3a

Methods: A Deeper Look

OBJECTIVES

In this lecture you will learn:

- How `static` methods and variables are associated with a class rather than specific instances of the class.
- How the method call/return mechanism is supported by the method-call stack and activation records.
- How to use random-number generation to implement game-playing applications.
- How the visibility of declarations is limited to specific regions of applications.

OBJECTIVES

- What method overloading is and how to create overloaded methods.
- What recursive methods are.
- The differences between passing method arguments by value and by reference.

Outline

Outline

7.1 Introduction

The best way to develop and maintain a large application is to construct it from small, simple pieces.

This technique is called **divide and conquer**.

7.2 Packaging Code in C#

Common ways of packaging code are properties, methods, classes and namespaces.

The Framework Class Library provides many predefined classes that contain methods for performing common tasks.

Good Programming Practice 7.1

Familiarize yourself with the classes and methods provided by the Framework Class Library.

Software Engineering Observation 7.1

Don’t try to “reinvent the wheel.” When possible, reuse Framework Class Library classes and methods.

7.2 Packaging Code in C# (Cont.)

Methods allow you to modularize an application by separating its tasks into reusable units.

Methods that you write are sometimes referred to as **user-defined methods.**

- The "divide-and-conquer" approach makes application development more manageable by constructing applications from small, simple pieces.
- Software reusability—existing methods can be used as building blocks to create new applications.
- Avoid repeating code.

Dividing an application into meaningful methods makes the application easier to debug and maintain.

7.2 Packaging Code in C# (Cont.)

Software Engineering Observation 7.2

To promote software reusability, every method should be limited to performing a single, well-defined task, and the name of the method should express that task effectively. Such methods make applications easier to write, debug, maintain and modify.

Error-Prevention Tip 7.1

A small method that performs one task is easier to test and debug than a larger method that performs many tasks.

Software Engineering Observation 7.3

If you cannot choose a concise name that expresses a method's task, your method might be attempting to perform too many diverse tasks. It is usually best to break such a method into several smaller methods.

7.2 Packaging Code in C# (Cont.)

The code that calls a method is known as the client code.

An analogy to the method-call-and-return structure is the hierarchical form of management (Figure 7.1).

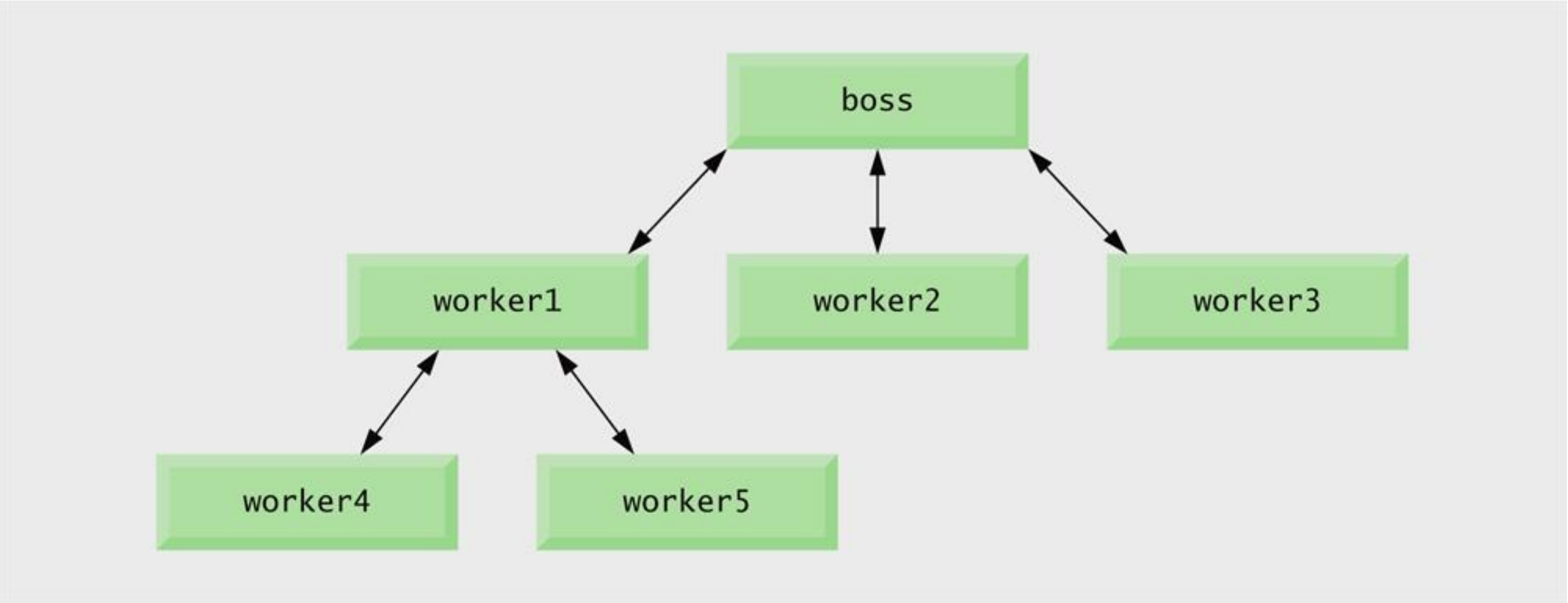


Fig. 7.1 | Hierarchical boss-method/worker-method relationship. (Part 1 of 2.)

7.2 Packaging Code in C# (Cont.)

- The boss method does not know how the worker method performs its designated tasks.
- The worker may also call other worker methods.

This “hiding” of implementation details promotes good software engineering.

7.3 `static` Methods, `static` Variables and Class `Math`

A method that applies to the class in which it is declared as a whole is known as a `static` method.

To declare a method as `static`, place the keyword `static` before the return type in the method's declaration.

You call any `static` method by specifying the name of the class in which the method is declared, followed by the member access (`.`) operator and the method name.

7.3 `static` Methods, `static` Variables and Class `Math` (Cont.)

Class `Math` provides a collection of `static` methods that enable you to perform common mathematical calculations.

You do not need to create a `Math` object before calling method `Sqrt`.

Method arguments may be constants, variables or expressions.

7.3 `static` Methods, `static` Variables and Class `Math` (Cont.)

- Figure 7.2 summarizes several `Math` class methods. In the figure, *x* and *y* are of type `double`.

Method	Description	Example

`Abs( x )`	absolute value of *x*	`Abs( 23.7 )` is `23.7`
		`Abs( 0.0 )` is `0.0`
		`Abs( -23.7 )` is `23.7`

`Ceiling( x )`	rounds *x* to the smallest integer not less than *x*	`Ceiling( 9.2 )` is `10.0` `Ceiling( -9.8 )` is `-9.0`
`Cos( x )`	trigonometric cosine of *x* (*x* in radians)	`Cos( 0.0 )` is `1.0`

Fig. 7.2 | `Math` class methods. (Part 1 of 3.)

7.3 `static` Methods, `static` Variables and Class `Math` (Cont.)

Method	Description	Example

`Abs( x )`	absolute value of *x*	`Abs( 23.7 )` is `23.7`
		`Abs( 0.0 )` is `0.0`
		`Abs( -23.7 )` is `23.7`

`Ceiling( x )`	rounds *x* to the smallest integer not less than *x*	`Ceiling( 9.2 )` is `10.0` `Ceiling( -9.8 )` is `-9.0`
`Cos( x )`	trigonometric cosine of *x* (*x* in radians)	`Cos( 0.0 )` is `1.0`

Fig. 7.2 | `Math` class methods. (Part 2 of 3.)

7.3 `static` Methods, `static` Variables and Class `Math` (Cont.)

