
while (i < numIterations)
```

Is `i < numIterations` true? This time, `i` has the value 3 and `numIterations` has the value 3; so *no*, the Boolean expression has now become `false`. Therefore, the loop stops executing.

Chapter 6 Repetition

Loops are very powerful and can be used to accomplish many programming tasks. Here are some example loops:

```
// 1. read integer from console, and say hello that many times
Console.Write("How many times? ");
int numTimes = int.Parse(Console.ReadLine());
while (numTimes > 0)
{
    Console.WriteLine("Hello!");
    numTimes--;
}
```

One Possible Dialogue

```
How many times? 4
Hello!
Hello!
Hello!
Hello!
```

```
// 2. keep prompting the user until a valid option is given
string option = null;
while (option != "a" && option != "r" && option != "f")
{
    Console.Write("(A)bort, (R)etry, (F)ail? ");
    option = Console.ReadLine();
}
```

One Possible Dialogue

```
(A)bort, (R)etry, (F)ail? q
(A)bort, (R)etry, (F)ail? x
(A)bort, (R)etry, (F)ail? a
```

```
// 3. generate a random number that is divisible by 6
Random rand = new Random();
int randomNumber = rand.Next();
while (randomNumber % 6 != 0)
{
    randomNumber = rand.Next();
    Console.WriteLine(randomNumber);
}
```

One Possible Result

```
342154
145
6042
```

Remember that once a `while` loop starts executing, it executes until the Boolean expression that controls it becomes `false`. The `++`, `+=`, `*=`, and other increment-and-assign operators introduced in Chapter 2 are often used to modify values inside a loop, so that the controlling expression eventually does become `false`. Which operator is appropriate to use depends on the loop's purpose. If the loop is counting something, or if it is repeating an action for every part of a whole, the `++` operator is usually the right choice.

For example, the following code prints each character of a `string` on a separate line:

```
string s = "Howdy";
int index = 0;
while (index < s.Length)
{
    Console.WriteLine("Letter {0} is {1}",
                      index, s[index]);
    index++;
}
```

Output

```
H
o
w
d
y
```

The `+=`, `-=`, and other modify-and-assign operators are useful when you wish to increment (increase) or decrement (decrease) loop control variables by values other than 1. In the next example, the loop control variable `j` increments by 2 at the end of each iteration:

```
int j = 0;
while (j <= 10)
{
    Console.Write("{0} ", j);
    j += 2;    // Count by twos
}
```

Output

0 2 4 6 8 10

The following example prints powers of 2, up to 1024:

```
int j = 1;
while (j <= 1024)
{
    Console.Write("{0} ", j);
    j *= 2;
}
```

Output

1 2 4 8 16 32 64 128 256 512 1024

Self-Check

6-1 Write the output generated by the following code:

(a)

```
int j = 1;

while (j <= 5)
{
    Console.Write(j);
    j++;
}
```

(b)

```
int j = 5;

while (j > 0)
{
    j--;
    Console.Write(j);
}
```

6-2 Rewrite the following code fragments to use `while` loops:

(a)

```
Console.WriteLine(2);
Console.WriteLine(1);
Console.WriteLine(0);
```

(b)

```
Console.WriteLine("a");
Console.WriteLine("aa");
Console.WriteLine("aaa");
```

(c)

```
Console.WriteLine(1);
Console.WriteLine(2);
Console.WriteLine(4);
Console.WriteLine(8);
```

(d)

```
Console.WriteLine(2);
Console.WriteLine(4);
Console.WriteLine(16);
Console.WriteLine(256);
```

6-3 Write code that asks the user to enter an integer, and then sums all integers that the user enters, until a negative integer is entered.

6-4 Write code that asks to user to enter integers, until the difference between two consecutive integers is greater than 100. Assume that there are always at least two inputs. The number of inputs may be even or odd and can range from 2 inputs to many. Print the values of the first two consecutive numbers that differ by more than 100.

Infinite Loops and Empty Loops

It is possible that a `while` loop will never execute, if its Boolean expression is `false` initially. It is also possible that a `while` loop will never terminate. A loop that never stops running (unless the program is forcefully shut down) is called an *infinite loop*.

Infinite loops are usually undesirable. The simplest example of an infinite loop is the `while (true)` loop, shown below:

```
while (true)
{
    Console.WriteLine("This is an infinite loop!");
}
```

Since the Boolean expression for the above loop is simply `true`, the loop's condition will never be `false`. The loop will continue executing forever.

Unfortunately, it is also possible to write infinite loops unintentionally, as shown in the following example. The loop below is an infinite loop; the condition, `i >= 0`, is initially `true`, and (as currently written) the loop body will never cause it to become `false` (`i = i;` should be `i = i + 1;`).

```
int i = 0;
while (i <= 10)
{
    Console.WriteLine(i);
    i = i; // Need to update the value of i
}
```

When you write programs that use loops, it is easy to accidentally write an infinite loop. Usually, infinite loops are easy to spot, because the program will print endless output or will hang indefinitely.

To avoid infinite loops, always check the following things:

- Are the variables before the loop initialized correctly?
- Is the loop test correct?
- Does the body of the loop perform some action that will eventually terminate the loop?

Unfortunately, it is also legal to write a loop with an empty body; this loop will do nothing! This type of loop is called an *empty loop*. Most of the time, an empty loop occurs as a result of a typing or intent error. Empty loops often end up also being infinite loops. Because their body is empty, they cannot perform any update step to make the loop terminate.

The following is an empty (and infinite) loop:

```
int i = 0;
while (i < 10)
{
    // no statements in the loop body!
}
```

It might seem easy to identify empty infinite loops, but they can be tricky to spot. For example, the following is also a legal loop, whose body consists of a single empty statement:

```
int i = 0;
while (i < 10) // the loop's body is the semicolon below
    ;
```

Because C# is not sensitive to whitespace, and because method calls usually have semicolons after them, programmers often write something like the following, by mistake:

```
int i = 0;
while (i < 10); // The loop's body is the semicolon on this line!
    i++; // The programmer wanted this to be the body!
```

As you can see, it is easy to accidentally write an infinite or empty loop without intending to do so. As mentioned above, the symptom of such a loop is that the program either prints endless output, or it runs forever, without producing output.

If either of these things happens, close the program manually and inspect each loop carefully to correct mistakes like those above. Also, follow the steps outlined in this section for making sure that each loop will eventually terminate. This is also a strong argument for always using the braces `{` and `}` around the body of a loop, even if it only contains one statement. The preceding errors would not be possible with such a brace-surrounded block of statements.

Self-Check

6-5 Write the output from the following C# program fragments:

(a)

```
int n = 3;
int counter = 1;
while (counter <= n)
{
    Console.WriteLine("{0} ", counter);
    counter++;
}
```

(b)

```
int last = 10;
int j = 2;
while (j <= last)
{
    Console.WriteLine("{0} ", j);
    j += 2;
}
```

Chapter 6 Repetition

(c)

```
counter = 10;
// Tricky, but an easy-to-make mistake
while (counter >= 0);
{
    Console.WriteLine(counter);
    counter++;
}
```

6-6 Write the number of times that “Hello” is printed. “Zero” and “Infinite” are valid answers.

(a)

```
int counter = 1;
int n = 20;
while (counter <= n)
{
    Console.Write("Hello ");
    counter++;
}
```

(b)

```
int counter = 1;
int n = 0;
while (counter <= n)
{
    Console.Write("Hello ");
}
```

(c)

```
int counter = 1;
int n = 5;
while (counter <= n)
{
    Console.Write("Hello ");
    counter++;
}
```

(d)

```
int n = 5;
int j = 1;
while (j <= n)
```

```

    Console.WriteLine("Hello ");
    j++;

```

(e)

```

int j = 1;
int n = 5 ;
while (j <= 5)
{
    Console.WriteLine("Hello ");
    n++;
}

```

(f)

```

int j = 2;
int n = 1024;
while (j <= n)
{
    Console.WriteLine("Hello ");
    j = j * 2;
}

```

6-7 Describe how to fix each of the following infinite loops:

```

int sum = 0;
int j = 1;

```

(a)

```

while (j <= 100)
{ // Sum the first 100 integers
    sum += j;
}

```

(b)

```

while (j <= 100);
{ // Sum the first 100 integers
    sum += j;
    j++;
}

```

(c)

```

while (j <= 100)
    // Sum the first 100 integers
    sum += j;
    j++;

```

6.3 Determinate and Indeterminate Loops

By looking at the many `while` loop examples in the previous section, you can see that loops can be categorized based on how many times they will execute. Some loops never execute at all; some execute a fixed number of times; some execute an unfixed number of times until a condition is met; and some execute forever!

A key observation here is that many loops execute a specific number of times. For example, to find the average of 142 test grades, you would repeat a set of statements exactly 142 times. To pay 89 employees, you would repeat a set of statements 89 times. To produce grade reports for 32,675 students, you would repeat a set of statements 32,675 times. This generalizes; if there are n students, you would repeat the statements n times to print the reports.

To create a loop that executes a fixed number of times, you must put that number in the C# program code. This number should be stored either as a literal integer, or as a variable's value. It must be established before the loop begins to execute. Executing statements a predetermined number of times this way is referred to here as the *Determinate Loop pattern*.

Algorithmic Pattern: Determinate Loop

Problem: Do something exactly n times, where n is known in advance.

Outline: Determine n as the number of times to repeat the actions.
Repeat the following n times.
Execute the actions to be repeated.

Example:

