The set of namespaces available in the .NET Framework Class Library is quite large. Besides those summarized in Fig. 3.4, the .NET Framework Class Library contains namespaces for complex graphics, advanced graphical user interfaces, printing, advanced networking, security, database processing, multimedia, accessibility (for people with dis- abilities) and many other capabilities—over 100 namespaces in all.

3.7 Case Study: Random-Number Generation

In this and the next section, we develop a nicely structured game-playing application with multiple methods. The application uses most of the control statements presented thus far in the book and introduces several new C# programming concepts.

There is something in the air of a casino that invigorates people—from the high rollers at the plush mahogany-and-felt craps tables to the quarter poppers at the one-armed bandits. It's the element of chance, the possibility that luck will convert a pocketful of money into a mountain of wealth. The element of chance can be introduced in an application via an object of class Random (of namespace System). Objects of class Random can produce random byte, int and double values. In the next several examples, we use objects of class Random to produce random numbers.

A new random-number generator object can be created as follows:

```
Random randomNumbers = new Random();
```

The random-number generator object can then be used to generate random byte, int and double values—we discuss only random int values here.Consider the following statement:

```
int randomValue = randomNumbers.Next();
```

Method Next of class Random generates a random int value from 0 to +2,147,483,646, in- clusive. If the Next method truly produces values at random, then every value in that range should have an equal chance (or probability) of being chosen each time method Next is called. 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 (which, of course, changes constantly) to seed the random-number generator such that each execution of an application yields a different sequence of random values.

The range of values produced directly by method Next often differs from the range of values required in a particular C# application. For example, an application that simulates coin tossing might require only 0 for "heads" and 1 for "tails." An application that simulates the rolling of a six-sided die might require random integers in the range 1–6. A video game that randomly predicts the next type of spaceship (out of four possibilities) that will fly across the horizon might require random integers in the range 1–4. For cases like these, class Random provides other versions of method Next. One receives an int argument and returns a value from 0 up to, but not including, the argument's value. For example, you might use the statement

```
int randomValue = randomNumbers.Next( 6 );
```

which returns 0, 1, 2, 3, 4 or 5. The argument 6—called the scaling factor—represents the number of unique values that Next should produce (in this case, six—0, 1, 2, 3, 4 and 5). This manipulation is called scaling the range of values produced by Random method Next.

Suppose we wanted to simulate a six-sided die that has the numbers 1–6 on its faces, not 0–5. Scaling the range of values alone is not enough. So we shift the range of numbers produced. We could do this by adding a shifting value—in this case 1—to the result of method Next, as in

```
face = 1 + randomNumbers.Next( 6 );
```

The shifting value (1) specifies the first value in the desired set of random integers. The preceding statement assigns to face a random integer in the range 1–6.

The third alternative of method Next provides a more intuitive way to express both shifting and scaling. This method receives two int arguments

and returns a value from the first argument's value up to, but not including, the second argument's value. We could use this method to write a statement equivalent to our previous statement, as in

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

Rolling a Six-Sided Die

To demonstrate random numbers, let's develop an application that simulates 20 rolls of a six-sided die and displays each roll's value. Figure 3.5 shows two sample outputs, which confirm that the results of the preceding calculation are integers in the range 1–6 and that each run of the application can produce a different sequence of random numbers. The using directive (line 3) enables us to use class Random without fully qualifying its name. Line 9 creates the Random object randomNumbers to produce random values. Line 16 executes 20 times in a loop to roll the die. The if statement (lines 21–22) starts a new line of output after every five numbers, so the results can be presented on multiple lines.

```
1  // Fig. 3.5: RandomIntegers.cs
2  // Shifted and scaled random integers.
3  using System;
4
5  public class RandomIntegers
6  {
7     public static void Main( string[] args )
8     {
9        Random randomNumbers = new Random(); // random-number generator
10       int face; // stores each random integer generated
11
12       // loop 20 times
13       for ( int counter = 1; counter <= 20; counter++ )
14       {
15          // pick random integer from 1 to 6
16          face = randomNumbers.Next( 1, 7 );
17
18          Console.Write( "{0}  ", face ); // display generated value
19
20          // if counter is divisible by 5, start a new line of output
21          if ( counter % 5 == 0 )
22             Console.WriteLine();
23       } // end for
24    } // end Main
25 } // 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
```

Fig. 3.5 | Shifted and scaled random integers.