Method	Description	Example
`Exp( x )`	exponential method e^x	`Exp( 1.0 )` is `2.71828` `Exp( 2.0 )` is `7.38906`
`Floor( x )`	rounds *x* to the largest integer not greater than *x*	`Floor( 9.2 )` is `9.0` `Floor( -9.8 )` is `-10.0`
`Log( x )`	natural logarithm of *x* (base *e*)	`Log( Math.E )` is `1.0` `Log( Math.E * Math.E )` is `2.0`
`Max( x, y )`	larger value of *x* and *y*	`Max( 2.3, 12.7 )` is `12.7` `Max( -2.3, -12.7 )` is `-2.3`

Fig. 7.2 | `Math` class methods. (Part 3 of 3.)

7.3 `static` Methods, `static` Variables and Class `Math` (Cont.)

Class `Math` also declares two `static` constants that represent commonly used mathematical values: `Math.PI` and `Math.E`.

These constants are declared in class `Math` as `public` and `const`.

- `public` allows other programmers to use these variables in their own classes.
- Keyword `const` prevents its value from being changed after the constant is declared.

7.3 `static` Methods, `static` Variables and Class `Math` (Cont.)

Common Programming Error 7.1

Every constant declared in a class, but not inside a method of the class is implicitly `static`, so it is a syntax error to declare such a constant with keyword `static` explicitly.

Because these constants are `static`, you can access them via the class name `Math` and the member access (`.`) operator, just like class `Math`'s methods.

When objects of a class containing `static` variables are created, all the objects of that class share one copy of the class's `static` variables.

Together the `static` variables and instance variables represent the **fields** of a class.

7.3 `static` Methods, `static` Variables and Class `Math` (Cont.)

Why Is Method `Main` Declared `static`?

The `Main` method is sometimes called the application's **entry point**.

Declaring `Main` as `static` allows the execution environment to invoke `Main` without creating an instance of the class.

When you execute your application from the command line, you type the application name, followed by **command-line arguments** that specify a list of `string`s separated by spaces.

The execution environment will pass these arguments to the `Main` method of your application.

7.3 static Methods, static Variables and Class Math (Cont.)

Additional Comments about Method Main

Applications that do not take command-line arguments may omit the `string[] args` parameter.

The `public` keyword may be omitted.

You can declare `Main` with return type `int` (instead of `void`) to enable `Main` to return an error code with the `return` statement.

You can declare only one `Main` method in each class.

7.3 static Methods, static Variables and Class Math (Cont.)

You can place a Main method in every class you declare.

However, you need to indicate the application's entry point.

Do this by clicking the menu **Project > [ProjectName] Properties...** and selecting the class containing the Main method that should be the entry point from the **Startup object** list box.

- A `MaximumFinder` class is presented in Fig. 7.3.

MaximumFinder.cs

(1 of 3)

```
// Fig. 7.3: MaximumFinder.cs
// User-defined method Maximum.
using System;

public class MaximumFinder
{
   // obtain three floating-point values and determine maximum value
   public void DetermineMaximum()
   {
      // prompt for and input three floating-point values
      Console.WriteLine( "Enter three floating-point values,\n"
         + "  pressing 'Enter' after each one: " );
      double number1 = Convert.ToDouble( Console.ReadLine() );
      double number2 = Convert.ToDouble( Console.ReadLine() );
      double number3 = Convert.ToDouble( Console.ReadLine() );
```

Prompt the user to enter three `double` values and read them from the user.

Fig. 7.3 | User-defined method `Maximum`. (Part 1 of 3.)

```

      // determine the maximum value
      double result = Maximum( number1, number2, number3 );

      // display maximum value
      Console.WriteLine( "Maximum is: " + result );
   } // end method DetermineMaximum

   // returns the maximum of its three double parameters
   public double Maximum( double x, double y, double z )
   {
      double maximumValue = x; // assume x is the largest to start

      // determine whether y is greater than maximumValue
      if ( y > maximumValue )
```

Call method `Maximum` to determine the largest of the three `double` values passed as arguments to the method.

The method's name is `Maximum` and that the method requires three `double` parameters to accomplish its task

Fig. 7.3 | User-defined method `Maximum`. (Part 2 of 3.)

MaximumFinder.cs

(3 of 3)

```
            maximumValue = y;

         // determine whether z is greater than maximumValue
         if ( z > maximumValue )
            maximumValue = z;

         return maximumValue;
      } // end method Maximum
} // end class MaximumFinder
```

Fig. 7.3 | User-defined method `Maximum`. (Part 3 of 3.)

- Class MaximumFinderTest (Fig. 7.4) contains the application’s entry point.

MaximumFinderTest.cs


```
// Fig. 7.4: MaximumFinderTest.cs
// Application to test class MaximumFinder.
public class MaximumFinderTest
{
   // application starting point
   public static void Main( string[] args )
   {
      MaximumFinder maximumFinder = new MaximumFinder();
      maximumFinder.DetermineMaximum();
   } // end Main
} // end class MaximumFinderTest
```

Create an object of class `MaximumFinder`

Calls the object’s `DetermineMaximum` method to produce the application’s output

Fig. 7.4 | Application to test class `MaximumFinder`. (Part 1 of 2.)

```
Enter three floating-point values,
  pressing 'Enter' after each one:
3.33
2.22
1.11
Maximum is: 3.33
```

```
Enter three floating-point values,
  pressing 'Enter' after each one:
2.22
3.33
1.11
Maximum is: 3.33
```

```
Enter three floating-point values,
  pressing 'Enter' after each one:
1.11
2.22
867.5309
Maximum is: 867.5309
```


Fig. 7.4 | Application to test class `MaximumFinder`. (Part 2 of 2.)

7.4 Declaring Methods with Multiple Parameters (Cont.)

When a method has more than one parameter, the parameters are specified as a comma-separated list.

There must be one argument in the method call for each parameter (sometimes called a **formal parameter**) in the method declaration.

Each argument must be consistent with the type of the corresponding parameter.

When program control returns from a method, that method's parameters are no longer accessible in memory.

Methods can return at most one value.

7.4 Declaring Methods with Multiple Parameters (Cont.)

Variables should be declared as fields of a class only if they are required for use in more than one method or if the application needs to save their values between calls to the class's methods.

Common Programming Error 7.2

Declaring method parameters of the same type as `float x, y` instead of `float x, float y` is a syntax error—a type is required for each parameter in the parameter list.

Software Engineering Observation 7.4

A method that has many parameters may be performing too many tasks. Consider dividing the method into smaller methods that perform the separate tasks. As a guideline, try to fit the method header on one line if possible.

7.4 Declaring Methods with Multiple Parameters (Cont.)

***Implementing Method* `Maximum` *by Reusing Method* `Math.Max`**

The entire body of our maximum method could also be implemented with nested calls to `Math.Max`, as follows:

```
return Math.Max( x, Math.Max( y, z ) );
```

Before any method can be called, all its arguments must be evaluated to determine their values.

`Math.Max( y, z )` is evaluated first, then the result is passed as the second argument to the other call to `Math.Max`

7.4 Declaring Methods with Multiple Parameters (Cont.)

Assembling Strings with String Concatenation

`string` concatenation allows you to combine `string`s using operator `+` .

When one of the `+` operator's operands is a `string`, the other is implicitly converted to a `string`, then the two are concatenated.

If a `bool` is concatenated with a `string`, the `bool` is converted to the `string` `"True"` or `"False"`.

7.4 Declaring Methods with Multiple Parameters (Cont.)

All objects have a `ToString` method that returns a `string` representation of the object.

When an object is concatenated with a `string`, the object's `ToString` method is implicitly called to obtain the `string` representation of the object.

A large `string` literal in a program can be broken into several smaller `strings` and placed them on multiple lines for readability, and reassembled using string concatenation or string

7.4 Declaring Methods with Multiple Parameters (Cont.)

Common Programming Error 7.3

If a `string` does not fit on one line, split the `string` into several smaller `strings` and use concatenation to form the desired `string.` C# also provides so-called verbatim `string` literals, which are preceeded by the @ character. Such literals can be split over multiple lines and the characters in the literal are processed exactly as they appear in the literal.