```
double sum = 0.0;
double input;
int count = 0;
while (count < 10)
{
    // these statements will be executed 10 times
    Console.Write("Enter a number: ");
    input = double.Parse(Console.ReadLine());
    sum += input;
    count++;
}
```

The Determinate Loop pattern uses an `int` variable—named `n` here—to represent how often the actions must repeat. However, you can use any appropriate variable name, such as `numberOfEmployees`.

The first thing to do in the Determinate Loop pattern is to determine the number of iterations. This number might be known at compile time. In this case, it would be written into the code as a constant value, such as the constant 10 in the code example above. Alternatively, the number of repetitions might come from user input, such as `int n = int.Parse(Console.ReadLine());`. The value might even be passed as an argument to a method, as in `public void Display(int n)`. Once the number of repetitions is set, another `int` variable (named `count` in the program above) controls the number of loop iterations.

However, not all loops are determinate; sometimes it is not possible to know exactly how many times a loop will execute. However, just because a loop executes an unknown number of times does not mean it is an infinite loop. Consider the following example:

```
// generate a random number that is divisible by 6
Random rand = new Random();
int randomNumber = rand.Next();
while (randomNumber % 6 != 0)
{
    randomNumber = rand.Next();
}
```

How many times will the above loop execute? Is it an infinite loop? One could imagine a scenario where the `rand` number generator produces infinitely many numbers that are not divisible by 6. This would cause the loop to keep executing forever. However, this becomes more and more unlikely as more random numbers are chosen, so over time there is only an infinitely small chance that a number divisible by 6 will not have been chosen. Therefore, the above loop would not be considered the infinite loop.

Although the Determinate Loop pattern occurs in many algorithms, it has a serious limitation. Somehow, you must determine the number of repetitions in advance. Quite often this is impossible, or at least is very inconvenient or difficult. For example, an instructor may have a different number of tests to average as attendance varies between school terms. A company may not have a constant number of employees because of hires, fires, layoffs, transfers, and retirements. Also, consider a program with a loop based on user input: perhaps the program loops until the user enters a certain string or number (such as “quit” or `-1`). How many times should this loop run? The result is not fixed and cannot be predicted ahead of time.

The *Indeterminate Loop pattern* describes this idea of repetition, where the number of iterations is not constant or is not known in advance. With indeterminate loops, repetitions can continue an arbitrary number of times. Generally, an indeterminate loop continues executing until some special event occurs, or some special unpredictable condition is met.

Algorithmic Pattern: Indeterminate Loop

Pattern:	Indeterminate Loop
Problem:	A process must repeat an unknown number of times, so some event is needed to terminate the loop.
Outline:	while the termination event has not occurred, { Perform these actions. Do something to bring the loop closer to termination. }
Example:	<pre>// Continue until user wants to quit. int n = 0; while (n != -1) { Console.Write("Enter a number (-1 to quit): "); n = int.Parse(Console.ReadLine()); Console.WriteLine("You entered {0}", n); }</pre>

Extended examples of determinate and indeterminate loops will be shown later in this chapter.

6.4 The for Loop

As discussed in the previous section, for many loops the number of iterations is known in advance; these are called determinate loops. Also, it is very common to declare a special “counter variable” that is used only to govern the number of loop iterations. At the end of the loop, the counter variable is updated in some way that brings the loop closer to termination. The loop below should look similar to many in this chapter:

```
int j = 0; // initialize counter variable
while (j < 10) // boolean test
{
    Console.Write("{0} ", j); // loop body
    j++; // increment counter variable
}
```

Since this determinate pattern of loop is so common, C# includes an additional looping statement that is tailored to this pattern. C#'s `for` statement provides a loop that is designed to implement the Determinate Loop pattern. The `for` statement begins with the keyword `for`, followed by a three-part parenthesized section: an initialization, a Boolean test for determining when to stop repeating, and an update step to perform after each repetition. The update step is generally used to bring the loop one step closer to terminating.

The `for` loop was added to programming languages because the Determinate Loop pattern arises so often. Here is the general form of the C# *for loop statement*:

General Form: for loop statement

```
for (initialization statement; Boolean expression; update statement(s))
{
    statement(s) to execute;
}
```

Notes:

The braces are optional if only one statement is used.

Examples:

```
for (int j = 0; j < 10; j++)
{
    Console.WriteLine(j);
}

double sum = 0.0;
for (double pow = 10.0; pow >= 1.0; pow -= 2.0)
{
    sum += Math.Pow(2, pow);
    Console.WriteLine("Sum so far is {0}", sum);
}
```

The `for` statement's syntax is admittedly confusing at first. But it is a useful statement because it combines several individual steps into a single compact entity. The confusing part about its syntax is understanding what is executed when.

When a `for` loop is first encountered, the *initialization statement* is executed first, exactly one time. Next, the *Boolean expression* is evaluated to either `true` or `false`; this is done before each iteration of the loop. Assuming that the Boolean test is initially true, the *statement(s) to execute* are executed. After these statement(s) have been executed, the program executes the *update statement(s)*. After the update, the Boolean test is re-evaluated. If it is still `true`, the loop executes again. This process continues until the loop test evaluates to `false`. Here is a descrip-

Chapter 6 Repetition

tion of the flow of execution of a `for` loop:

1. Execute initialization statement.
2. Evaluate Boolean expression.
3. If Boolean test is false, stop.
4. Otherwise (if the test is true),
 - Execute statement(s) to execute.
 - Execute update statement(s).
 - Go back to step 2.

The following figure also summarizes the behavior of a `for` loop:

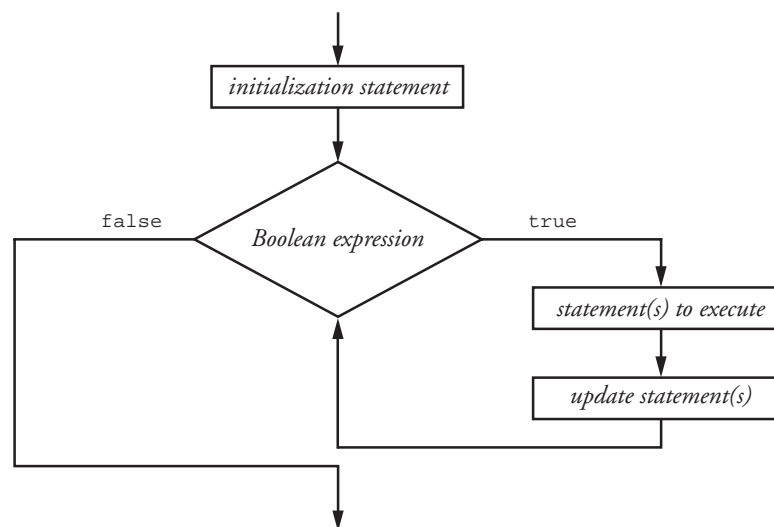


Figure 6.2: For loop flow chart

Here is the `while` loop shown at the start of this section, converted into an equivalent `for` loop:

```
for (int j = 0; j < 10; j++)  
{  
    Console.WriteLine("{0} ", j);    // loop body  
}
```

Output

```
0 1 2 3 4 5 6 7 8 9
```

In the preceding `for` loop, `j` is first assigned the value of 0. Next, the Boolean test `j < 10` is evaluated; it results in a value of `true`, because `0 < 10`; so the block executes. When the statements inside the block are done, the update step is executed, incrementing `j` by 1 (`j++`). These three components ensure that the block executes precisely `n` times. To recap:

```
int j = 0;           // Initializes the counter
j < 10;            // A test to see if the loop should continue repeating
j++               // Updates the counter
```

Like an `if` statement or `while` loop, a `for` loop can have a single statement as its body, rather than an entire block of code. The loop above could have been written as:

```
for (int j = 0; j < 10; j++)
    Console.WriteLine("{0} ", j);           // 1-line loop body, no braces
```

However, as stated before, consider always using a block with braces when writing a `for` loop. This helps avoid difficult-to-detect logic errors.

Legal syntax, but discouraged:

```
for (int i = 0; i < 10; i++)
    Console.WriteLine(i);
```

Encouraged:

```
for (int i = 0; i < 10; i++)
{
    Console.WriteLine(i);
}
```

The `for` loop is shown next in the context of a small program that improves the design of the task to average a set of numbers. In this example, the `for` statement implements the repetition needed to solve the problem.

Code Sample: DoAverage Class

```
1 using System;
2
3 // This class accumulates and averages numbers.
4 class Averager
5 {
6     private int n = 0;
7     double sum = 0.0;
```

Chapter 6 Repetition

```
8
9 // Adds the given number to the running total.
10 public void AddNumber(double num)
11 {
12     n++;
13     sum += num;
14 }
15
16 // The current average of all numbers entered so far.
17 public double Average
18 {
19     get
20     {
21         if (n == 0)
22             return 0.0;
23         else
24             return sum / n;
25     }
26 }
27 }
28
29 // Demonstrates a determinate loop using the Averager class.
30 class TestAverage
31 {
32     static void Main()
33     {
34         Averager averager = new Averager();
35
36         Console.Write("How many numbers do you need to average? ");
37         int n = int.Parse(Console.ReadLine());
38
39         // Now accumulate n numbers
40         for (int j = 0; j < n; j++)
41         {
42             // Repeat these statements n times
43             Console.Write("Enter number: ");
44             double input = double.Parse(Console.ReadLine());
45             averager.AddNumber(input);
46         }
47
48         // Compute the average only when there is at least one input
49         Console.WriteLine("Average of {0} numbers is {1:F1}",
50                             n, averager.Average);
51     }
52 }
```

Dialogue