Rolling a Six-Sided Die 6000 Times

To show that the numbers produced by Next occur with approximately equal likelihood, let's simulate 6000 rolls of a die (Fig. 3.6). Each integer from 1 to 6 should appear approximately 1000 times.

```
1  // Fig. 3.6: RollDie.cs
2  // Roll a six-sided die 6000 times.
3  using System;
4
5  public class RollDie
6  {
7     public static void Main( string[] args )
8     {
9        Random randomNumbers = new Random(); // random-number generator
10
11       int frequency1 = 0; // count of 1s rolled
12       int frequency2 = 0; // count of 2s rolled
13       int frequency3 = 0; // count of 3s rolled
14       int frequency4 = 0; // count of 4s rolled
15       int frequency5 = 0; // count of 5s rolled
16       int frequency6 = 0; // count of 6s rolled
17
18       int face; // stores most recently rolled value
19
20       // summarize results of 6000 rolls of a die
21       for ( int roll = 1; roll <= 6000; roll++ )
22       {
23          face = randomNumbers.Next( 1, 7 ); // number from 1 to 6
24
25          // determine roll value 1-6 and increment appropriate counter
26          switch ( face )
27          {
28             case 1:
29                ++frequency1; // increment the 1s counter
30                break;
31             case 2:
32                ++frequency2; // increment the 2s counter
```

```
33                  break;
34               case 3:
35                  ++frequency3; // increment the 3s counter
36                  break;
37               case 4:
38                  ++frequency4; // increment the 4s counter
39                  break;
40               case 5:
41                  ++frequency5; // increment the 5s counter
42                  break;
43               case 6:
44                  ++frequency6; // increment the 6s counter
45                  break;
46            } // end switch
47         } // end for
48
49         Console.WriteLine( "Face\tFrequency" ); // output headers
50         Console.WriteLine(
51            "1\t{0}\n2\t{1}\n3\t{2}\n4\t{3}\n5\t{4}\n6\t{5}",frequency1,
52            frequency2, frequency3, frequency4, frequency5, frequency6 );
53      } // end Main
54   } // end class RollDie
```

```
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. 3.6 | Roll a six-sided die 6000 times.

As the sample outputs show, the values produced by Next enable the application to realistically simulate rolling a six-sided die. We used nested control statements (the switch is nested inside the for) to determine the number of times each side of the die occurred. Lines 21–47 iterate 6000 times. Line 23 produces a random value from 1 to 6. This face value is then used as the switch expression (line 26) in the switch statement (lines 26–46). Based on the face value, the switch

statement increments one of the six counter variables during each iteration of the loop. The switch statement has no default label because we have a case label for every possible die value that the expression in line 23 can produce. Run the application several times and observe the results. You'll see that every time you execute this application, it produces different results.

3.9 Method Overloading

Methods of the same name can be declared in the same class, as long as they have different sets of parameters (determined by the number, types and order of the parameters). This is called method overloading. 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 commonly used to create several methods with the same name that perform the same or similar tasks, but on different types or different numbers of arguments. For example, Math methods Min and Max (summarized in Section 3.3) are overloaded with 11 versions. These find the minimum and maximum, respectively, of two values of each of the 11 numeric simple types. Our next example demonstrates declaring and invoking overloaded methods. You'll see examples of overloaded constructors in Chapter 10.

Declaring Overloaded Methods

In class MethodOverload (Fig. 3.7), we include two overloaded versions of a method called Square—one that calculates the square of an int (and returns an int) and one that calculates the square of a double (and returns a double). Although these methods have the same name and similar parameter lists and bodies, you can think of them simply as different methods. It may help to think of the method names as "Square of int" and "Square of double," respectively.