Common Programming Error 7.4

Confusing the + operator used for string concatenation with the + operator used for addition can lead to strange results. The + operator is left-associative. For example, if integer variable `y` has the value `5`, the expression `"y + 2 = " + y + 2` results in the string `"y + 2 = 52"`, not `"y + 2 = 7"`, because first the value of `y` (`5`) is concatenated with the string `"y + 2 = "`, then the value `2` is concatenated with the new larger string `"y + 2 = 5"`. The expression `"y + 2 = " + (y + 2)` produces the desired result `"y + 2 = 7"`.

7.5 Notes on Declaring and Using Methods

To access the class's non-`static` members, a `static` method must use a reference to an object of the class.

There are three ways to return control to the statement that calls a method.

- Reaching the end of the method.
- A return statement without a value.
- A return statement with a value.

7.5 Notes on Declaring and Using Methods (Cont.)

Common Programming Error 7.5

Declaring a method outside the body of a class declaration or inside the body of another method is a syntax error.

Common Programming Error 7.6

Omitting the return type in a method declaration is a syntax error.

Common Programming Error 7.7

Redeclaring a method parameter as a local variable in the method's body is a compilation error.

Common Programming Error 7.8

Forgetting to return a value from a method that should return one is a compilation error. If a return type other than `void` is specified, the method must contain a `return` statement in each possible execution path.

7.6 Method-Call Stack and Activation Records

A **stack** is a **last-in, first-out (LIFO) data structure**.

- Elements are added by **pushing** them onto the top of the stack.
- Elements are removed by **popping** them off the top of the stack.

When an application calls a method, the return address of the calling method is pushed onto the **program-execution stack**.

7.6 Method-Call Stack and Activation Records (Cont.)

The program-execution stack also stores local variables. This data is known as the **activation record** or **stack frame** of the method call.

- When a method call is made, its activation record is pushed onto the program-execution stack.
- When the method call is popped off the stack, the local variables are no longer known to the application.

If so many method calls occur that the stack runs out of memory, an error known as a **stack overflow** occurs.

7.7 Argument Promotion and Casting

Argument promotion is the implicit conversion of an argument's value to the type that the method expects to receive.

These conversions generate compile errors if they don't follow C#'s **promotion rules**; these specify which conversions can be performed without losing data.

- An `int` can be converted to a `double` without changing its value.
- A `double` cannot be converted to an `int` without loss of data.
- Converting large integer types to small integer types (e.g., `long` to `int`) can also result in changed values.

The types of the original values remain unchanged.

7.7 Argument Promotion and Casting (Cont.)

- Figure 7.5 lists the simple types alphabetically and the types to which each can be promoted.

Type	Conversion types
`bool`	no possible implicit conversions to other simple types
`byte`	`ushort`, `short`, `uint`, `int`, `ulong`, `long`, `decimal`, `float` or `double`
`char`	`ushort`, `int`, `uint`, `long`, `ulong`, `decimal`, `float` or `double`

Fig. 7.5 | Implicit conversions between simple types. (Part 1 of 2.)

7.7 Argument Promotion and Casting (Cont.)

Type	Conversion types
`decimal`	no possible implicit conversions to other simple types
`double`	no possible implicit conversions to other simple types
`float`	`double`
`int`	`long`, `decimal`, `float` or `double`
`long`	`decimal`, `float` or `double`
`sbyte`	`short`, `int`, `long`, `decimal`, `float` or `double`
`short`	`int`, `long`, `decimal`, `float` or `double`
`uint`	`ulong`, `long`, `decimal`, `float` or `double`
`ulong`	`decimal`, `float` or `double`
`ushort`	`uint`, `int`, `ulong`, `long`, `decimal`, `float` or `double`

Fig. 7.5 | Implicit conversions between simple types. (Part 2 of 2.)

7.7 Argument Promotion and Casting (Cont.)

All simple types can also be implicitly converted to type object.

In cases where information may be lost, you must use a cast operator to explicitly force the conversion.

Common Programming Error 7.9

Converting a simple-type value to a value of another simple type may change the value if the promotion is not allowed. For example, converting a floating-point value to an integral value may introduce truncation errors (loss of the fractional part) in the result.

7.8 The .NET Class Library

Predefined classes are grouped into categories of related classes called namespaces.

Together, these namespaces are referred to as the .NET Class Library.

7.8 The .NET Class Library (Cont.)

NET APIs include **classes**, interfaces, delegates, and value types that expedite and optimize the development process and provide access to system functionality. To facilitate interoperability between languages, most .NET types are CLS-compliant and can therefore be used from any programming language whose **compiler conforms to the common language specification** (CLS).

Namespaces have the following properties:

- They organize large code projects.
- They're delimited by using the . operator.
- They may be nested
- The **`using`** directive obviates the requirement to specify the name of the namespace for every class.
- The global namespace is the "root" namespace: **`global::System`** will always refer to the .NET **`System`** namespace.

```
using firstNS;
using firstNS.secondNS;
namespace firstNS
{
    class MyClass
    {
        public void Method()
        {
            Console.WriteLine("Helloworld.....");
        }
    }
    namespace secondNS
    {
        class MyClass
        {
            public void Method()
            {
                Console.WriteLine("This is example of nested namespace.....");
            }
        }
    }
}
class NestedNamespaceDemo  // in the default namespace
{
    static void Main(string[] args)   {
        firstNS.MyClass cls = new();
        firstNS.secondNS.MyClass cls1 = new();
        cls.Method();
        cls1.Method();
        Console.ReadKey();
    }
}
```

7.8 The .NET Class Library (Cont.)

Namespace	Description
`System.IO`	Namespace is used for I/O related type Files and Buffering..

`System.Windows.Controls`
`System.Windows.Input`
`System.Windows.Media`
`System.Windows.Shapes`
	Contain the classes of the Windows Presentation Foundation for GUIs, 2-D and 3-D graphics, multimedia and animation.

`System.Linq`	Contains the classes that support Language Integrated Query (LINQ).
`System.Data` `System.Data.Linq`	Contain the classes for manipulating data in databases (i.e., organized collections of data), including support for LINQ to SQL.

Fig. 7.6 | .NET Class Library namespaces (a subset). (Part 1 of 2.)

7.8 The .NET Class Library (Cont.)

Namespace	Description
`System.Threading`	This Application is used when we are making a multithreading application in .Net .
`System.Web`	Contains classes used for creating and maintaining web applications, which are accessible over the Internet.
`System.Xml.Linq`	Contains the classes that support Language Integrated Query (LINQ) for XML documents.
`System.Xml`	Contains classes for creating and manipulating XML data. Data can be read from or written to XML files.
`System.Collections` `System.Collections.Generic`	Contain classes that define data structures for maintaining collections of data.
`System.Text`	Contains classes that enable programs to manipulate characters and strings.

Fig. 7.6 | .NET Class Library namespaces (a subset). (Part 2 of 2.)

7.8 The .NET Class Library (Cont.)

Good Programming Practice 7.2

The online .NET documentation is easy to search and provides many details about each class. As you learn each class in this book, you should review the class in the online documentation for additional information.

7.9 Case Study: Random-Number Generation

Objects of class `Random` can produce random `byte`, `int` and `double` values.

Method `Next` of class `Random` generates a random `int` value.

The values returned by `Next` are actually **pseudorandom numbers**—a sequence of values produced by a complex mathematical calculation.

The calculation uses the current time of day to **seed** the random-number generator.

7.9 Case Study: Random-Number Generation (Cont.)

If you supply the `Next` method with an argument—called the **scaling factor**—it returns a value from 0 up to, but not including, the argument's value.

You can also **shift** the range of numbers produced by adding a **shifting value** to the number returned by the `Next` method.

Finally, if you provide `Next` with two `int` arguments, it returns a value from the first argument's value up to, but not including, the second argument's value.