```

How many numbers do you need to average? 4
Enter number: 70.0
Enter number: 80.0
Enter number: 90.0
Enter number: 86.0
Average of 4 numbers is 81.5

```

The following trace of the execution shows how the `sum` variable in the `Averager` class accumulates the numbers as the loop repeats the block of statements four times.

Loop Number	<code>j</code>	input	sum
Before loop	NA	?	0.0
1	1	70.0	70.0
2	2	80.0	150.0
3	3	90.0	240.0
4	4	86.0	326.0

Self-Check

- 6-8 Does a `for` loop execute the update step at the beginning of each iteration?
- 6-9 Must an update step increment the loop counter by `+ 1`?
- 6-10 Do `for` loops always execute the repeated part at least once?
- 6-11 Describe a situation when the loop test `j <= n` of a `for` loop never becomes false and the loop never terminates on its own.
- 6-12 Write the output generated by the following `for` loops.

(a)

```

for (int j = 1; j < 5; j = j + 1)
{
    Console.WriteLine("{0} ", j);
}

```

Chapter 6 Repetition

(b)

```
int n = 5;
for (int j = 1; j <= n; j++)
{
    Console.WriteLine("{0} ", j);
}
```

(c)

```
int n = 3;
for (int j = -3; j <= n; j += 2)
{
    Console.WriteLine("{0} ", j);
}
```

(d)

```
for (int j = 0; j < 5; j++)
{
    Console.WriteLine("{0} ", j);
}
```

(e)

```
for (int j = 5; j >= 1; j--)
    Console.WriteLine("{0} ", j);
```

(f)

```
int n = 0;
Console.WriteLine("before ");
for (int j = 1; j <= n; j++)
{
    Console.WriteLine("{0} ", j);
}
Console.WriteLine(" after");
```

- 6-13** Write a `for` loop that displays all of the integers from 1 to 100 inclusive, on separate lines.
- 6-14** Write a `for` loop that displays all of the integers from 10 down to 1 inclusive, on separate lines.

Equivalence of `while` and `for` Loops

Though `while` and `for` loops look very different, they have the same computational and expressive power. To convert a `for` loop into an equivalent `while` loop, first move the initialization to before the `while` loop. Then move the update step to the bottom of the repeated statements to execute.

```
initialization
while (Boolean-test)
{
    // Activities to be repeated
    update-step
}
```

The following code represents an alternate implementation of a determinate loop:

```
// Sum the first n integers
int sum = 0;           // Initialization
int j = 1;            // Initialization
int n = 5;            // Initialization
while (j <= n)       // Loop test
{
    sum += j;         // Action
    j++;             // Update step
}
```

```
Console.WriteLine("Sum of the first {0} integers is {1}", n, sum);
```

Converting a `while` loop to an equivalent `for` loop is similar. Though not all `while` loops seem to follow the pattern that `for` loops require (a counter variable that is initialized and updated at each step), any `while` loop can be converted to a `for` loop that has the exact same behavior. Consider the following `while` loop:

```
// generate a random number that is divisible by 6
Random rand = new Random();
int num = rand.Next();
while (num % 6 != 0)
{
    num = rand.Next();
}
```

The equivalent `for` loop would look like this:

Chapter 6 Repetition

```
Random rand = new Random();
for (int num = rand.Next(); num % 6 != 0; num = rand.Next())
{
}
```

Notice that the `for` loop body here is empty! This does not mean that the loop serves no purpose, however. Its purpose is that its update step, `num = rand.Next()`, will keep executing until the Boolean test `num % 6 != 0` becomes `true`.

The syntax of `for` loops is somewhat flexible. Any of the three parts of the `for` loop header (the initialization, the Boolean test, and the update step) may be left empty (the initialization, test and update must still be separated by semicolons, even if empty). If the initialization is omitted, no initialization is performed. If the Boolean test is omitted, the test is assumed to be `true` (which generally leads to infinite loops!). If the update step is omitted, the loop starts each iteration without any update being performed. So the infinite `for` loop (sometimes called the “forever loop”) equivalent to the infinite `while (true)` loop looks like this:

```
for (;;) // no initialization, test, or update steps
{
    // do something here
    Console.WriteLine("This is an infinite loop!");
}
```

Also, the initialization statement and update statements can actually be multiple initializations and updates, separated by commas.

```
for (int pow = 1, result = 2; pow < 10; pow++, result *= 2)
{
    Console.WriteLine("2 to the {0} = {1}", pow, result);
}
```

Output

```
2 to the 1 = 2
2 to the 2 = 4
2 to the 3 = 8
2 to the 4 = 16
2 to the 5 = 32
2 to the 6 = 64
2 to the 7 = 128
2 to the 8 = 256
2 to the 9 = 512
```

Although the `while` loop can implement determinate loops, the `for` loop is more concise and convenient. It is recommended that you use the `for` loop when the number of iterations is known in advance. When this cannot be determined, use the `while` statement instead.

6.5 Loop Examples and Applications

This section shows how loops can be used to solve larger programming problems. One major example is shown for the Determinate Loop pattern, and one for the Indeterminate Loop pattern. A key concept from this section is that it is important to analyze a problem to decide what type of repetition, if any, is needed to solve it, and to choose the appropriate loop to solve the problem.

Application of Determinate Loop Pattern: Temperature Ranges

Problem: Write a program that determines a range of temperature readings. “Range” is defined as the difference between the highest and lowest. The number of temperature readings will be known in advance. One dialogue would look like this:

Dialogue

```
Enter 6 temperature readings:
11
15
19
23
20
16
Range: 13
```

To find the range without the aid of a computer (this is easier with a small number of temperature readings), you could look at the list of numbers and simply keep track of the highest and lowest ones, while scanning the list from top to bottom.

Let’s walk through some temperatures. `aTemp` is the name assigned to the user input.

<i>aTemp</i>	<i>highest</i>	<i>lowest</i>
-5	-5	-5
8	8	-5
22	22	-5
-7	22	-7
15	22	-7

For this set of data, $\text{range} = \text{highest} - \text{lowest} + 1 = 22 - (-7) + 1 = 30$.

Let the number of temperature readings be n . Again, the variable named `aTemp` stores the individual temperature readings as each is read in. The range of temperature readings is the difference between the highest and lowest temperature readings in the list of inputs, + 1. The following table summarizes the problem:

Problem Description	Variable Name	Sample Values	Input/Output
Compute the range of temperature readings	<code>n</code>	5	Input
	<code>aTemp</code>	-5, 8, 22, -7, 15	Input
	<code>highest</code>	22	
	<code>lowest</code>	-7	
	<code>range</code>	30	Output

For a large list—an approach more suited to a computer—the algorithm mimics the repetition of the hand-operated version just suggested. It uses a determinate loop to compare every temperature reading in the list to the highest and the lowest readings, and to update these readings, if necessary.

Algorithm: Determining the range for each temperature reading

```

for each temperature reading
{
  input aTemp from user
  if aTemp is greater than highest so far,
    store it as highest
  if aTemp is less than lowest so far,
    store it as lowest
}
range = highest - lowest + 1

```

As usual, it is a good idea to walk through the algorithm to verify its soundness.

1. Input the number of temperature readings ($n == 5$).
2. Input `aTemp` from the user ($aTemp == -3$).
3. If `aTemp` is greater than highest so far ($-3 > . . .$). Whoops!

There is no value yet for `highest`! And there is no value yet for `lowest`. To fix this, let's assume that the program will initialize both `highest` and `lowest` to 0:

1. Initialize lowest and highest to 0 (`lowest = 0; highest = 0;`).
2. Input the number of temperature readings (`n == 5`).
3. Input `aTemp` from the user (`aTemp == -3`).
4. If `aTemp` is greater than `highest` (`-3 > 0`), store `aTemp` as `highest` (`highest` stays the same).
5. If `aTemp` is less than `lowest` so far (`-3 < 0`), store `aTemp` as `lowest` (`lowest = -3`).

This seems to work. Consider one more iteration.

1. Input `aTemp` from the user (`aTemp == 8`).
2. If `aTemp` is greater than `highest` so far (`8 > 0`), store it as `highest` (`highest == 8`).
3. If `aTemp` is less than `lowest` so far (`8 < -3`), store it as `lowest` (`lowest` stays `-3`).

This seems okay. Now try the next three inputs to verify that `highest` and `lowest` are correct. Finally, the last step in the algorithm (after the repetition) produces the range: `range = highest - lowest + 1`.

Self-Check

- 6-15** Using the previous algorithm, determine the range when `n` is 4 and the user inputs the four temperature readings **3**, **4**, **2**, and **6**.

If you did the previous self-check question correctly, you noticed that our algorithm is not totally correct yet: `lowest` stays 0. The initial value of `lowest` is less than all subsequent inputs. The first test set before the self-check only worked because a negative temperature was input (something lower than the initial value for `lowest`, which is 0). But this algorithm does not work on a warmer day, when all temperatures are positive.

To solve this problem, instead of initializing both `highest` and `lowest` to 0, consider setting `highest` to something ridiculously low, like `-999`. This is so low that any input will have to be higher. Similarly, set `lowest` to something ridiculously high, like `999`, so that any input will have to be lower. Remember, though, that these mysterious values have no real meaning. Also, if you wanted a generalized loop that would find the range of any set of numbers, the algorithm would not always work.

A better alternative is to read in the first input, and then set both `highest` and `lowest` to that value. This is no longer phony. The first input is the highest and the lowest read in so far (assuming that the user is seeking the range of more than zero

inputs). The second input can then be compared to the first. Now, walk through the modified algorithm with $n == 4$ and inputs of **3**, **4**, **2**, and **6** to verify that the algorithm works.

Algorithm: Finding the range of n integers