```
1  // Fig. 3.7: MethodOverload.cs
2  // Overloaded method declarations.
3  using System;
4
5  public class MethodOverload
6  {
7     // test overloaded square methods
8     public static void Main( string[] args )
9     {
10       Console.WriteLine( "Square of integer 7 is {0}", Square( 7 ) );
11       Console.WriteLine( "Square of double 7.5 is {0}", Square( 7.5 ) );
12    } // end Main
13
14    // square method with int argument
15    public static int Square( int intValue )
16    {
17       Console.WriteLine( "Called square with int argument: {0}",
18          intValue );
19       return intValue * intValue;
20    } // end method Square with int argument
21
22    // square method with double argument
23    public static double Square( double doubleValue )
24    {
25       Console.WriteLine( "Called square with double argument: {0}",
26          doubleValue );
27       return doubleValue * doubleValue;
28    } // end method Square with double argument
29 } // end class MethodOverload
```

```
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. 3.7 | Overloaded method declarations.

Line 10 in Main invokes method Square with the argument 7. Literal integer values are treated as type int, so the method call in line 10 invokes the version of Square at lines 15–20 that specifies an int parameter. Similarly, line 11 invokes method Square with the argument 7.5. Literal real-number values are treated as type double, so the method call in line 11 invokes the version of Square at lines 23–28 that specifies a double parameter. Each method first outputs a line of text to prove that the proper method was called in each case.

Notice that the overloaded methods in Fig. 3.7 perform the same calculation, but with two different types. C#'s generics feature provides a mechanism for writing a single "generic method" that can perform the same tasks as an entire set of overloaded methods.

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. The signature also includes the way those parameters are passed, which can be modified by the ***ref*** *and* ***out*** *keywords. If the compiler looked only at method names during compilation, the code in Fig. 3.7 would be ambiguous—the compiler would not know how to distinguish between the Square methods (lines 15–20 and 23–28). Internally, the compiler uses signatures to determine whether a class's methods are unique in that class.*

For example, in Fig. 3.7, the compiler will use the method signatures to distinguish between the "Square of int" method (the Square method that specifies an int parameter) and the "Square of double" method (the Square method that specifies a double parameter).

If Method1's declaration begins as

```
void Method1( int a, float b )
```

then that method will have a different signature than the method declared beginning with

```
void Method1( float a, int b )
```

The order of the parameter types is important—the compiler considers the preceding two Method1 headers to be distinct.

Return Types of Overloaded Methods

In discussing the logical names of methods used by the compiler, we did not mention the return types of the methods. This is because

method calls cannot be distinguished by return type. The application in Fig. 3.8 illustrates the compiler errors generated when two methods have the same signature but different return types. Overloaded methods can have the same or different return types if the methods have different parameter lists. Also, overloaded methods need not have the same number of parameters.

```
1  // Fig. 3.8: MethodOverload.cs
2  // Overloaded methods with identical signatures
3  // cause compilation errors, even if return types are different.
4  public class MethodOverloadError
5  {
6     // declaration of method Square with int argument
7     public int Square( int x )
8     {
9        return x * x;
10    } // end method Square
11
12    // second declaration of method Square with int argument
13    // causes compilation error even though return types are different
14    public double  Square( int y )
15    {
16       return y * y;
17    } // end method Square
18 } // end class MethodOverloadError
```

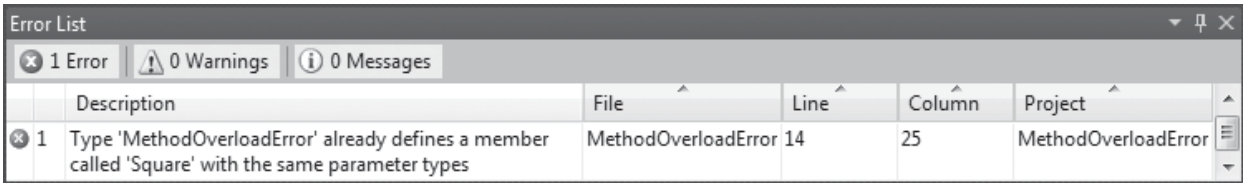

Fig. 3.8 | Overloaded methods with identical signatures cause compilation errors, even if return types are different.