Rolling a Six-Sided Die

- Figure 7.7 shows two sample outputs of an application that simulates 20 rolls of a six-sided die and displays each roll's value.


```
// Fig. 7.7: RandomIntegers.cs
// Shifted and scaled random integers.
using System;

public class RandomIntegers
{
   public static void Main( string[] args )
   {
      Random randomNumbers = new Random(); // random-number generator
      int face; // stores each random integer generated

      // loop 20 times
      for ( int counter = 1; counter <= 20; counter++ )
      {
         // pick random integer from 1 to 6
```

Create the `Random` object `randomNumbers` to produce random values.

Fig. 7.7 | Shifted and scaled random integers. (Part 1 of 2.)

```
            face = randomNumbers.Next( 1, 7 );

            Console.Write( "{0}  ", face ); // display generated value

            // if counter is divisible by 5, start a new line of output
            if ( counter % 5 == 0 )
               Console.WriteLine();
         } // end for
      } // end Main
} // end class RandomIntegers
```

```
3  3  3  1  1
2  1  2  4  2
2  3  6  2  5
3  4  6  6  1
```

```
6  2  5  1  3
5  2  1  6  5
4  1  6  1  3
3  1  4  3  4
```

RandomIntegers.cs

(2 of 2)

Call `Next` with two arguments.

Fig. 7.7 | Shifted and scaled random integers. (Part 2 of 2.)

Rolling a Six-Sided Die 6000 Times

- The application in Fig. 7.8 simulates 6000 rolls of a die. Each integer from 1 to 6 should appear approximately 1000 times.

RollDie.cs

(1 of 4)

```
// Fig. 7.8: RollDie.cs
// Roll a six-sided die 6000 times.
using System;

public class RollDie
{
   public static void Main( string[] args )
   {
      Random randomNumbers = new Random(); // random-number generator

      int frequency1 = 0; // count of 1s rolled
      int frequency2 = 0; // count of 2s rolled
      int frequency3 = 0; // count of 3s rolled
      int frequency4 = 0; // count of 4s rolled
      int frequency5 = 0; // count of 5s rolled
      int frequency6 = 0; // count of 6s rolled
```

Fig. 7.8 | Roll a six-sided die 6000 times. (Part 1 of 4.)

Outline

RollDie.cs

(2 of 4)

```
      int face; // stores most recently rolled value

      // summarize results of 6000 rolls of a die
      for ( int roll = 1; roll <= 6000; roll++ )
      {
         face = randomNumbers.Next( 1, 7 ); // number from 1 to 6

         // determine roll value 1-6 and increment appropriate counter
         switch ( face )
         {
            case 1:
               ++frequency1; // increment the 1s counter
               break;
            case 2:
               ++frequency2; // increment the 2s counter
               break;
            case 3:
               ++frequency3; // increment the 3s counter
               break;
```

Fig. 7.8 | Roll a six-sided die 6000 times. (Part 2 of 4.)

RollDie.cs

(3 of 4)

```
            case 4:
               ++frequency4; // increment the 4s counter
               break;
            case 5:
               ++frequency5; // increment the 5s counter
               break;
            case 6:
               ++frequency6; // increment the 6s counter
               break;
         } // end switch
      } // end for

      Console.WriteLine( "Face\tFrequency" ); // output headers
      Console.WriteLine(
         "1\t{0}\n2\t{1}\n3\t{2}\n4\t{3}\n5\t{4}\n6\t{5}", frequency1,
         frequency2, frequency3, frequency4, frequency5, frequency6 );
   } // end Main
} // end class RollDie
```

Fig. 7.8 | Roll a six-sided die 6000 times. (Part 3 of 4.)

RollDie.cs

(4 of 4)

```
Face   Frequency
1      1039
2      994
3      991
4      970
5      978
6      1028
```

```
Face   Frequency
1      985
2      985
3      1001
4      1017
5      1002
6      1010
```

Fig. 7.8 | Roll a six-sided die 6000 times. (Part 4 of 4.)

- The values produced by method `Next` enable the application to realistically simulate rolling a six-sided die.

7.9 Case Study: Random-Number Generation (Cont.)

7.9.1 Scaling and Shifting Random Numbers

Given two arguments, the next method allows scaling and shifting as follows:

`number` = `randomNumbers.Next(` *shiftingValue*, *shiftingValue* + *scalingFactor*);

- *shiftingValue* specifies the first number in the desired range of consecutive integers.
- *scalingFactor* specifies how many numbers are in the range.

7.9 Case Study: Random-Number Generation (Cont.)

To choose integers at random from sets of values other than ranges of consecutive integers, it is simpler to use the version of the `Next` method that takes only one argument:

`number` = *shiftingValue* + *differenceBetweenValues* * `randomNumbers.Next(` *scalingFactor*);

- *shiftingValue* specifies the first number in the desired range of values.
- *differenceBetweenValues* represents the difference between consecutive numbers in the sequence.
- *scalingFactor* specifies how many numbers are in the range.

7.9 Case Study: Random-Number Generation (Cont.)

7.9.2 Random-Number Repeatability for Testing and Debugging

The calculation that produces the pseudorandom numbers uses the time of day as a **seed value** to change the sequence's starting point.

You can pass a seed value to the `Random` object's constructor.

Given the same seed value, the `Random` object will produce the same sequence of random numbers.

7.9 Case Study: Random-Number Generation (Cont.)

Error-Prevention Tip 7.2

While an application is under development, create the `Random` object with a specific seed value to produce a repeatable sequence of random numbers each time the application executes. If a logic error occurs, fix the error and test the application again with the same seed value—this allows you to reconstruct the same sequence of random numbers that caused the error. Once the logic errors have been removed, create the `Random` object without using a seed value, causing the `Random` object

7.10 Case Study: A Game of Chance (Introducing Enumerations)

The rules of the dice game craps are as follows:

You roll two dice. Each die has six faces, which contain one, two, three, four, five and six spots, respectively. After the dice have come to rest, the sum of the spots on the two upward faces is calculated. If the sum is 7 or 11 on the first throw, you win. If the sum is 2, 3 or 12 on the first throw (called "craps"), you lose (i.e., "the house" wins). If the sum is 4, 5, 6, 8, 9 or 10 on the first throw, that sum becomes your "point." To win, you must continue rolling the dice until you "make your point" (i.e., roll that same point value). You lose by rolling a 7 before making your point.

- The declaration of class `Craps` is shown in Fig. 7.9.

`Craps.cs`


```
// Fig. 7.9: Craps.cs
// Craps class simulates the dice game craps.
using System;

public class Craps
{
   // create random-number generator for use in method RollDice
   private Random randomNumbers = new Random();

   // enumeration with constants that represent the game status
   private enum Status { CONTINUE, WON, LOST }

   // enumeration with constants that represent common rolls of the dice
```

A user-defined type called an **enumeration** declares a set of constants represented by identifiers, and is introduced by the keyword `enum` and a type name.

Fig. 7.9 | `Craps` class simulates the dice game craps. (Part 1 of 4.)

Craps.cs

(2 of 4)

```
   private enum DiceNames
   {
      SNAKE_EYES = 2,
      TREY = 3,
      SEVEN = 7,
      YO_LEVEN = 11,
      BOX_CARS = 12
   }

   // plays one game of craps
   public void Play()
   {
      // gameStatus can contain CONTINUE, WON or LOST
      Status gameStatus = Status.CONTINUE;
      int myPoint = 0; // point if no win or loss on first roll

      int sumOfDice = RollDice(); // first roll of the dice

      // determine game status and point based on first roll
      switch ( ( DiceNames ) sumOfDice )
      {
         case DiceNames.SEVEN: // win with 7 on first roll
         case DiceNames.YO_LEVEN: // win with 11 on first roll
            gameStatus = Status.WON;
```

Sums of the dice that would result in a win or loss on the first roll are declared in an enumeration.

Initialization is not strictly necessary because it is assigned a value in every branch of the `switch` statement.

Must be initialized to 0 because it is not assigned a value in every branch of the `switch` statement.

Call method `RollDice` for the first roll of the game.

