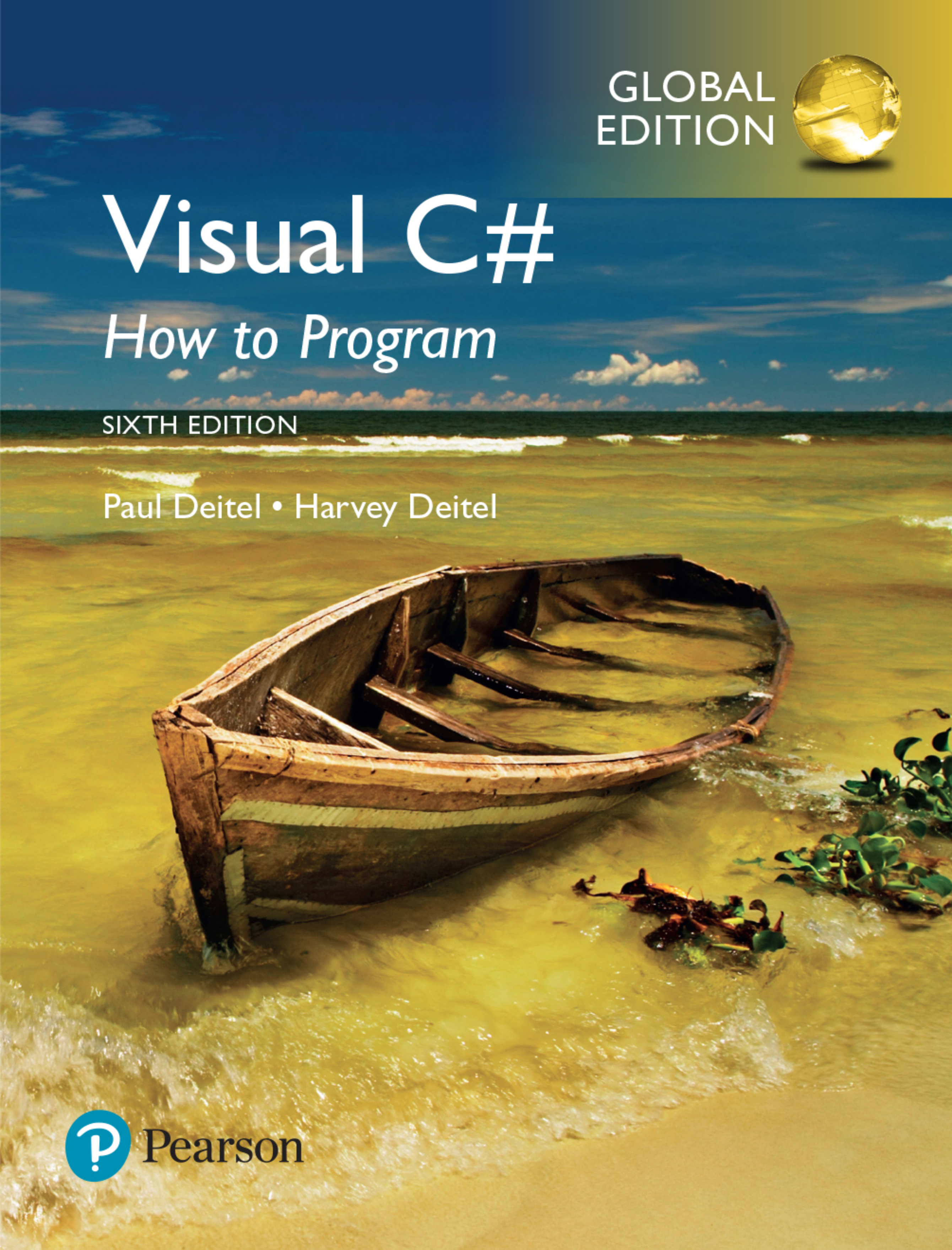
GLOBAL EDITION

# Visual C#

## *How to Program*

SIXTH EDITION

Paul Deitel • Harvey Deitel



Pearson

The factorial of an integer, `number`, greater than or equal to 0 can be calculated iteratively (nonrecursively) using the `for` statement as follows:

```
long factorial = 1;

for (long 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. 7.16. Figure 7.16(a) shows how the succession of recursive calls proceeds until 1! is evaluated to be 1, which terminates the recursion. Figure 7.16(b) shows the values returned from each recursive call to its caller until the value is calculated and returned.

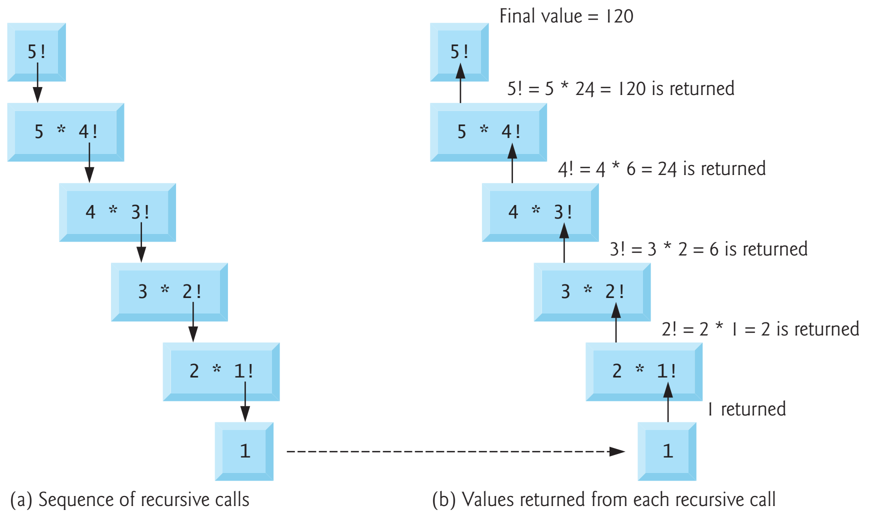

5!
5 * 4!
4 * 3!
3 * 2!
2 * 1!
1
(a) Sequence of recursive calls
Final value = 120
5!
5! = 5 * 24 = 120 is returned
5 * 4!
4! = 4 * 6 = 24 is returned
4 * 3!
3! = 3 * 2 = 6 is returned
3 * 2!
2! = 2 * 1 = 2 is returned
2 * 1!
1 returned
1
(b) Values returned from each recursive call

**Fig. 7.16** | Recursive evaluation of 5!.

### 7.16.3 Implementing Factorial Recursively

Figure 7.17 uses recursion to calculate and display the factorials of the integers from 0 to 10. The recursive method `Factorial` (lines 17–28) first tests to determine whether a terminating condition (line 20) is `true`. If `number` is less than or equal to `1` (the base case), `Factorial` returns `1` and no further recursion is necessary. If `number` is greater than `1`, line

26 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 calculation, `Factorial(number)`.

```
 1  // Fig. 7.17: FactorialTest.cs
 2  // Recursive Factorial method.
 3  using System;
 4
 5  class FactorialTest
 6  {
 7     static void Main()
 8     {
 9        // calculate the factorials of 0 through 10
10        for (long counter = 0; counter <= 10; ++counter)
11        {
12           Console.WriteLine($"{counter}! = {Factorial(counter)}");
13        }
14     }
15
16     // recursive declaration of method Factorial
17     static long Factorial(long number)
18     {
19        // base case
20        if (number <= 1)
21        {
22           return 1;
23        }
24        else // recursion step