```

input aTemp from the user
set highest so far to aTemp
set lowest so far to aTemp
for each of the remaining (2 through  $n$ ) temperature readings
{
    input aTemp from the user
    if aTemp is greater than highest so far,
        store it as highest
    if aTemp is less than lowest so far,
        store it as lowest
}
range = highest - lowest + 1

```

From Algorithm to Implementation

The problem stated that the number of inputs (n) would be known in advance. If n is 5, the user is prompted for five integers. This is an instance of the Determinate Loop pattern.

Code Sample: Class DemonstrateDeterminateLoop

```

1 // Read n integers and display the range of inputs
2 using System;
3 public class DemonstrateDeterminateLoop
4 {
5     static void Main()
6     {
7         int aTemp;
8         int n = 5; // Only works with 5
9
10        // Input first integer and record it as highest and lowest
11        Console.WriteLine("Enter {0} numbers ", n);
12        aTemp = int.Parse(Console.ReadLine());
13        int highest = aTemp;
14        int lowest = aTemp;
15

```

```

16 // Process n inputs. Since the first was already
17 // processed, start the loop counter at 2
18 for (int j = 2; j <= n; j++)
19 {
20 // Get the next input
21 aTemp = int.Parse(Console.ReadLine());
22
23 // Update the highest so far, if necessary
24 if (aTemp > highest)
25     highest = aTemp;
26
27 // Update the lowest so far, if necessary
28 if (aTemp < lowest)
29     lowest = aTemp;
30 }
31
32 int range = highest - lowest + 1;
33 Console.WriteLine("Range: {0}", range);
34 }
35 }

```

Dialogue

```

Enter 5 numbers
67
78
89
101
95
Range: 35

```

Testing the Implementation

To test this code, you should compare many hand-checked results with the range displayed by the program. For example, one set of inputs could include values that are all the same—the range should be 1. With two temperatures to check, the range should be computed as one more than the difference between those two values. Another test is the entry of one temperature only. This should result in the range of 1. The testing should also include a set of values where the number of inputs is greater than 2. This leads to many possible test sets, especially when you are attempting to input all possible orderings. Three inputs have six orderings, four inputs have 24 orderings, and in general n inputs have $n!$ (n factorial) orderings, or $(n * (n - 1) * (n - 2) * \dots * 3 * 2 * 1)$. Such exhaustive testing is impractical. It is also unnecessary.

A tester begins to gain confidence in the code by picking an arbitrary number of tests—for example, when n is 5 and the inputs were -5, 8, 22, -7, and 15. In this set of inputs, the difference between the highest and lowest is $(22 - (-7) + 1)$, or 30. Looking at the dialogue and seeing that the range is 30 could lead us to believe that the algorithm and implementation are correct. However, the only thing that is sure is this: when those particular five temperatures are entered, the correct range is returned.

However, always keep in mind that testing only reveals the presence of errors, not the absence of errors. If the range were shown as an obviously incorrect answer (-11, for example), hopefully you would detect the presence of an error.

Debugging Loops with Output Statements

When you detect an intent error, and a loop is involved, consider displaying the important values that might be incorrectly calculated. The program developed in the preceding sections tries to find the lowest and highest value from a group of n numbers. However, it would be easy to write it with an intent error, so it would be useful to print relevant variables' values such as `highest` and `lowest`. Writing an output statement inside the loop lets you see the changing values during each iteration of the loop. This simple debugging tool is called a *debugging output statement*. By showing you what is happening, it can make your task much easier.

A well-placed debugging output statement can be very revealing. For example, you could include one at the bottom of the loop below to show what is going on during the loop. The dialogue would look like this (some sections of the program are omitted with . . .):

```
...
for (int j = 1; j <= n; j++)
{
    ...

    else if (currentInput < lowest)
        lowest = aTemp;

    // DEBUGGING: Add a WriteLine to help aid debugging
    Console.WriteLine("Lowest: {0} Highest: {1}", highest, lowest);
}
```

Dialog

```
Enter 3 integers
5
Lowest: 5 Highest: 999
12
Lowest: 12 Highest: 999
```

16

Lowest: 16 Highest: 999
 Range: -983

The debugging output statement vividly shows that the highest value is not changing, even though 5, 12, and 16 are all less than 999. This suggests that the program's intent error is in the code that updates the variable `highest`.

Self-Check

Use this code to answer the questions that follow.

```
highest = -999;
lowest = 999;
int n = 5;
for (int j = 1; j <= n; j++)
{
    currentInput = int.Parse(Console.ReadLine());
    if (currentInput > highest)
        highest = currentInput;
    else if (currentInput < lowest)
        lowest = currentInput;
}
range = highest - lowest + 1;
```

6-16 Trace through the code above using these inputs:

-5 8 22 -7 15

Predict the value stored in `range`. Is it correct?

6-17 Trace through the same code with these inputs:

5 4 3 2 1

Predict the value stored in `range`. Is it correct?

6-18 Trace through the same code with these inputs:

1 2 3 4 5

Predict the value stored in `range`. Is it correct?

- 6-19 When is range incorrectly computed?
- (a) When the input is entered in descending order.
 - (b) When the input is entered in ascending order.
 - (c) When the input is in neither ascending nor descending order.
- 6-20 What must be done to correct the error?

The Fencepost Problem

Consider the problem of writing a method `PrintLetters` that prints the characters in a string, with a comma between each character. On the surface, it sounds simple to solve: write a loop that prints each letter followed by a comma. An initial implementation might look something like this:

```
// an incorrect implementation
public static void PrintLetters(string arg)
{
    for (int index = 0; index < arg.Length; index++)
    {
        Console.Write("{0}, ", arg[index]);
    }
    Console.WriteLine(); // end the line once
}
```

But as shown here, the output is not quite correct:

```
PrintLetters("Mississippi");
```

Output

```
M, i, s, s, i, s, s, i, p, p, i,
```

Notice the comma following the final `i` in `Mississippi`. It looks like the method is implemented incorrectly. The first thought many programmers have at this point is that the comma in `Console.Write("{0}, ", arg[index]);` should not have been placed after the letter, but before it. This suggests a second attempted implementation:

```
// sadly, also incorrect
public static void PrintLetters(string arg)
{
    for (int index = 0; index < arg.Length; index++)
```

```

{
    Console.Write(", {0}", arg[index]);
}
Console.WriteLine(); // end the line once
}

```

But this one has a new problem:

```
PrintLetters("Mississippi");
```

Output

```
, M, i, s, s, i, s, s, i, p, p, i
```

Now there is a comma before the first `M` in "Mississippi"! Neither of the above solutions is quite right.

This simple example demonstrates a common problem that arises in computer science, called the *fencepost problem*. The fencepost problem takes its name from the idea that every picket fence needs exactly one more post than the number of connecting areas between posts. The diagram below shows a fence with five posts and four connecting areas:

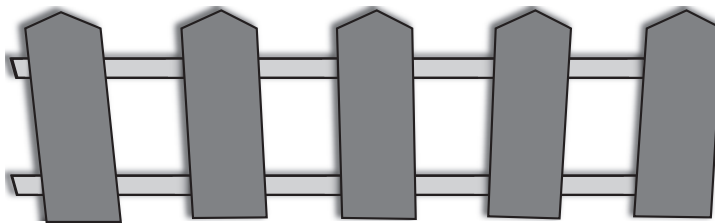


Figure 6.3: Fenceposts

The analogy of the fence can be applied to problems like the `PrintLetters` method just shown. The method needs to print every letter, with a comma in between, just as a picket fence needs every post to have a connecting area between it and the next. The letters of the string `arg` are the fence posts, and the commas are the connecting areas. But there should be no comma before the first letter or after the last letter, just as there should be no connecting area before the first fence post or after the last.

The two algorithms shown previously for `PrintLetters` were incorrect because they were essentially equivalent to the following algorithm for building a fence:

How to build a fence, with N posts. (attempt 1)

1. Repeat the following actions N times:

- Place a post.
- Place a connecting area.

If this algorithm were followed to build a fence with 4 posts, it would incorrectly look like the following figure.

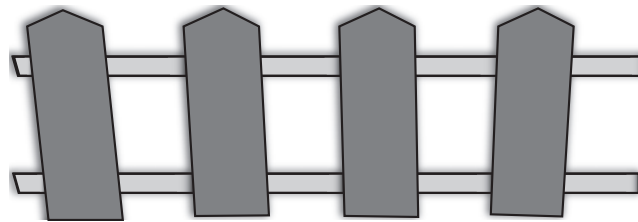


Figure 6.4: Fence with wrong number of posts

What was wrong with the algorithm? It needed to not place a connecting area after the last post. A second algorithm for building a fence might look like this:

How to build a fence, with N posts. (attempt 2)

1. Repeat the following actions N times:

- Place a post.
- If I am not on post # N (the last post), place a connecting area.

This algorithm works correctly, despite being a bit verbose. The code for `PrintLetters` would look something like this, if it used the preceding algorithm:

```
public static void PrintLetters(string arg)
{
    for (int index = 0; index < arg.Length; index++)
    {
        Console.Write("{0}", arg[index]);
        if (index < arg.Length - 1)
            Console.Write(", ");
    }
    Console.WriteLine(); // end the line once
}
```

The second fencepost algorithm is correct, but it can be improved and made simpler. A third way to fix the fencepost problem is with the following algorithm:

How to build a fence, with N posts. (attempt 3)

1. *Repeat the following actions $N - 1$ times:*
 - *Place a post.*
 - *Place a connecting area.*
2. *Place one final post to complete the fence.*

This third algorithm is somewhat better, because it does not require the fence builder to check how many posts have been placed every time. However, the algorithm has a minor incorrectness: if the builder were asked to place a fence with 0 posts, she would skip Step 1 entirely and then execute Step 2, incorrectly placing one post. A minor modification is needed to ensure the algorithm's correctness:

How to build a fence, with N posts. (attempt 4)

1. *Repeat the following actions $N - 1$ times:*
 - *Place a post.*
 - *Place a connecting area.*
2. *If there is a dangling last connecting area that needs a last fence post, that is, If N is greater than 0, then place one final post to complete the fence.*

This last algorithm is the most common solution to the fencepost problem, and is commonly called the “loop-and-a-half” solution. It is named this way because the code is usually implemented as a loop that does most of the work, and then a last step that does the final “half” iteration, which is slightly different from the others (following the analogy, it is the iteration that places only a fence post, and not a connecting area).

Implementing the loop-and-a-half solution to the fencepost problem gives a `PrintLetters` method like the following:

Code Sample: Class Fencepost

```

1 using System;
2
3 class Fencepost
4 {
5     public static void PrintLetters(string arg)
6     {
7         // write all letters of the string except the last one

```

Chapter 6 Repetition

```
8 // (place all fence posts and connecting areas but last one)
9 for (int index = 0; index < arg.Length - 1; index++)
10 {
11     Console.Write("{0}, ", arg[index]);
12 }
13
14 // last "half" loop iteration; write last letter
15 if (arg.Length > 0)
16     Console.Write(arg[arg.Length - 1]);
17
18 Console.WriteLine(); // end the line once
19 }
20
21 static void Main()
22 {
23     PrintLetters("Mississippi");
24 }
25 }
```

Output

```
M, i, s, s, i, s, s, i, p, p, i
```

Sentinel Loops: Indeterminate Loops and Fencepost Problem

The fencepost problem is often seen when solving problems involving sentinel loops. A *sentinel* is a specific input value used to terminate a loop. The loop that uses the sentinel value is called a *sentinel loop*. For example, perhaps a program repeatedly reads numbers from the user until the user enters -1 to stop, then averages all of the numbers the user entered. Sentinel loops like this one are indeterminate, because it is unknown how many numbers the user will enter before entering the sentinel value.

Dialogue

```
Enter test scores or -1 to quit:
80
90
70
-1
Average of 3 tests = 80.0
```

A sentinel value is of the same type of data as the other input; in this case, an integer `-1` just like the other integers the user enters. However, the sentinel itself is not an input number that the program is supposed to process. The user does not actually want `-1` to be included when calculating the average of the numbers. Because of this, the sentinel value must not be treated the same as other input. If `-1` were included with the scores just shown, the average would be considerably lower than `80.0`, and the student would be graded unfairly.

An incorrect algorithm for solving such a problem essentially looks like this (the variable `SENTINEL` is written in all uppercase because it will be implemented as a class constant):

```
number = something other than the SENTINEL;
while (number != SENTINEL)
{
    // Read the next number...

    // Process input...
}
```

The algorithm just shown has a problem. The last time through the loop, when the user enters the sentinel value, it will be processed just like the rest of the numbers. Since this is incorrect, the algorithm could be modified as follows to produce a correct, yet clunky, solution:

```
number = something other than the SENTINEL;
while (number != SENTINEL)
{
    // Read the next number...

    if (number != SENTINEL)
    {
        // Process input...
    }
}
```

This second algorithm is technically correct because it will not process the sentinel, and the loop will then terminate at the start of the next iteration. However, it is awkward because the loop test of `number != SENTINEL` must be repeated redundantly during each iteration through the loop. In addition, initializing the variable `number` to something other than the sentinel is also bad style, because if the sentinel ever changes, `number` might accidentally be initialized to the sentinel's value, which would break the program's correctness. This is poor coding style.

A better implementation of the algorithm uses the loop-and-a-half solution.

Chapter 6 Repetition

```
number = // Read the first number...

while (number != SENTINEL)
{
    // Process input...

    // Read the next number...
}
```

In this last implementation, the loop has been turned on its head. The “half” loop iteration is done first, by prompting the user and reading the first number. Then the loop begins; if the first number entered by the user is the sentinel, the loop is skipped. Otherwise, the number is processed, another number is read, and the loop starts over. If the newly read number is the sentinel, the loop test fails, and the loop terminates.

This last implementation solves the problems of the one before it; it does not repeat the loop test twice, and it does not initialize the `number` variable to an otherwise arbitrary value. The following program demonstrates the sentinel loop. This program asks the user either to enter data in the range of 0 through 100 inclusive, or to enter `-1` to signal the end of the data. With sentinel loops, you should always display a message that tells the user how to notify the program to terminate the loop when there is no more input. In the dialogue above, `-1` is the sentinel. This could easily have some been other value.

For this algorithm, the value for `number` must be read once before the loop test is made. This is called a “priming read,” which goes before the first iteration of the loop. Immediately after the `ReadLine` message at the bottom of the loop, `number` is always compared to `SENTINEL`. When the sentinel value `-1` is entered, the loop terminates, rather than attempting to process the number `-1` or attempting to add one too many to the counter `n`. The odd part of the algorithm is that the loop processes the valid data from the *previous* iteration of the loop (or, in the case of the first iteration of the loop, it is processing the data that was read just before the start of the loop).

Code Sample: DemonstrateIndeterminateLoop Class

```
1 using System;
2
3 // Find average by using SENTINEL of -1 to terminate loop
4 // that counts number of inputs and accumulates them.
5 class DemonstrateIndeterminateLoop
6 {
7     public static readonly int SENTINEL = -1;
8
9     static void Main()
10    {
```

```

11 Console.WriteLine("This program computes an average on");
12 Console.WriteLine("numbers entered before {0}", SENTINEL);
13 Console.WriteLine();
14
15 int number;
16 double accumulator = 0.0;
17 int n = 0;
18
19 Console.Write("Enter number or {0} to quit: ", SENTINEL);
20 number = int.Parse(Console.ReadLine());
21 while (number != SENTINEL)
22 {
23     // Process input
24     accumulator += number;
25
26     // Read the next string to be processed as a number or
27     // the sentinel value that terminates the loop
28     Console.Write("Enter number or {0} to quit: ",
29                 SENTINEL);
30     number = int.Parse(Console.ReadLine());
31
32     n++;
33 }
34
35 if (n == 0)
36     Console.WriteLine("Can't average zero numbers");
37 else
38     Console.WriteLine("Average {0:F1}", accumulator / n);
39 }
40 }

```

Dialogue

This program computes an average on
numbers entered before -1

```

Enter number or -1 to quit: 76
Enter number or -1 to quit: 92
Enter number or -1 to quit: 93
Enter number or -1 to quit: 80
Enter number or -1 to quit: -1
Average 85.3

```

The following table traces the changing state of the important variables, to simulate execution of the previous program. The variable named `accumulator` maintains the running sum of the test scores. The loop also increments `n` by `+1` for each valid `number` entered by the user. Notice that the number `-1` is not processed.

Location in Code	number	accumulator	n	loop test
First loop test	76	0.0	0	true
End of Iteration 1	92	76.0	1	true
End of Iteration 2	93	168.0	2	true
End of Iteration 3	80	261.0	3	true
End of Iteration 4	-1	341.0	4	true
Final Loop test	-1	341.0	4	false

At this point, `accumulator` and `n` are the correct values, and the loop terminates when the user wishes. A check is made to ensure that there was at least one valid number entered before computing the average.

Self-Check

- 6-21 Determine the value assigned to `average` for each of the following code fragments by simulating execution when the user inputs `70.0`, `60.0`, `80.0`, and `-1.0`. For both code segments, assume the following:

```
double currentInput;
int n = 0;
double accumulator = 0.0;
double SENTINEL = -1.0;
```

(a)

```
currentInput = double.Parse(Console.ReadLine());
while (currentInput != SENTINEL)
{
    currentInput = double.Parse(Console.ReadLine());
    accumulator += currentInput; // Update accumulator
    n++;                       // Update total # of inputs
}
double average = accumulator / n;
```

(b)

```
currentInput = double.Parse(Console.ReadLine());
while (currentInput != SENTINEL)
{
```

```

    accumulator += currentInput; // Update accumulator
    n++; // Update # of inputs
    currentInput = double.Parse(Console.ReadLine());
}
double average = accumulator / n;

```

- 6-22 Explain why the values are different for parts (a) and (b) of the last question. If you answered 70.0 for both (a) and (b) above, redo both until you get different answers for (a) and (b).
- 6-23 How many iterations of the loop just shown will occur when the user enters the sentinel (-1.0) as the very first input?
- 6-24 What activity should be added to the previous while statement, so that each loop iteration brings the loop one step closer to termination?

6.6 More Loops: `do/while` and `foreach`

The `for` loop is useful for processing determinate loops. Other C# looping statements exist for specialized types of loops. These additional loop statements are equivalent in power to the existing loops presented, but they are sometimes cleaner or more concise when implementing certain types of algorithms.

The `do/while` Loop Statement

The *`do/while` loop statement* is similar to the `while` loop. It allows a collection of statements to be repeated while an expression is `true`. The primary difference is the time at which the loop test is evaluated. Whereas the `while` loop test is evaluated at the beginning of each iteration, the `do/while` statement evaluates the loop test at the end of the loop. This means that the `do/while` loop always executes its repeated part at least once.

Here is the general form of the `do/while` loop.

General Form: `do/while` loop statement

```

do
{
    statement(s) to execute;
}
while (Boolean expression);

```

Notes:

The braces are *not* optional, even if only one statement is used!

Example:

```
string option = "?";
do
{
    Console.WriteLine("Enter <yes> or <no>: ");
    option = Console.ReadLine().Trim();
}
while (option != "yes" && option != "no" );
```

Dialog

```
Enter <yes> or <no>: Huh?
Enter <yes> or <no>: y
Enter <yes> or <no>: YES
Enter <yes> or <no>: yes
```

Note the semicolon after the `while` line of the `do/while` loop! It is easy to forget, but it is required for the syntax to be correct.

When a `do/while` statement is encountered, all statements execute within the block, from `{` to `}`. The *Boolean-test* evaluates at the *end* of the loop—not at the beginning. If the test expression is `true`, the *statement(s)-to-execute* execute again. If the test expression is `false`, the loop terminates.

Here is an example of the `do/while` loop. To help visualize the execution, this loop displays the increasing value of `counter`.