Fig. 7.9 | `Craps` class simulates the dice game craps. (Part 2 of 4.)

```
            break;
         case DiceNames.SNAKE_EYES: // lose with 2 on first roll
         case DiceNames.TREY: // lose with 3 on first roll
         case DiceNames.BOX_CARS: // lose with 12 on first roll
            gameStatus = Status.LOST;
            break;
         default: // did not win or lose, so remember point
            gameStatus = Status.CONTINUE; // game is not over
            myPoint = sumOfDice; // remember the point
            Console.WriteLine( "Point is {0}", myPoint );
            break;
      } // end switch

      // while game is not complete
      while ( gameStatus == Status.CONTINUE ) // game not WON or LOST
      {
         sumOfDice = RollDice(); // roll dice again

         // determine game status
         if ( sumOfDice == myPoint ) // win by making point
            gameStatus = Status.WON;
         else
            // lose by rolling 7 before point
            if ( sumOfDice == ( int ) DiceNames.SEVEN )
               gameStatus = Status.LOST;
      } // end while
```

Call method `RollDice` for subsequent rolls.

Fig. 7.9 | `Craps` class simulates the dice game craps. (Part 3 of 4.)

```
64
65       // display won or lost message
66       if ( gameStatus == Status.WON )
67          Console.WriteLine( "Player wins" );
68       else
69          Console.WriteLine( "Player loses" );
70    } // end method Play
71
72    // roll dice, calculate sum and display results
73    public int RollDice()
74    {
75       // pick random die values
76       int die1 = randomNumbers.Next( 1, 7 ); // first die roll
77       int die2 = randomNumbers.Next( 1, 7 ); // second die roll
78
79       int sum = die1 + die2; // sum of die values
80
81       // display results of this roll
82       Console.WriteLine( "Player rolled {0} + {1} = {2}",
83          die1, die2, sum );
84       return sum; // return sum of dice
85    } // end method RollDice
86 } // end class Craps
```

Craps.cs

(4 of 4)

Declare method `RollDice` to roll the dice and compute and display their sum.

Fig. 7.9 | `Craps` class simulates the dice game craps. (Part 4 of 4.)

7.10 Case Study: A Game of Chance (Introducing Enumerations) (Cont.)

A user-defined type called an **enumeration** declares a set of constants represented by identifiers, and is introduced by the keyword **`enum`** and a type name.

As with a class, braces (`{` and `}`) delimit the body of an `enum` declaration. Inside the braces is a comma-separated list of **enumeration constants**.

The `enum` constant names must be unique, but the value associated with each constant need not be.

7.10 Case Study: A Game of Chance (Introducing Enumerations) (Cont.)

Good Programming Practice 7.3

Use only uppercase letters in the names of constants. This makes the constants stand out in an application and reminds you that enumeration constants are not variables.

- Variables of an enumeration type should be assigned only the values declared in the enumeration.

Good Programming Practice 7.4

Using enumeration constants (like `Status.WON`, `Status.LOST` and `Status.CONTINUE`) rather than literal integer values (such as 0, 1 and 2) can make code easier to read and maintain.

7.10 Case Study: A Game of Chance (Introducing Enumerations) (Cont.)

- When an `enum` is declared, each constant in the `enum` declaration is a constant value of type `int`.
- If you do not assign a value to an identifier in the enum declaration, the compiler will do so.
  - If the first `enum` constant is unassigned, the compiler gives it the value 0.
  - If any other `enum` constant is unassigned, the compiler gives it a value equal to one more than the value of the preceding `enum` constant.

7.10 Case Study: A Game of Chance (Introducing Enumerations) (Cont.)

- You can declare an enum’s underlying type to be `byte`, `sbyte`, `short`, `ushort`, `int`, `uint`, `long` or `ulong` by writing

`private enum MyEnum` : *typeName* `{` *Constant1* `,` *Constant2* `,` ... `}`

  – *typeName* represents one of the integral simple types.

- To compare a simple integral type value to the underlying value of an enumeration constant, you must use a cast operator.

- The `Main` method is in class `CrapsTest` (Fig. 7.10).

```
// Fig. 7.10: CrapsTest.cs
// Application to test class Craps.
public class CrapsTest
{
   public static void Main( string[] args )
   {
      Craps game = new Craps();
      game.Play(); // play one game of craps
   } // end Main
} // end class CrapsTest
```

Create an object of class `Craps`.

Call the `game` object's `Play` method to start the game.

Fig. 7.10 | Application to test class `Craps`. (Part 1 of 2.)

```
Player rolled 2 + 5 = 7
Player wins
```

```
Player rolled 2 + 1 = 3
Player loses
```

```
Player rolled 4 + 6 = 10
Point is 10
Player rolled 1 + 3 = 4
Player rolled 1 + 3 = 4
Player rolled 2 + 3 = 5
Player rolled 4 + 4 = 8
Player rolled 6 + 6 = 12
Player rolled 4 + 4 = 8
Player rolled 4 + 5 = 9
Player rolled 2 + 6 = 8
Player rolled 6 + 6 = 12
Player rolled 6 + 4 = 10
Player wins
```

```
Player rolled 2 + 4 = 6
Point is 6
Player rolled 3 + 1 = 4
Player rolled 5 + 5 = 10
Player rolled 6 + 1 = 7
Player loses
```

CrapsTest.cs

(2 of 2)

Fig. 7.10 | Application to test class `Craps`. (Part 2 of 2.)

7.11 Scope of Declarations

The scope of a declaration is the portion of the application that can refer to the declared entity by its unqualified name.

The basic scope rules are as follows:

- The scope of a parameter declaration is the body of the method in which the declaration appears.
- The scope of a local-variable declaration is from the point at which the declaration appears to the end of the block containing the declaration.
- The scope of a non-`static` method, property or field of a class is the entire body of the class.

If a local variable or parameter in a method has the same name as a field, the field is hidden until the block terminates.

7.11 Scope of Declarations (Cont.)

Error-Prevention Tip 7.3

Use different names for fields and local variables to help prevent subtle logic errors that occur when a method is called and a local variable of the method hides a field of the same name in the class.

Class Scope (Fig. 7.11) demonstrates scoping issues with fields and local variables.

Scope.cs

(1 of 3)

```
1  // Fig. 7.11: Scope.cs
2  // Scope class demonstrates instance- and local-variable scopes.
3  using System;
4
5  public class Scope
6  {
7     // instance variable that is accessible to all methods of this class
8     private int x = 1;
9
10    // method Begin creates and initializes local variable x
11    // and calls methods UseLocalVariable and UseInstanceVariable
12    public void Begin()
13    {
14       int x = 5; // method's local variable x hides instance variable x
15
16       Console.WriteLine( "local x in method Begin is {0}", x );
```

Local variable x hides instance variable x (declared in line 8) in method Begin.

Fig. 7.11 | Scope class demonstrates instance- and local-variable scopes. (Part 1 of 3.)

Outline

Scope.cs

(2 of 3)

```
      // UseLocalVariable has its own local x
      UseLocalVariable();

      // UseInstanceVariable uses class Scope's instance variable x
      UseInstanceVariable();

      // UseLocalVariable reinitializes its own local x
      UseLocalVariable();

      // class Scope's instance variable x retains its value
      UseInstanceVariable();

      Console.WriteLine( "\nlocal x in method Begin is {0}", x );
   } // end method Begin

   // create and initialize local variable x during each call
   public void UseLocalVariable()
   {
```

Fig. 7.11 | Scope class demonstrates instance- and local-variable scopes. (Part 2 of 3.)

Outline

Scope.cs

(3 of 3)

```
36       int x = 25; // initialized each time UseLocalVariable is called
37
38       Console.WriteLine(
39          "\nlocal x on entering method UseLocalVariable is {0}", x );
40       ++x; // modifies this method's local variable x
41       Console.WriteLine(
42          "local x before exiting method UseLocalVariable is {0}", x );
43    } // end method UseLocalVariable
44
45    // modify class Scope's instance variable x during each call
46    public void UseInstanceVariable()
47    {
48       Console.WriteLine( "\ninstance variable x on entering {0} is {1}",
49          "method UseInstanceVariable", x );
50       x *= 10; // modifies class Scope's instance variable x
51       Console.WriteLine( "instance variable x before exiting {0} is {1}",
52          "method UseInstanceVariable", x );
53    } // end method UseInstanceVariable
54 } // end class Scope
```

Local variable `x` is declared within `UseLocalVariable` and goes out of scope when the method returns.