25        {
26           return number * Factorial(number - 1);
27        }
28     }
29  }
```

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

**Fig. 7.17** | Recursive `Factorial` method.

Method `Factorial` (lines 17–28) receives a parameter of type `long` and returns a result of type `long`. As you can see in Fig. 7.17, factorial values become large quickly. We chose type `long` (which can represent relatively large integers) so that the app 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 some programming languages—the languages are *not easily extended* to handle the unique requirements of various apps. As you know, C# allows you to create new types. For example, you could create a type `HugeInteger` for arbitrarily large integers (as you'll do in Exercise 10.9). This class would enable an app to calculate the factorials of larger numbers. The .NET Framework's `BigInteger` type (from namespace `System.Numerics`) supports arbitrarily large integers.

**Common Programming Error 7.11**

*Either omitting the base case or writing the recursion step incorrectly so that it does not converge on the base case will cause infinite recursion, eventually exhausting memory. This error is analogous to the problem of an infinite loop in an iterative (nonrecursive) solution.*

## 7.17 Value Types vs. Reference Types

Types in C# are divided into two categories—*value types* and *reference types.*

### *Value Types*

C#'s simple types (like `int`, `double` and `decimal`) are all **value types**. A variable of a value type simply contains a *value* of that type. For example, Fig. 7.18 shows an `int` variable named `count` that contains the value `7`.

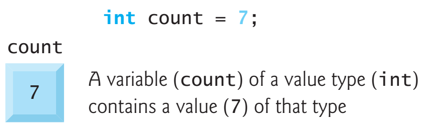

int count = 7;
count
7
A variable (count) of a value type (int) contains a value (7) of that type

**Fig. 7.18** | Value-type variable.

### *Reference Types*

By contrast, a variable of a **reference type** (also called a **reference**) contains the *location* where the data referred to by that variable is stored. Such a variable is said to **refer to an object** in the program. For example, the statement