3.10 Optional Parameters

As of Visual C# 2010, methods can have optional parameters that allow the calling method to vary the number of arguments to pass. An optional parameter specifies a default value that's assigned to the parameter if the optional argument is omitted. You can create methods with one or more optional parameters. All optional parameters must be placed to the right of the method's non-optional parameters—that is, at the end of the parameter list.

When a parameter has a default value, the caller has the option of passing that particular argument. For example, the method header

```
public int Power( int baseValue, int exponentValue = 2 )
```

specifies an optional second parameter. A call to Power must pass at least an argument for the parameter baseValue, or a compilation error occurs. Optionally, a second argument (for the exponentValue parameter) can be passed to Power. Consider the following calls to Power:

```
Power()
Power(10)
Power(10, 3)
```

The first call generates a compilation error because this method requires a minimum of one argument. The second call is valid because one argument (10) is being passed—the optional exponentValue is not specified in the method call. The last call is also valid—10 is passed as the required argument and 3 is passed as the optional argument.

In the call that passes only one argument (10), parameter exponentValue defaults to 2, which is the default value specified in the method's header. Each optional parameter must specify a default value by using an equal (=) sign followed by the value. For example, the header for Power sets 2 as exponentValue's default value. Figure 3.9 demonstrates an optional parameter. The program calculates the result of raising a base value to an exponent. Method Power (Fig. 3.9, lines 15–23) specifies that its second parameter is optional. In method DisplayPowers, lines 10–11 of Fig. 3.9 call method Power. Line 10 calls the method without the optional second argument. In this case, the compiler provides the second argument, 2, using the default value of the optional argument, which is not visible to you in the call.

```
1  // Fig. 3.9: Power.vb
2  // Optional argument demonstration with method Power.
3  using System;
4
5  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 otal

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

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

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

Fig. 3.9 | Optional argument demonstration with method `Power`.

3.11 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 a Time class that stores the time of day in 24-hour clock format as int values representing the hour (0–23), minute (0–59) and second (0–59). Such a class might provide a SetTime method with optional parameters like

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

In the preceding method header, all of three of SetTime's parameters are optional. Assuming that we have a Time object named t, we can call SetTime as follows:

```
SetTime(); // sets the time to 12:00:00 AM
SetTime( 12 ); // sets the time to 12:00:00 PM
SetTime( 12, 30 ); // sets the time to 12:30:00  PM
SetTime( 12, 30, 22 ); // sets the time to 12:30:22 PM
```

In the first call, no arguments are specified, so the compiler assigns 0 to each parameter. In the second call, the compiler assigns the

argument, 12, to the first parameter, hour, and assigns default values of 0 to the minute and second parameters. In the third call, the compiler assigns the two arguments, 12 and 30, to the parameters hour and minute, respectively, and assigns the default value 0 to the parameter second. In the last call, the compiler assigns the three arguments, 12, 30 and 22, to the parameters hour, minute and second, respectively. What if you wanted to specify only arguments for the hour and second? You might think that you could call the method as follows:

```
SetTime( 12, , 22 ); // COMPILATION ERROR
```

Unlike some programming languages, C# doesn't allow you to skip an argument as shown in the preceding statement. However, Visual C# 2010 provides a new feature called named parameters, which enable you to call methods that receive optional parameters by providing only the optional arguments you wish to specify. To do so, you explicitly specify the parameter's name and value—separated by a colon (:)—in the argument list of the method call. For example, the preceding statement can be implemented in Visual C# 2010 as follows:

```
SetTime( hour: 12, second: 22 ); // sets the time to 12:00:22
```

In this case, the compiler assigns parameter hour the argument 12 and parameter second the argument 22. The parameter minute is not specified, so the compiler assigns it the de- fault value 0. It's also possible to specify the arguments out of order when using named parameters. The arguments for the required parameters must always be supplied.