Because no local variable x is declared in `UseInstanceVariable`, instance variable `x` (line 8) of the class is used.

Fig. 7.11 | `Scope` class demonstrates instance- and local-variable scopes. (Part 3 of 3.)

A class that tests the `Scope` class is shown in Fig. 7.12

`ScopeTest.cs`

(1 of 2)

```
// Fig. 7.12: ScopeTest.cs
// Application to test class Scope.
public class ScopeTest
{
   // application starting point
   public static void Main( string[] args )
   {
      Scope testScope = new Scope();
      testScope.Begin();
   } // end Main
} // end class ScopeTest
```

Fig. 7.12 | Application to test class `Scope`. (Part 1 of 2.)

Outline

ScopeTest.cs


```
local x in method Begin is 5

local x on entering method UseLocalVariable is 25
local x before exiting method UseLocalVariable is 26

instance variable x on entering method UseInstanceVariable is 1
instance variable x before exiting method UseInstanceVariable is 10

local x on entering method UseLocalVariable is 25
local x before exiting method UseLocalVariable is 26

instance variable x on entering method UseInstanceVariable is 10
instance variable x before exiting method UseInstanceVariable is 100

local x in method Begin is 5
```

Fig. 7.12 | Application to test class `Scope`. (Part 2 of 2.)

7.12 Method Overloading

Methods of the same name can be declared in the same class, or **overloaded**, as long as they have different sets of parameters.

When an **overloaded method** is called, the C# compiler selects the appropriate method by examining the number, types and order of the arguments in the call.

Method overloading is used to create several methods with the same name that perform the same tasks, but on different types or numbers of arguments.

Declaring Overloaded Methods

Class `MethodOverload` (Fig. 7.13) includes two overloaded versions of a method called `Square`.


```
// Fig. 7.13: MethodOverload.cs
// Overloaded method declarations.
using System;

public class MethodOverload
{
   // test overloaded square methods
   public void TestOverloadedMethods()
   {
      Console.WriteLine( "Square of integer 7 is {0}", Square( 7 ) );
      Console.WriteLine( "Square of double 7.5 is {0}", Square( 7.5 ) );
   } // end method TestOverloadedMethods

   // square method with int argument
   public int Square( int intValue )
```

Overloaded version of the method that operates on an integer.

Overloaded version of the method that operates on a double.

Fig. 7.13 | Overloaded method declarations. (Part 1 of 2.)

```
   {
      Console.WriteLine( "Called square with int argument: {0}",
         intValue );
      return intValue * intValue;
   } // end method Square with int argument

   // square method with double argument
   public double Square( double doubleValue )
   {
      Console.WriteLine( "Called square with double argument: {0}",
         doubleValue );
      return doubleValue * doubleValue;
   } // end method Square with double argument
} // end class MethodOverload
```

Overloaded version of the method that operates on a double.

Fig. 7.13 | Overloaded method declarations. (Part 2 of 2.)

Class `MethodOverloadTest` (Fig. 7.14) tests class `MethodOverload`.

MethodOverloadTest.cs

```
// Fig. 7.14: MethodOverloadTest.cs
// Application to test class MethodOverload.
public class MethodOverloadTest
{
   public static void Main( string[] args )
   {
      MethodOverload methodOverload = new MethodOverload();
      methodOverload.TestOverloadedMethods();
   } // end Main
} // end class MethodOverloadTest
```

```
Called square with int argument: 7
Square of integer 7 is 49
Called square with double argument: 7.5
Square of double 7.5 is 56.25
```

Fig. 7.14 | Application to test class `MethodOverload`.

Distinguishing Between Overloaded Methods

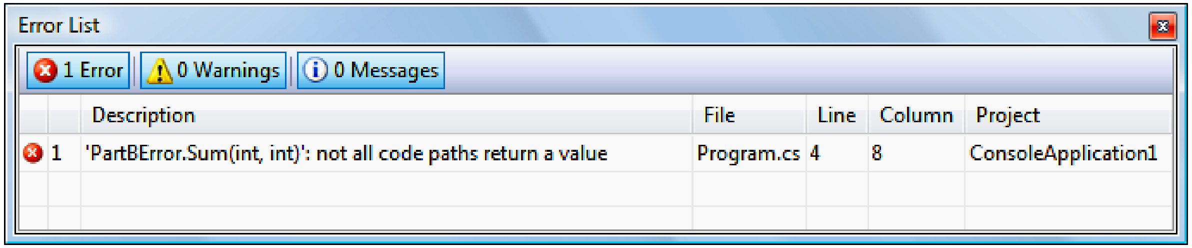
- The compiler distinguishes overloaded methods by their **signature**—a combination of the method's name and the number, types and order of its parameters.

Return Types of Overloaded Methods

- Method calls cannot be distinguished by return type.
- The application in Fig. 7.15 illustrates the compiler errors generated when two methods have the same signature but different return types.

```
// Fig. 7.15: MethodOverload.cs
// Overloaded methods with identical signatures
// cause compilation errors, even if return types are different.
public class MethodOverloadError
{
   // declaration of method Square with int argument
   public int Square( int x )
   {
      return x * x;
   }

   // second declaration of method Square with int argument
   // causes compilation error even though return types are different
   public double Square( int y )
   {
      return y * y;
   }
} // end class MethodOverloadError
```



	Description	File	Line	Column	Project
1	'PartBError.Sum(int, int)': not all code paths return a value	Program.cs	4	8	ConsoleApplication1

Fig. 7.15 | Overloaded methods with identical signatures cause compilation errors, even if return types are different.

- Overloaded methods can have the same or different return types if the methods have different parameter lists.
- Overloaded methods need not have the same number of parameters.

Common Programming Error 7.10

Declaring overloaded methods with identical parameter lists is a compilation error regardless of whether the return types are different.

7.13 Recursion

A **recursive method** is a method that calls itself.

A recursive method is capable of solving only the **base case(s)**.

Each method call divides the problem into two conceptual pieces: a piece that the method knows how to do and a **recursive call**, or **recursion step** that solves a smaller problem.

A sequence of returns ensues until the original method call returns the result to the caller.

7.13 Recursion (Cont.)

Recursive Factorial Calculations

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

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

Figure 7.16(b) shows the values returned from each recursive call in 5! to its caller until the value is calculated and returned.