```
int counter = 1;
int n = 4;
Console.WriteLine("Before loop...");

do
{
    Console.WriteLine("Loop #{0}", counter);
    counter++;
}
while (counter <= n);

Console.WriteLine("...After loop");
```

Output

```

Before loop...
Loop #1
Loop #2
Loop #3
Loop #4
...After loop

```

You can write a nearly equivalent `while` loop, including having a pre-test, as follows:

```

int counter = 1;
int n = 4;
Console.WriteLine("Before loop...");
while(counter <= n)
{
    Console.WriteLine("Loop #{0}", counter);
    counter++;
}
Console.WriteLine("...After loop");

```

However, there are times when a post-test loop is better. Although the `while` loop shown above produces the same exact output as its `do/while` counterpart when `n` is greater than or equal to `counter`, consider what happens when `n = 1` is replaced with `n = 0`. Now, the `while` loop does not execute its iterative part; but the `do/while` loop executes its iterative part once, even when `n` is 0:

```

Before loop...
Loop #1
...After loop

```

The `do/while` loop is a good choice for repetition whenever a set of statements must be executed at least once to initialize objects that are used later in the loop test. The `do/while` loop is the preferred statement when asking the user of the program to enter one of several options. For example, in the following example, the `do/while` loop in the method `GetNextOption` repeatedly asks the user to enter one of three choices. The loop does not terminate until the user enters a valid option. The `Main` method also uses a `do/while` loop to process as many deposits and withdrawals as the user wants.

Code Sample: DoWhileLoopDemo Class

```
1 using System;
2
3 class DoWhileLoopDemo
4 {
5     public static readonly double AMOUNT = 10.00;
6
7     static void Main()
8     {
9         BankAccount account = new BankAccount("Smith", 100.00);
10        string choice = "";
11
12        do
13        {
14            choice = GetNextOption(); // Call the method in the loop
15
16            switch (choice)
17            {
18                case "W":
19                    account.Withdraw(AMOUNT);
20                    Console.WriteLine("Withdrew {0}", AMOUNT);
21                    break;
22                case "D":
23                    account.Deposit(AMOUNT);
24                    Console.WriteLine("Deposited {0}", AMOUNT);
25                    break;
26                case "P":
27                    Console.WriteLine("Account: {0}", account);
28                    break;
29                default:
30                    Console.WriteLine("Have a nice day :)");
31                    break;
32            }
33        } while (choice != "Q");
34    }
35
36    // Prompts until user chooses an uppercase W, D, or Q
37    public static string GetNextOption()
38    {
39        string option = "";
40
41        // loop until one of the valid inputs is read
42        do
43        {
44            Console.Write("Withdraw, Deposit, Print or Quit: ");
```

```

45
46     // get the first no blank character in upper case
47     option = Console.ReadLine().ToUpper();
48
49     // end loop if option is not any of the three valid choices
50     }
51     while (option != "W" && option != "D"
52           && option != "P" && option != "Q");
53
54     return option;
55     }
56 }

```

Dialogue

```

W)ithdraw, D)eposit, P)rint or Q)uit: invalid
W)ithdraw, D)eposit, P)rint or Q)uit: x
W)ithdraw, D)eposit, P)rint or Q)uit: p
Account: Smith $100.00
W)ithdraw, D)eposit, P)rint or Q)uit: d
Deposited 10
W)ithdraw, D)eposit, P)rint or Q)uit: d
Deposited 10
W)ithdraw, D)eposit, P)rint or Q)uit: p
Account: Smith $120.00
W)ithdraw, D)eposit, P)rint or Q)uit: w
Withdrew 10
W)ithdraw, D)eposit, P)rint or Q)uit: p
Account: Smith $110.00
W)ithdraw, D)eposit, P)rint or Q)uit: q
Have a nice day :)

```

Because at least one character must be obtained from the keyboard before the test expression evaluates, a `do/while` loop is used in `GetNextOption` instead of a `while` loop—the loop must iterate at least once. In addition, a `do/while` loop is used in `Main` to get an option, because it needs at least one user input to evaluate what the user wants to do.

Self-Check

6-25 Write the output produced by the following code:

(a)

```
int counter = 1;
do
{
    Console.Write("{0} ", counter);
    counter++;
}
while (counter <= 3);
```

(b)

```
string str = "abcdef";
do
{
    Console.WriteLine(str);
    str = str.Substring(1, str.Length);
}
while (str.Length > 0);
```

6-26 Rewrite the following `while` loop to use an equivalent `do/while` loop:

```
// generate a random number that is divisible by 6
Random rand = new Random();
int randomNumber = rand.Next();
while (randomNumber % 6 != 0)
{
    randomNumber = rand.Next();
    Console.WriteLine(randomNumber);
}
```

6-27 Write a `do/while` loop that prompts for and inputs integers until the integer input is in the range of 1 through 10, inclusive.

The `foreach` Loop Statement

The *`foreach` loop statement* is a specialized instance of the `for` loop, designed as a determinate loop that iterates over each element of a collection. The major collection that will be seen in this book is the array. However, arrays are not introduced until the next chapter. For now, the collection will be a `string` object, which is a collection of many `char` characters that represent the letters in the string.

Consider the following `for` loop:

```
string str = "Howdy";
for (int index = 0; index < str.Length; index++)
{
    Console.WriteLine(str[index]);
}
```

Semantically, the real meaning of the above loop is to print each letter of the string on its own line. The `for` loop's somewhat ugly syntax makes it somewhat hard to decipher what the code is doing. The solution is to use a `foreach` loop, whose syntax in this case is closer to the meaning of the code.

General Form: `foreach` loop statement

```
foreach (variable-type variable-name in collection-name)
{
    statement(s) to execute;
}
```

Notes:

The braces are optional if only one statement is used.

Example:

```
string message = "Hi there!";
foreach (char ch in message)
{
    Console.WriteLine("Letter is: {0}", ch);
}
```

Note that the `foreach` loop does not need any update step or Boolean expression test. The update step is implicitly to move the counter variable forward to the next element of the collection—in this case, to the next character of the string. The Boolean expression test is implicitly to stop when the loop has exhausted all elements in the collection—all characters in the string, in this case.

The previous `for` loop example could be rewritten with a `foreach` loop like this:

```
string str = "Howdy";
foreach (char letter in str)
{
    Console.WriteLine(letter);
}
```

Not all `for` loops can be replaced by `foreach` loops! This is possible only with `for` loops that operate on elements in a collection, which in this case are characters of a `string`. Its usefulness is limited now, but in later chapters, when you see more collections, the `foreach` loop will prove to be a handy statement.

Self-Check

6-28 Write the output produced by the following code:

(a)

```
string message1 = "Hello";
string message2 = "";
foreach (char ch in message1)
{
    message2 = ch + message2;
}
Console.WriteLine(message2);
```

(b)

```
string str = "abcdef";
string str2 = "";
foreach (char letter in str)
{
    str2 += letter;
    str2 += letter;
}
Console.WriteLine(str2);
```

6-29 Rewrite the following `for` loop to use an equivalent `foreach` loop:

```
string myName = "amanda banana ranna";
for (int index = 0; index < myName.Length; index++)
{
    // only do this code on letters
    if (myName[index] >= 'a' && myName[index] <= 'z')
    {
        int letterNum = myName[index] - 'a' + 1;
        Console.WriteLine("{0} is letter {1} in the alphabet",
            myName[index], letterNum);
    }
}
```

- 6-30 Which kind of loop best accomplishes these tasks?
- (a) Sum the first five integers ($1 + 2 + 3 + 4 + 5$).
 - (b) Find the average value for a set of numbers when the size of the set is known.
 - (c) Find the average for a set of numbers when the size of the set cannot be determined until the data has been completely entered.
 - (d) From the user, obtain a character that must be an uppercase S or Q.

Chapter Summary

- Repetition is an important method of control for all programming languages. Typically, the body of a loop has statements that may change the state of one or more variables during each loop iteration, so that the loop can terminate.
- The `for` loop is often used to implement the Determinate Loop pattern, which requires that the number of repetitions be known before the loop is encountered.
- Determinate loops rely on a value representing the number of repetitions (*n* perhaps) and a properly initialized and incremented loop counter (*j* perhaps) to track the number of repetitions. The loop counter is compared to the known number of iterations at the start of each loop. The counter is automatically updated at the end of each `for` loop iteration.
- There are a number of ways to determine the number of loop iterations before a loop executes. The number of iterations may be input from the user, passed as an argument to a method, initialized in advance, or may be part of the state of some object. For example, every `string` object knows how many characters it has at any given moment.
- The Determinate Loop pattern is so common that a specific statement—the `for` loop—is built into almost all languages.
- Indeterminate loops rely on some external event for their termination. The terminating event may occur at any time.
- Indeterminate loops are used when a program is unable to determine, in advance, the number of times a loop must iterate. The terminating events include sentinels read from the keyboard (such as `-1` as a test or `"Q"` in a menu selection). These types of loops allow any number of users to execute any number of transactions, for example.

- Although you often need one repetitive statement to solve any computer problem, the `for` loop is the most convenient one under certain circumstances. The `for` loop requires the program to take care of the initialization, loop test, and repeated statement all at once. The compiler protests if one of these important steps is missing. The `for` loop provides a more compact and less error-prone determinate loop than some other looping structures.
- Remember these steps if you are having trouble designing loops:
 - Determine which type of loop to use.
 - Determine the loop test.
 - Write the statements to be repeated.
 - Bring the loop one step closer to termination.
 - Initialize variables if necessary.
 - Use debugging output statements to discover and fix intent errors. (Make sure to remove or comment them out later!)

Key Terms

brute force

debugging output statement

Determinate Loop pattern

do/while loop statement

empty loop

fencepost problem

`for` loop statement`foreach` loop statement

Indeterminate Loop pattern

infinite loop

iteration

loop

repetition

sentinel

sentinel loop

while loop statement

Exercises

1. How many times will the following loops execute `Console.WriteLine("Hello");`? For this question, “Zero” and “Infinite” are legitimate possible answers.

(a)

```
int n = 5;
for (int j = 1; j <= n; j++)
{
    Console.WriteLine("Hello ");
}
```

(b)

```
int n = 0;
for (int j = 5; j >= n; j--)
{
    Console.WriteLine("Hello ");
}
```

(c)

```
int n = 5;
for (int j = 1; j <= n; j--)
{
    Console.WriteLine("Hello ");
    j++;
}
```

(d)

```
int n = 0;
for (int j = 1; j <= n; j++)
{
    Console.WriteLine("Hello ");
}
```

2. Write the output produced by these for loops:

```
for (int counter = 1; counter <= 5; counter++)
    Console.WriteLine(" {0}", counter);
Console.WriteLine(" Loop One");
```

```
for (int counter = 10; counter >= 1; counter--)
    Console.WriteLine(" {0}", counter);
Console.WriteLine(" Blast Off ");
```

3. Write loops to produce the outputs shown.

(a) 10 9 8 7 6 5 4 3 2 1 0

(b) 0 5 10 15 20 25 30 35 40 45 50

(c) -1000 -900 -800 -700 -600 -500 -400 -300 -200 -100 0

4. Write the output generated by the following code:

```
int j = 0;
while (j < 5)
{
    Console.WriteLine(" {0}", j);
    j++;
}
```

Chapter 6 Repetition

5. Write a `for` loop that sums all the integers between `start` and `stop` inclusive that are input from the keyboard. You may assume that `start` is always less than or equal to `stop`. If the input were 5 for `start` and 10 for `stop`, the sum would be $5 + 6 + 7 + 8 + 9 + 10$ (45).

6. How many times will `Hello` be displayed using the following program segments? For this question, “Zero,” “Unknown,” and “Infinite” are legitimate possible answers.

(a)

```
while (j <= 10)
    Console.WriteLine("Hello");
```

(b)

```
int j = 1;
while (j <= 7)
{
    Console.WriteLine("Hello");
    j++;
}
```

(c)

```
int j = 7;
while (j <= 1)
{
    Console.WriteLine("Hello");
}
```

(d)

```
int j = 1;
while (j <= 5)
    Console.WriteLine("Hello");
j++;
```

7. Write a `while` loop that produces this output:

```
-4 -3 -2 -1 0 1 2 3 4 5 6
```

8. Write a loop that displays 100, 95, . . . , 5, and 0 on separate lines.
9. Write a loop that counts how many perfect scores (scores of 100) are entered from the keyboard.
10. Convert the following code to its `for` loop counterpart:

```
Console.WriteLine("Enter number of ints to be summed: ");
int n = int.Parse(Console.ReadLine());
```

```

int counter = 1;
int sum = 0;

Console.WriteLine("Enter {0} integers on separate lines: ", n);
while (counter <= n)
{
    int anInt = int.Parse(Console.ReadLine());
    sum += anInt;
    counter++;
}
Console.WriteLine("Sum: {0}", sum);

```

11. Write a loop that counts the number of lines input by a user until the user enters the string `ENDOFDATA` (must be uppercase letters, no spaces) on a line by itself. The words will be entered one string per line.
12. Write the complete output generated by the following code when the user enters 1, 2, 3, 4, and -999 on separate lines.

```

double sum = 0.0;
Console.Write("Enter tests or a negative number to quit: ");
double test = double.Parse(Console.ReadLine());
while(test >= 0.0)
{
    sum += test;
    test = double.Parse(Console.ReadLine());
}
Console.Write("Sum: {0}", sum);

```

13. Write the output generated by the following code:

```

string choice = "BDWBQDW";
for (int j = 0; choice[j] != 'Q'; j++)
{
    Console.WriteLine("Opt: {0}", choice[j]);
}

```

14. How many times will the following loops print "Hello "? For this question, "Zero," "Unknown," and "Infinite" are legitimate possible answers.

```

(a)
int j = 1;
int n = 10;
do
{
    Console.Write("Hello ");
}

```

Chapter 6 Repetition

```
    }  
    while (j > n);  
  
    (b)  
    int n = 10;  
    int j = 1;  
    do  
    {  
        Console.WriteLine("Hello ");  
        j = j - 2;  
    }  
    while (j <= n);  
  
    (c)  
    int j = -1;  
    do  
    {  
        Console.WriteLine("Hello ");  
        j++;  
    }  
    while (j != j);  
  
    (d)  
    int j = 1;  
    do  
    {  
        Console.WriteLine("Hello ");  
        j++;  
    }  
    while (j <= 100);
```

15. Write a do/while loop that produces this output:

```
10 9 8 7 6 5 4 3 2 1 0
```

16. Write the output generated by the following program:

```
int j = -2;  
do  
{  
    Console.WriteLine("{0} ", j);  
    j--;  
}  
while (j > -6);
```

17. Rewrite the following `do/while` loop as a pretest `while` loop (the loop test is first, before the body):

```
int counter = 0;
do
{
    Console.WriteLine(counter);
    counter++;
}
while(counter <= 100);
```

18. Complete a method `EvenNumber` that prompts for, and returns, an even integer in the range of two integer arguments (assume that it is called from another method in the same class). If the user enters an odd number, or an integer that is out of the range, inform the user of the error until they enter an even number that is in the range.

```
using system;
public class EvenNumberLoop
{
    // Return even number in range of min through max inclusive
    public static int EvenNumber(int min, int max)
    {
        // Complete this code
    }

    static void Main()
    {
        int size = EvenNumber(1, 20);
        Console.WriteLine("You entered {0}", size);
    }
}
```

Dialogue

```
Enter an even integer in the range of 1 through 20: -1
Integer was not even
Integer was out of range
Enter an even integer in the range of 1 through 20: 0
Integer was out of range
Enter an even integer in the range of 1 through 20: 1
Integer was not even
Enter an even integer in the range of 1 through 20: 21
Integer was not even
Integer was out of range
Enter an even integer in the range of 1 through 20: 20
You entered 20
```

Programming Tips

1. Pick the type of loop you want to use.

After recognizing the need for repetition, decide if the number of repetitions can be determined in advance. If so, use a determinate loop, which is best implemented with a `for` loop. If the number of iterations cannot be determined in advance, determine the event that terminates the loop. For example, the loop might terminate when the user enters the word `STOP`. In this case, the termination condition is `word == "STOP"`. The loop test is the logical negation `word != "STOP"`.

```
while (word != "STOP")
{ // ...
}
```

2. Beware of infinite loops.

Watch out for infinite loops. They are easy to create, and sometimes they are very difficult to find. Can you spot why these are infinite loops?

```
while (j <= 100)
{ // Sum the first 100 integers
  sum += j;
}
j++;

for (j = 0; j <= 100; j++)
{ // Sum the first 100 integers
  sum += j;
  j--;
}

for (j = 1; j <= 10; j++);
  Console.WriteLine(j);
```

3. Always write a block for the iterative part of `while` loops.

This provides you a better chance of including any increment statement as part of the loop, rather than accidentally leaving it outside of the loop.

4. Use debugging outputs to find out what is going on in a loop.

Use debugging output statements inside a loop to display one or more variables that should be changing. This can be very revealing. Sometimes you'll spot an infinite loop. Other times you might spot that the loop test is never true.

```
while (. . .)
{ // ...
  mid = (lo + hi) / 2.0;
  Console.WriteLine("In loop, mid equals {0}", mid );
  // ...
}
```

5. Loops may not always execute the iterative part.

It is possible that a loop will execute zero times, or less than you might have thought.

