3.12 Recursion

The applications we have discussed thus far are generally structured as methods that call one another in a disciplined, hierarchical manner. For some problems, however, it's useful to have a method call itself. A recursive method is a method that calls itself, either directly or indirectly through another method.

We consider recursion conceptually first. Then we examine an application containing a recursive method. Recursive problem-solving approaches have a number of elements in common. When a recursive method is called to solve a problem, it actually is capable of solving only the simplest case(s), or base case(s). If the method is called with a base case, it returns a result. If the method is called with a more complex problem, it divides the problem into two conceptual pieces: a piece that the method knows how to do and a piece that it does not know how to do. To make recursion feasible, the latter piece must resemble the original problem, but be a slightly simpler or slightly smaller version of it. Because this new problem looks like the original problem, the method calls a fresh copy of itself to work on the smaller problem; this is referred to as a recursive call and is also called the recursion step. The recursion step normally includes a return statement, because its result will be combined with the portion of the problem the method knew how to solve to form a result that will be passed back to the original caller.

The recursion step executes while the original call to the method is still active (i.e., while it has not finished executing). The recursion step can result in many more recursive calls, as the method divides each new subproblem into two conceptual pieces. For the recursion to terminate eventually, each time the method calls itself with a slightly simpler version of the original problem, the sequence of smaller and smaller problems must con- verge on the base case. At that point, the method recognizes the base case and returns a result to the previous copy of the *method. A sequence of returns ensues until the original method call returns the result to the caller. This process sounds complex compared with the conventional problem solving we have performed to this point.*

Recursive Factorial Calculations

As an example of recursion concepts at work, let's write a recursive application to perform a popular mathematical calculation. Consider the factorial of a nonnegative integer n, written n! (and pronounced "n factorial"), which is the product

$$n \cdot (n-1) \cdot (n-2) \cdot \ldots \cdot 1$$

1! is equal to 1 and 0! is defined to be 1. For example, 5! is the product 5 · 4 · 3 · 2 · 1, which is equal to 120. The factorial of an integer, number, greater than or equal to 0 can be calculated iteratively (nonrecursively) using the for statement as follows:

```
factorial = 1;

for ( int counter = number; counter >= 1; counter-- )
   factorial *= counter;
```

A recursive declaration of the factorial method is arrived at by observing the following relationship:

$$n! = n \cdot (n-1)!$$

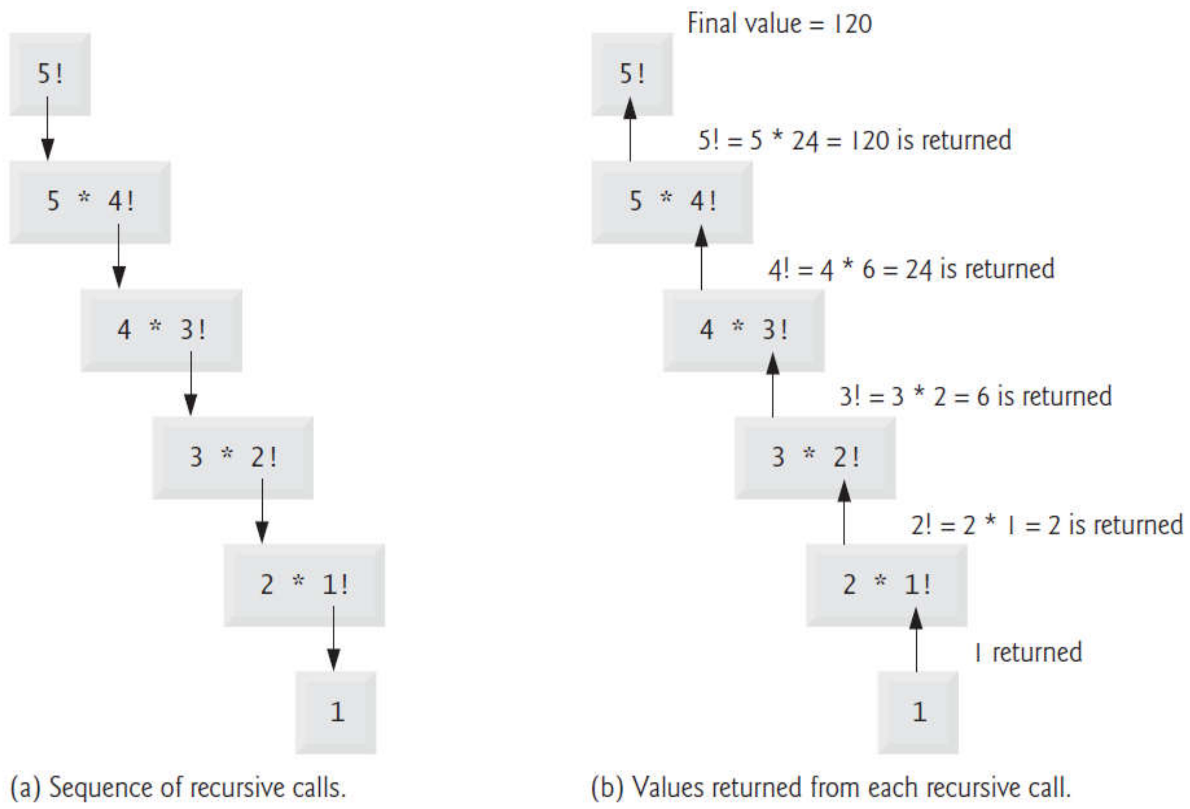
For example, 5! is clearly equal to 5 · 4!, as is shown by the following equations:

$$5! = 5 \cdot 4 \cdot 3 \cdot 2 \cdot 1$$
$$5! = 5 \cdot (4 \cdot 3 \cdot 2 \cdot 1)$$
$$5! = 5 \cdot (4!)$$

The evaluation of 5! would proceed as shown in Fig. 3.10. Figure 3.10(a) shows how the succession of recursive calls proceeds until 1! is evaluated to be 1, which terminates the recursion. Figure 3.10(b) shows the values returned from each recursive call to its caller until the value is calculated and returned. Figure 3.11 uses recursion to calculate and display the factorials of the integers from 0 to 10. The recursive method

Factorial (lines 16–24) first tests to determine whether a terminating condition (line 19) is true.

If number is less than or equal to 1 (the base case), Factorial returns 1, no further recursion is necessary and the method returns. If number is greater than 1, line 23 expresses the problem as the product of number and a recursive call to Factorial evaluating the factorial of number - 1, which is a slightly simpler problem than the original



5!
5 * 4!
4 * 3!
3 * 2!
2 * 1!
1
(a) Sequence of recursive calls.
Final value = 120
5!
5! = 5 * 24 = 120 is returned
5 * 4!
4! = 4 * 6 = 24 is returned
4 * 3!
3! = 3 * 2 = 6 is returned
3 * 2!
2! = 2 * 1 = 2 is returned
2 * 1!
1 returned
1
(b) Values returned from each recursive call.

calculation, Factorial(number).

Fig. 3.10 | Recursive evaluation of 5!.

Method Factorial (lines 16–24) receives a parameter of type long and returns a result of type long. As you can see in Fig. 3.11, factorial values become large quickly. We chose type long (which can represent relatively large integers) so that the application could calculate factorials greater than 20!. Unfortunately, the Factorial method produces large values so quickly that factorial values soon exceed even the maximum value that can *be stored in a long variable. Due to the restrictions on the integral types, variables of type float, double or decimal might ultimately be needed to calculate factorials of larger numbers. This situation points to a weakness in many programming languages—the languages are not easily extended to handle the unique requirements of various applications. As you know, C# allows you to create a type that supports arbitrarily large integers if you wish. For example, you could create a HugeInteger class that would enable an application to calculate the factorials of arbitrarily large numbers. You can also use the new type BigInteger from the .NET Framework's class library.*

```
// Fig. 3.11: FactorialTest.cs
// Recursive Factorial method.
using System;

public class FactorialTest
{
   public static void Main( string[] args )
   {
      // calculate the factorials of 0 through 10
      for ( long counter = 0; counter <= 10; counter++ )
         Console.WriteLine( "{0}! = {1}",
            counter, Factorial( counter ) );
   } // end Main

   // recursive declaration of method Factorial
   public static long Factorial( long number )
   {
      // base case
      if ( number <= 1 )
         return 1;
      // recursion step
      else
         return number * Factorial( number - 1 );
   } // end method Factorial
} // end class FactorialTest
```

```
0! = 1
1! = 1
2! = 2
3! = 6
4! = 24
5! = 120
6! = 720
7! = 5040
8! = 40320
9! = 362880
10! = 3628800
```

Fig. 3.11 | Recursive `Factorial` method.