7.13 Recursion (Cont.)

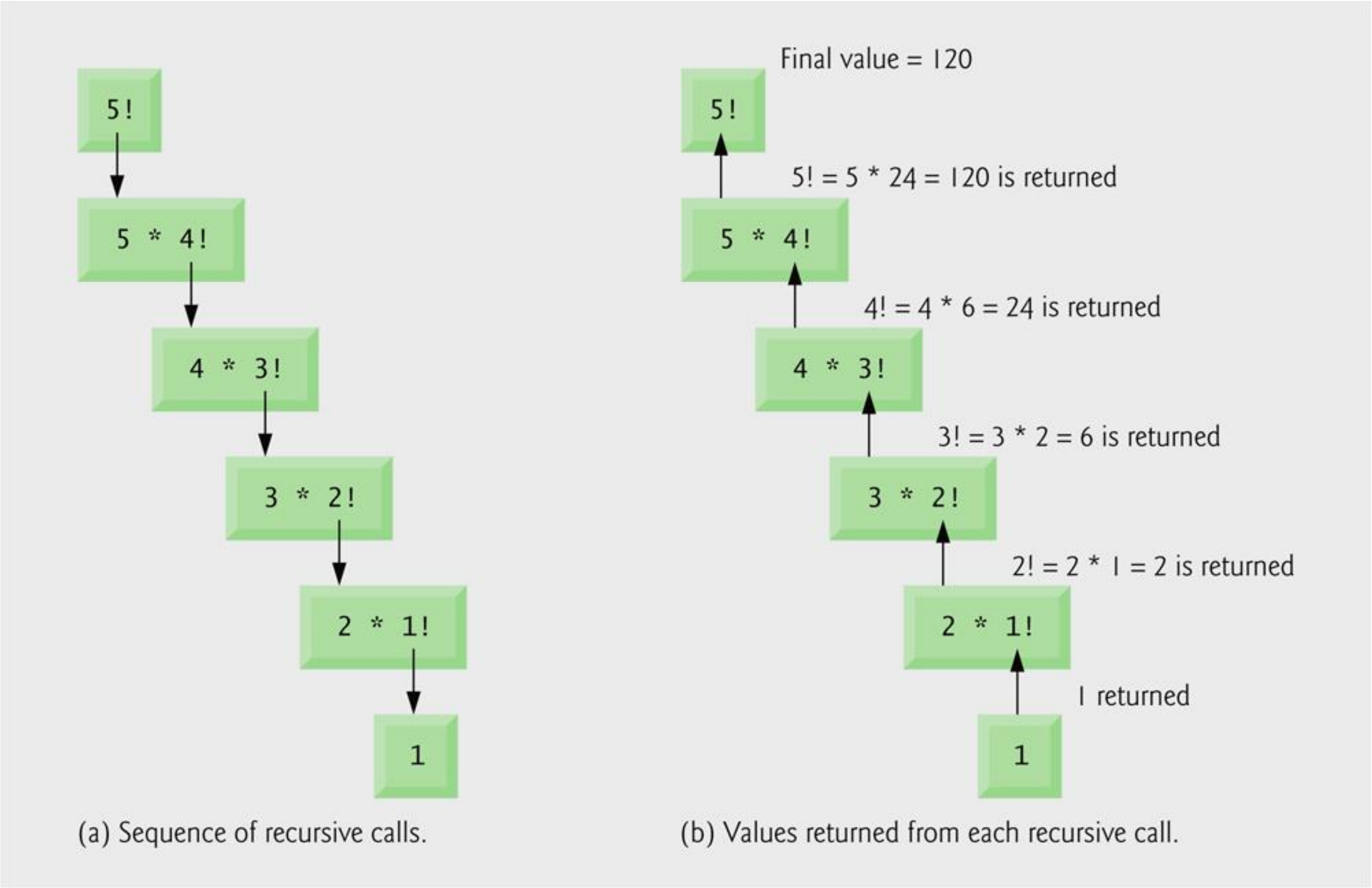


Fig. 7.16 | Recursive evaluation of 5!.

Figure 7.17 uses recursion to calculate and display the factorials of the integers from 0 to 10.


```
// Fig. 7.17: 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
```

Fig. 7.17 | Recursive `Factorial` method. (Part 1 of 2.)

```
   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
```

First, test to determine whether the terminating condition is `true`.

The recursive call solves a slightly simpler problem than the original calculation.

```
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. (Part 2 of 2.)

7.13 Recursion (Cont.)

First, test to determine whether the terminating condition is `true`.

The recursive call solves a slightly simpler problem than the original calculation.

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.

7.14 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.

When an argument is passed by reference, the caller gives the method the ability to access and modify the caller’s original variable.

7.14 Passing Arguments: Pass-by-Value vs. Pass-by-Reference (Cont.)

Performance Tip 7.1

Pass-by-reference is good for performance reasons, because it can eliminate the pass-by-value overhead of copying large amounts of data.

Software Engineering Observation 7.5

Pass-by-reference can weaken security, because the called function can corrupt the caller's data.

7.14 Passing Arguments: Pass-by-Value vs. Pass-by-Reference (Cont.)

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.

In the method body, the parameter will refer to the original object in memory, so the called method can access the original object directly.

Passing a **value-type variable** to a method **passes a copy of the value.**

7.14 Passing Arguments: Pass-by-Value vs. Pass-by-Reference (Cont.)

Passing a reference-type variable passes the method a copy of the actual reference that refers to the object.

– The reference itself is passed by value, but the method can still use the reference it receives to modify the original object in memory.

A `return` statement returns a copy of the value stored in a value-type variable or a copy of the reference stored in a reference-type variable.

In effect, objects are always passed by reference.

7.14 Passing Arguments: Pass-by-Value vs. Pass-by-Reference (Cont.)

- Applying the `ref` keyword to a parameter declaration allows you to pass a variable to a method by reference
- The `ref` keyword is used for variables that already have been initialized in the calling method.
- Preceding a parameter with keyword `out` creates an **output parameter**.
- This indicates to the compiler that **the argument will be passed by reference** and that the **called method will assign a value to it**.
- A method can return multiple output parameters.

7.14 Passing Arguments: Pass-by-Value vs. Pass-by-Reference (Cont.)

The `in` keyword complements the existing `ref` and `out` keywords to pass arguments by reference. The `in` keyword specifies passing the argument by reference, but **the called method doesn't modify the value**.

You may declare overloads that pass by value or by `readonly` reference, as shown in the following code:

```
static void M(S arg);
static void M(in S arg);
```

The first example is **passing by value** is **better** (preferred) than the by `readonly` reference version. To call the version with the `readonly` reference argument, you must include the `in` modifier when calling the method.

7.14 Passing Arguments: Pass-by-Value vs. Pass-by-Reference (Cont.)

Note:

- `ref` arguments may be modified. Variables passed as `in` arguments must be initialized before being passed in a method call
- `out` arguments must be modified by the called method
- `in` arguments must be initialized before being passed in a method call. However, the called method may not assign a value or modify the argument
- Modifications of `ref` and `out` arguments are observable in the calling context

Class `ReferenceAndOutputParameters` (Fig. 7.18) contains three methods that calculate the square of an integer.

ReferenceAndOutputParameters.cs


```
// Fig. 7.18: ReferenceAndOutputParameters.cs
// Reference, output and value parameters.
using System;

class ReferenceAndOutputParameters
{
   // call methods with reference, output and value parameters
   public void DemonstrateReferenceAndOutputParameters()
   {
      int y = 5; // initialize y to 5
```

Fig. 7.18 | Reference, output and value parameters. (Part 1 of 4.)

Outline

ReferenceAndOutputParameters.cs

(2 of 4)

```
      int z; // declares z, but does not initialize it

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

When you pass a variable to a method with a reference parameter, you must precede the argument with the same keyword (`ref` or `out`) that was used to declare the reference parameter.

Fig. 7.18 | Reference, output and value parameters. (Part 2 of 4.)

Outline

ReferenceAndOutputParameters.cs

(3 of 4)

```
      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 method DemonstrateReferenceAndOutputParameters

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

```

Modify caller's x.

Fig. 7.18 | Reference, output and value parameters. (Part 3 of 4.)

ReferenceAndOutputParameters.cs


```

   // uses output parameter x to assign a value
   // to an uninitialized variable
   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
   void Square( int x )
   {
      x = x * x;
   } // end method Square
} // end class ReferenceAndOutputParameters
```

Assign a value to caller's uninitialized `x`.

Doesn't modify any caller variable.

Fig. 7.18 | Reference, output and value parameters. (Part 4 of 4.)

7.14 Passing Arguments: Pass-by-Value vs. Pass-by-Reference (Cont.)

When you pass a variable to a method with a reference parameter, you must precede the argument with the same keyword (`ref` or `out`) that was used to declare the reference parameter.

Common Programming Error 7.12

The `ref` and `out` arguments in a method call must match the parameters specified in the method declaration; otherwise, a compilation error occurs.

7.14 Passing Arguments: Pass-by-Value vs. Pass-by-Reference (Cont.)

Software Engineering Observation 7.6

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 to methods. 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.