```
Account myAccount = new Account();
```

creates an object of our class `Account` (presented in Chapter 4), places it in memory and stores the object's reference in variable `myAccount` of type `Account`, as shown in Fig. 7.19. The `Account` object is shown with its `name` instance variable.

### *Reference-Type Instance Variables Are Initialized to `null` by Default*

Reference-type instance variables (such as `myAccount` in Fig. 7.19) are initialized by default to `null`. The type `string` is a reference type. For this reason, `string` instance variable `name` is shown in Fig. 7.19 with an empty box representing the `null`-valued variable. A `string` variable with the value `null` is *not* an empty `string`, which is represented by `""` or **`string.Empty`**. Rather, the value `null` represents a reference that does *not* refer to an

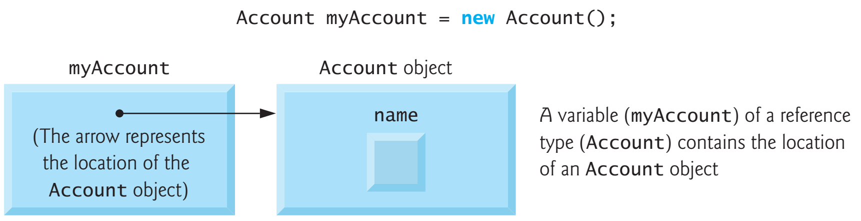

Account myAccount = new Account();
myAccount
(The arrow represents the location of the Account object)
Account object
name
A variable (myAccount) of a reference type (Account) contains the location of an Account object

**Fig. 7.19** | Reference-type variable.

object, whereas the empty `string` is a `string` object that does not contain any characters. In Section 7.18, we discuss value types and reference types in more detail.

**Software Engineering Observation 7.5**

*A variable's declared type (e.g.,* `int` *or* `Account`*) indicates whether the variable is of a value type or a reference type. If a variable's type is one of the simple types (Appendix B), an* `enum` *type or a* `struct` *type (which we introduce in Section 10.13), then it's a value type. Classes like* `Account` *are reference types.*

## 7.18 Passing Arguments By Value and By Reference

Two ways to pass arguments to methods 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 method. 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's been passed in the programs 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—no copy is passed.

To pass an object by reference into a method, simply provide as an argument in the method call the variable that refers to the object. Then, in the method body, reference the object using the corresponding parameter name. The parameter refers to the original object in memory, so the called method can access the original object directly.

In the previous section, we began discussing the differences between *value types* and *reference types*. A major difference 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, whereas
- *reference-type variables store references to objects*, so specifying a reference-type variable as an argument passes the method a *copy of the 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. When a reference is returned, the calling method can use that reference to interact with the referenced object.

**Performance Tip 7.1**

*A disadvantage of pass-by-value is that, if a large data item is being passed, copying that data can take a considerable amount of execution time and memory space.*

**Performance Tip 7.2**

*Pass-by-reference improves performance by eliminating the pass-by-value overhead of copying large objects.*

**Software Engineering Observation 7.6**

*Pass-by-reference can weaken security; the called method can corrupt the caller's data.*

### 7.18.1 `ref` and `out` Parameters

What if you would like to pass a variable by reference so the called method can modify the variable's value in the caller? To do this, C# provides keywords **`ref`** and **`out`**.

#### `ref` *Parameters*

Applying the `ref` keyword to a parameter declaration allows you to pass a variable to a method by reference—the method will be able to modify the original variable in the caller. Keyword `ref` is used for variables that already have been initialized in the calling method.

**Common Programming Error 7.12**

*When a method call contains an uninitialized variable as an argument to a* `ref` *parameter, the compiler generates an error.*

#### `out` *Parameters*

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. This also prevents the compiler from generating an error message for an uninitialized variable that's passed as an argument to a method.

**Common Programming Error 7.13**

*If the method does not assign a value to the* `out` *parameter in every possible path of execution, the compiler generates an error. Also, reading an* `out` *parameter before it's assigned a value is also a compilation error.*

**Software Engineering Observation 7.7**

*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.*

#### *Passing Reference-Type Variables by Reference*

You also can pass a reference-type variable by reference, which allows you to modify it so that it refers to a new object. Passing a reference by reference is a tricky but powerful technique that we discuss in Section 8.13.

**Software Engineering Observation 7.8**

*By default, C# does not allow you to choose whether to pass each argument by value or by reference. Value types are passed by value. Objects are not passed to methods; rather, references to objects are passed—the references themselves are passed by value. When a method receives a reference to an object, the method can manipulate the object directly, but the reference value cannot be changed to refer to a new object.*

## 7.18.2 Demonstrating `ref`, `out` and Value Parameters

The app in Fig. 7.20 uses the `ref` and `out` keywords to manipulate integer values. The class contains three methods that calculate the square of an integer.

```csharp
// Fig. 7.20: ReferenceAndOutputParameters.cs
// Reference, output and value parameters.
using System;

class ReferenceAndOutputParameters
{
   // call methods with reference, output and value parameters
   static void Main()
   {
      int y = 5; // initialize y to 5
      int z; // declares z, but does not initialize it

      // display original values of y and z
      Console.WriteLine($"Original value of y: {y}");
      Console.WriteLine("Original value of z: uninitialized\n");

      // 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're modified by
      // methods SquareRef and SquareOut, respectively
      Console.WriteLine($"Value of y after SquareRef: {y}");
      Console.WriteLine($"Value of z after SquareOut: {z}\n");

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

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

   // uses reference parameter x to modify caller's variable
   static void SquareRef(ref int x)
   {
```

**Fig. 7.20** | Reference, output and value parameters. (Part 1 of 2.)