3.13 Passing Arguments: Pass-by-Value vs. Pass-by- Reference

Two ways to pass arguments to functions in many programming languages are pass-by- value and pass-by-reference. When an argument is passed by value (the default in C#), a copy of its value is made and passed to the called function. Changes to the copy do not affect the original variable's value in the caller. This prevents the accidental side effects that so greatly hinder the development of correct and reliable software systems. Each argument that has been passed in the programs in this chapter so far has been passed by value. When an argument is passed by reference, the caller gives the method the ability to access and modify the caller's original variable.

Previously, we discussed the difference between value types and reference types. A major difference between them is that value-type variables store values, so specifying a value-type variable in a method call passes a copy of that variable's value to the method. Reference -type variables store references to objects, so specifying a reference-type variable as an argument passes the method a copy of the actual reference that refers to the object. Even though the reference itself is passed by value, the method can still use the reference it receives to interact with—and possibly modify—the original object. Similarly, when returning information from a method via a return statement, the method returns a copy of the value stored in a value-type variable or a copy of the reference stored in a reference- type variable.

*What if you would like to pass a variable by reference so the called method can modify the variable's value? To do this, C# provides keywords **ref** and **out**. Applying the **ref** keyword to a parameter declaration allows you to pass a variable to a method by reference— the called method will be able to modify the original variable in the caller. The ref key- word is used for variables that already have been initialized in the calling method. Normally, when a method call contains an uninitialized variable as an argument, the compiler generates an error. Preceding a parameter with keyword out creates an output parameter.*

This indicates to the compiler that the argument will be passed into the called method by reference and that the called method will assign a value to the original variable in the caller. If the method does not assign a value to the output parameter in every possible path of execution, the compiler generates an error. This also prevents the compiler from generating an error message for an uninitialized variable that's passed as an argument to a method. A method can return only one value to its caller via a return statement, but can return many values by specifying multiple output (ref and/or out) parameters.

The application in Fig. 3.12 uses the ref and out keywords to manipulate integer values. The class contains three methods that calculate the square of an integer. Method SquareRef (lines 37–40) multiplies its parameter x by itself and assigns the new value to x. SquareRef's parameter is declared as ref int, which indicates that the argument passed to this method must be an integer that's passed by reference. Because the argument is passed by reference, the assignment at line 39 modifies the original argument's value in the caller.

Method SquareOut (lines 44–48) assigns its parameter the value 6 (line 46), then squares that value. SquareOut's parameter is declared as out int, which indicates that the argument passed to this method must be an integer that's passed by reference and that the argument does not need to be initialized in advance.

```
 1  // Fig. 3.12: ReferenceAndOutputParameters.cs
 2  // Reference, output and value parameters.
 3  using System;
 4
 5  class ReferenceAndOutputParameters
 6  {
 7     // call methods with reference, output and value parameters
 8     public static void Main( string[] args )
 9     {
10        int y = 5; // initialize y to 5
11        int z; // declares z, but does not initialize it
12
13        // display original values of y and z
14        Console.WriteLine( "Original value of y: {0}", y );
15        Console.WriteLine( "Original value of z: uninitialized\n" );
16
```

```
      // pass y and z by reference
      SquareRef( ref y ); // must use keyword ref
      SquareOut( out z ); // must use keyword out

      // display values of y and z after they are modified by
      // methods SquareRef and SquareOut, respectively
      Console.WriteLine( "Value of y after SquareRef: {0}", y );
      Console.WriteLine( "Value of z after SquareOut: {0}\n", z );

      // pass y and z by value
      Square( y );
      Square( z );

    // display values of y and z after they are passed to method Square
      // to demonstrate that arguments passed by value are not modified
      Console.WriteLine( "Value of y after Square: {0}", y );
      Console.WriteLine( "Value of z after Square: {0}", z );
   } // end Main

   // uses reference parameter x to modify caller's variable
   static void SquareRef(ref int x )
   {
      x = x * x; // squares value of caller's variable
   } // end method SquareRef

   // uses output parameter x to assign a value
   // to an uninitialized variable
   static void SquareOut(    out int x )
   {
      x = 6; // assigns a value to caller's variable
      x = x * x; // squares value of caller's variable
   } // end method SquareOut

   // parameter x receives a copy of the value passed as an argument,
   // so this method cannot modify the caller's variable
   static void Square(int x )
   {
      x = x * x;
   } // end method Square
} // end class ReferenceAndOutputParameters
```

```
Original value of y: 5
Original value of z: uninitialized

Value of y after SquareRef: 25
Value of z after SquareOut: 36

Value of y after Square: 25
Value of z after Square: 36
```

Fig. 3.12 | Reference, output and value parameters.