```
int n = 3;
for (int j = 1; j >= n; j++) // 1 >= n is false
{
  Console.Write("You'll not see me!");
}

for (int j = 1; j <= n; j++); // Get rid of ;
{
  Console.Write("You see me only once, not thrice");
  Console.Write("for I'm not part of the for!");
}
```

Programming Projects

6A Wind Speed

Write a program (a class with the `Main` method only) that determines the lowest, highest, and average of a set of wind speed readings. Prompt the user for the number of wind speed readings in the set. After the set has been entered, compute and display the high, low, and average. Assume that the user always enters at least one wind speed reading. Round the average (after `Ave`;) to the nearest integer. (Use the `Math` class to do this.)

Chapter 6 Repetition

```
Enter number of wind speed readings: 4
4
6
8
3

High: 8
Low: 3
Count: 4
Ave: 5
```

6B Factorial

Write a program (a class with the `main` method only) that asks for a number and displays $N!$ (N factorial) where $N!$ is the product of all the positive integers from 1 to N . For example, $5! = 1 \times 2 \times 3 \times 4 \times 5 = 120$; $4! = 1 \times 2 \times 3 \times 4 = 24$; and $0! = 1$ (by definition). Test your program with these arguments: 0, 1, 2, and 7. Here is one log of execution when the user enters 4:

```
Enter n: 4
4! is 24
```

6C Squaring Integers

The square of an integer value n can be found by adding the first n positive odd integers. For example, both 4 squared and -4 squared are the sum of the first four positive odd integers ($1 + 3 + 5 + 7 = 16$). Write a program (a class with the `main` method only) that reads an integer and displays the square of that integer, using this algorithm. Do not use the built-in method `Math.Pow` or the multiplication operator `*`. Test your program with various inputs, including negative integers and zero. Here is one sample log of execution:

```
Enter integer: -4
-4 squared is 16
```

6D The Elevator Class

Complete the `Elevator` class with a constructor that places an elevator with an ID at a selected floor. The constructor requires exactly three arguments:

1. a string as the elevator ID
2. an int as the starting floor
3. an int as the top floor

The lowest floor that can be selected is 1. The `Select` method allows floors to be selected and issues an error message when the wrong floor is selected—stating that the selected floor is either too high or too low. For every floor, the elevator ID should be displayed before the message “going up” or “going down” and the current floor of the elevator.

The sample output shown below gives you an idea of what one simulated `Elevator` object must look like on your screen. Notice that a message is displayed when both elevators are constructed. The output also shows that there are several places where the constructor and the `select` method must check for and report errors.

```

1 // Test drive the Elevator class by constructing one
2 // Elevator object and sending several select(floor)
3 // messages, some of which are invalid.
4 class TestElevator
5 {
6     static void Main()
7     {
8         Elevator error1 = new Elevator("No way 1", 1, -1);
9         Elevator error2 = new Elevator("No way 2", 3, 2);
10        Elevator error3 = new Elevator("No way 3", 0, 9);
11        Elevator liftOne = new Elevator("West", 2, 12);
12        Elevator liftTwo = new Elevator("East", 1, 6);
13
14        liftOne.Select(5);
15        liftTwo.Select(3);
16        liftOne.Select(5); // 5 is current floor; open at 5 again
17        liftTwo.Select(7);
18        liftOne.Select(1);
19        liftTwo.Select(1);
20        liftOne.Select(13); // Out of range--elevator does not change
21        liftOne.Select(0); // Out of range--elevator does not change
22    }
23 }
```

Output

```

**ERROR** max floor for 'No Way 1' must be greater than 1
**ERROR** starting floor for 'No Way 2' must be in range of 1..2
**ERROR** starting floor for 'No Way 3' must be in range of 1..9
West begins at floor 2
East begins at floor 1
West going from 2 up to 5
    going up to: 3
    going up to: 4
    going up to: 5
West open at 5
East going from 1 up to 3
    going up to: 2
    going up to: 3
East open at 3
West door opens at 5 (current floor == selected floor)
**ERROR** 7 not in range of 1..6 for East
East open at 3
West going from 5 down to 1
    going down to: 4
    going down to: 3
    going down to: 2
    going down to: 1
West open at 1
East going from 3 down to 1
    going down to: 2
    going down to: 1
East open at 1
**ERROR** 13 not in range of 1..12 for West
West open at 1
**ERROR** 0 not in range of 1..12 for West
West open at 1

```

6E Mini-Teller

Write a program (a class with the `Main` method only) that allows a user to make as many withdrawals, deposits, and balance queries as desired to one `BankAccount` object that you construct. Your program should notify the user of insufficient funds if the withdrawal fails due to lack of funds. Use the following dialogue to help you understand the problem specification, which assumes that your single `BankAccount` was constructed like this:

```
BankAccount currentAccount = new BankAccount("Jackson", 100.00);
```

```
Dialogue
Initial balance for Jackson is 100.0

Withdraw, Deposit, Balance, or Quit: d
Enter deposit amount: 1.00

Withdraw, Deposit, Balance, or Quit: b
Current balance: 101.0

Withdraw, Deposit, Balance, or Quit: w
Enter withdrawal amount: 2.00

Withdraw, Deposit, Balance, or Quit: b
Current balance: 99.0

Withdraw, Deposit, Balance, or Quit: w
Enter withdrawal amount: 100.00
**Amount requested exceeds account balance**

Withdraw, Deposit, Balance, or Quit: b
Current balance: 99.0

Withdraw, Deposit, Balance, or Quit: q
Have a nice day :)
```

Also, notice that upper- and lower-case choices are to be allowed. You might wish to use one of the `string` methods in Chapter 3 to convert the user's input to uppercase, to avoid having to check for both options such as `q` and `Q`.

6F Wind Speed Again

Write a program (a class with the `Main` method only) that determines the lowest, highest, and average of a set of wind speed readings that are all positive or zero. Terminate the loop with any negative input. Be sure that you notify the user how to terminate data entry. Compute and display the high, the low, the number of inputs (initially undetermined), and the average (rounded to the nearest tenth). Here is one sample dialogue:

Chapter 6 Repetition

Enter wind speed readings or a number less than 0 to quit:

```
4
6
8
3
-5
```

```
High: 8
Low: 3
Count: 4
Ave: 5.3
```

6G Guessing Game

Write a C# class that implements a guessing game. Ask the user to guess a random number that you have generated, in the range of 1 through 100, inclusive. If the guess is larger than the number, tell the user that the guess was too high. If the guess is too low, tell that to the user. When the guess matches the random number, tell that to the user, state how many guesses the user took, and terminate the program. Here is one sample dialogue:

```
Play one game
Pick a number from 1..100: 50
50 is too high
Pick a number from 1..100: 25
25 is too high
Pick a number from 1..100: 12
12 is too low
Pick a number from 1..100: 18
18 is too high
Pick a number from 1..100: 15
15 is just right
Congrats, you needed 5 guesses
```

The name of the class is `GuessingGame`. The method named `PlayOneGame` must get one guessing game going. The `NumberOfGuesses` property must return the number of guesses the user has taken at any moment (starts at 0 and increases each time the user makes a guess).

```
GuessingGame aGuessingGame = new GuessingGame();

aGuessingGame.PlayOneGame();

Console.WriteLine("#Tries {0}", aGuessingGame.NumberOfGuesses);
```

This project requires the use of random numbers. Use the `Random` class covered in Chapter 3 for this.

6H Piggy Bank

As we have seen in this chapter, loops are useful when writing classes that need repetitive behavior. Consider a child's piggy bank in countries that have penny, nickel, dime, and quarter coins. Imagine that a child wants you to add a minimum number of money, in cents. You can add any of the four different types of coins to get to this minimum, and you can even go over the minimum requested.

The following dialogue shows that the input continues until the desired minimum is reached. There are many ways to get to the minimum shown here—this is just one:

```
Piggy bank has 0 cents, desired minimum 61
Add coins to reach a minimum of 61
```

```
Add P)enny N)ickel D)ime Q)uarter: Q
25 cents, desired minimum 61
```

```
Add P)enny N)ickel D)ime Q)uarter: q
50 cents, desired minimum 61
```

```
Add P)enny N)ickel D)ime Q)uarter: d
60 cents, desired minimum 61
```

```
Add P)enny N)ickel D)ime Q)uarter: p
61 cents, desired minimum 61
```

The `while` loop shown below keeps asking for money until the `PiggyBank` has at least the minimum coins required (`PiggyBank` members are in boldface).

Code Sample: TestPiggyBank Class

```
1 using System;
2
3 class TestPiggyBank
4 {
5     static void Main()
6     {
7         // PiggyBank with 0 cents and a desired minimum of 61 cents
8         PiggyBank bank = new PiggyBank(0, 61);
9
10        // Prompt the user to get things going
11        Console.WriteLine("Piggy bank has {0}", bank);
12        Console.WriteLine("Add coins to reach a minimum of {0}",
13                            Bank.Minimum);
14
15        // loop goes until the bank does not need more cents
16        while (bank.NeedsMore())
17        {
18            // Add a penny, nickel, dime, or quarter
19            bank.AddMoreMoney();
20
21            // Show updated state of the PiggyBank
22            Console.WriteLine(bank);
23            Console.WriteLine();
24        }
25    }
26 }
```

After a `PiggyBank` object is constructed, with an arbitrary number of cents (0) and a desired minimum amount (61), the `while` loop repeats until the `PiggyBank` has at least the minimum number of cents requested (it could get more). This loop may execute from 0 to many times. The exact number of loop iterations cannot be predetermined. The minimum may vary, and/or the user may enter different combinations of coins.

Write the `PiggyBank` class for this programming project so that it works with the above test code.