Class `ReferenceAndOutputParametersTest` tests the `ReferenceAndOutputParameters` class.

ReferenceAndOutputParamtersTest.cs

```
// Fig. 7.19: ReferenceAndOutputParamtersTest.cs
// Application to test class ReferenceAndOutputParameters.
class ReferenceAndOutputParamtersTest
{
   public static void Main( string[] args )
   {
      ReferenceAndOutputParameters test =
         new ReferenceAndOutputParameters();
      test.DemonstrateReferenceAndOutputParameters();
   } // end Main
} // end class ReferenceAndOutputParamtersTest
```

```
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. 7.19 | Application to test class `ReferenceAndOutputParameters`.

7.14 Passing Arguments: Pass-by-Value vs. Pass-by-Reference (Cont.)

Currently in C#, using **`out`** parameters isn't as fluid as we'd like. Before you can call a method with out parameters you first have to declare variables to pass to it. Since you typically aren't initializing these variables (they are going to be overwritten by the method after all), you also cannot use **`var`** to declare them, but need to specify the full type:

```
public void PrintCoordinates(Point p)
{
    int x, y; // have to "predeclare"
    p.GetCoordinates(out x, out y);
    Console.WriteLine($"({x}, {y})");
}
```

7.14 Passing Arguments: Pass-by-Value vs. Pass-by-Reference (Cont.)

C# 7.0 introduces **`out`** ***variables***; the ability to **declare a variable right at the point where it is passed as an `out` argument**:

```
public void PrintCoordinates(Point p)
{
    p.GetCoordinates(out int x, out int y);
    Console.WriteLine($"({x}, {y})");
}
```

Note that the variables are in scope in the enclosing block, so **the subsequent line can use them**. Most kinds of statements do not establish their own scope, so **`out`** variables declared in them **are usually introduced into the enclosing scope**.

7.14 Passing Arguments: Pass-by-Value vs. Pass-by-Reference (Cont.)

A common use of **`out parameters`** is the **`Try...`** pattern, where a **`boolean`** return value indicates success, and **`out`** parameters carry the results obtained:

```
public void PrintStars(string s)
{
    if (int.TryParse(s, out var i)) {
      Console.WriteLine(new string('*', i));
     }
    else {
      Console.WriteLine("Cloudy - no!"); }
}
```

Here **`i`** **is only used within the if-statement that defines it.**

7.14 Passing Arguments: Pass-by-Value vs. Pass-by-Reference (Cont.)

It is common to want to **return more than one value from a method**. The options available today are less than optimal:

✓`out` *parameters*: Use is clunky (even with the improvements described above), and **they don't work with `async` methods**.

✓**`System.Tuple<...>`** *return types*: Verbose to use and require an allocation of a tuple object.

✓*Custom-built transport type for every method*: A lot of code overhead for a type whose purpose is just to temporarily group a few values.

✓*Anonymous types returned through a* **`dynamic`** *return type*: High performance overhead and no static type checking.

7.14 Passing Arguments: Pass-by-Value vs. Pass-by-Reference (Cont.)

To do better at this, C# 7.0 adds ***tuple types*** and ***tuple literals***:

```
(string, string, string) LookupName(long id) // tuple
return type
{
    // retrieve first, middle and last from data storage
    return (first, middle, last); // tuple literal
}
```

The method now effectively returns **three strings, wrapped up as elements in a tuple value**.

The caller of the method will now receive a tuple, and can **access the elements individually**:

```
var names = LookupName(id);
Console.WriteLine($"found {names.Item1} {names.Item3}.");
```

7.14 Passing Arguments: Pass-by-Value vs. Pass-by-Reference (Cont.)

```
class Program
{   // Compile with     with C# 4.7.2
        0 references
        static void Main(string[] args)
        {
            var (sum, count) = myMethod(2, 2);
            Console.Write("{0}{1}", sum, count);


        }
        1 reference
        public static (int sum, int count) myMethod(int a, int b)
        {
            int s = a, c = b;
            return (s, c);
        }
```

7.15a Optional Parameters

- As of C# 2010 methods can have optional parameters that allow the calling method to vary the arguments to pass. An optional parameter **specifies the default value** that is assigned to the parameter, if the optional argument is omitted.
- *All optional parameters* ***must be placed to the right*** *of the method's non- optional parameters* i.e. at the end of the parameter list.

Common Programming Error 7.11

Declaring a non-optional parameter to the right of an optional one is a compilation error.

7.15 Optional Parameters

- When parameter has a default value, the caller has the option of passing that particular argument.
- Example:

Optional parameter.

```
public int Power(int baseValue, int expValue = 2){}
```

- Consider the following calls:

```
Power(); //syntax error, minimum one arg expected
Power(10); //valid,baseValue = 10, expValue = 2
Power(10,3); //valid,baseValue = 10, expValue = 3
```

```
// Fig. 7.12: Power.vb
// Optional argument demonstration with method Power.
using System;

class CalculatePowers
{
   // call Power with and without optional arguments
   public static void Main( string[] args )
   {
      Console.WriteLine( "Power(10) = {0}", Power( 10 ) ) ;
      Console.WriteLine( "Power(2, 10) = {0}", Power( 2, 10 ) );
   } // end Main

   // use iteration to calculate power
   public int Power( int baseValue, int exponentValue = 2 )
   {
      int result = 1; // initialize total

      for ( int i = 1; i <= exponentValue; i++ )
         result *= baseValue;

      return result;
   } // end method Power
} // end class CalculatePowers
```

Fig. 7.12 | Optional argument demonstration with method Power.

```
Power(10) = 100
Power(2, 10) = 1024
```

Fig. 7.12 | Optional argument demonstration with method `Power`.

7.15 Optional Parameters

- **Not all types of parameters** can be used as optional parameters.
  - You can use value types as optional parameters as long as the default value is determinable at compile time.
  - You can only use a reference type as an optional parameter, if the default value is null.

Data Types	Parameter Types: Value	ref	out	params
Value Type	Yes	No	No	No
Reference Type	Only null default	No	No	No

7.16 Named Parameters

- Normally, when **calling a method that has optional parameters**, the argument values in order are assigned to the parameters *from left to right* in the parameter list.
- Consider the **`Time`** class that stores the time of the day as the **`hour`** (`0-23`), **`minute`** (`0- 59`) and **`second`** (`o- 59`). Such a class might have a **`SetTime`** method with optional parameters

```
public int SetTime(int hour = 0, int minute = 0, int
second = 0){}
```

7.16 Named Parameters

- Starting with C# 4.0, you can list the actual parameters in your method invocation in any order, as long as you explicitly specify the names of the parameters. The details are the following:
  - Nothing changes in the declaration of the method. The formal parameters already have names.
  - In the **method invocation**, however, you **use the formal parameter name, followed by a colon,** in front of the actual parameter value or expression, as shown in the following method invocation

7.16 Named Parameters

- Assuming that we have an instance t of class Time

```
t.SetTime(); // sets the time 12:00:00 AM
t.SetTime(12); // sets the time 12:00:00 PM
t.SetTime(12, 30); // sets the time 12:30:00 PM
t.SetTime(12, 30, 45); // sets the time 12:30:45PM
```

7.16 Named Parameters

- What, if you wanted

```
t.SetTime(12,, 45);//to set  the time 12:00:45PM, wrong
```

- Results in compilation error
- The correct syntax using named parameters is

```
t.SetTime(hour:12,second:45);//correct
//also correct
t.SetTime(hour:12,second:40 + 5);
```

7.17 Local methods

- Sometimes a helper function only makes sense inside of a single method that uses it. You can now declare such functions inside other function bodies as a *local method*:

7.17 Local methods

```
public int Fibonacci(int x)
{
    if (x < 0) throw new ArgumentException("Less negativity please!", nameof(x));
    return Fib(x).current;

    (int current, int previous) Fib(int i)
    {
        if (i == 0) return (1, 0);
        var (p, pp) = Fib(i - 1);
        return (p + pp, p);
    }
}
```