5

6

Control Statements: Part 1 & 2

5.4 Control Structures

- Normally, statements are executed one after the other in **sequential execution**.
- Various C# statements enable you to specify the next statement to execute. This is called **transfer of control**.
- Structured applications are clearer, easier to debug and modify, and more likely to be bug free.

5.4 Control Structures (Cont)

- **Single-entry/single-exit control statements** make it easy to build applications.
- Control statements are "attached" to one another by connecting the exit point of one to the entry point of the next.
- This procedure is called **control-statement stacking**.
- **Control-statement nesting** allows a control statement to appear inside another control statement.

5.6 if…else Double-Selection Statement (Cont.)

- if ... else works exactly how you'd expect

- The **conditional operator** (**?:**) can be used in place of an if…else statement.

```
Console.WriteLine( grade >= 60 ? "Passed" :
  "Failed" );
```

  - The first operand is a **boolean** expression that evaluates to **true** or **false**.
  - The second operand is the value if the expression is true
  - The third operand is the value if the expression is false. `

Repetition Statements, etc

- While
  - Counter controlled
  - Sentinel controlled
- For

```
for ( int j = x; j <= 4 * x * y; j += y / x ){...}
for ( int number = 2; number <= 20; total += number, number += 2 );
```

- Top-down, step-wise refinement
- Nested vs Stacked Control Structures

6.4 In-class Example

- Consider the following problem:

A person invests $1,000 in a savings account yielding 5% interest, compounded yearly. Calculate and display the amount of money in the account at the end of each year for 10 years.

$$a = p\,(1 + r\,)^{n}$$

p is the original amount invested (i.e., the principal)
r is the annual interest rate (use 0.05 for 5%)
n is the number of years
a is the amount on deposit at the end of the *n*th year.

- The application shown in Fig. 6.6 uses a loop that performs the calculation for each of the 10 years the money remains on deposit.

Interest.cs

(1 of 2)

```
// Fig. 6.6: Interest.cs
// Compound-interest calculations with for.
using System;

public class Interest
{
   public static void Main( string[] args )
   {
      decimal amount; // amount on deposit at end of each year
      decimal principal = 1000; // initial amount before interest
      double rate = 0.05; // interest rate

      // display headers
      Console.WriteLine( "Year{0,20}", "Amount on deposit" );

      // calculate amount on deposit for each of ten years
```

Format item `{0,20}` indicates that the value output should be displayed with a **field width** of 20.

Fig. 6.6 | Compound-interest calculations with `for`. (Part 1 of 2.)

Interest.cs

(2 of 2)

```
      for ( int year = 1; year <= 10; year++ )
      {
         // calculate new amount for specified year
         amount = principal *
            ( ( decimal ) Math.Pow( 1.0 + rate, year ) );

         // display the year and the amount
         Console.WriteLine( "{0,4}{1,20:C}", year, amount );
      } // end for
   } // end Main
} // end class Interest
```

The `for` statement executes 10 times, varying `year` from `1` to `10` in increments of `1`.

```
Year   Amount on deposit
   1           $1,050.00
   2           $1,102.50
   3           $1,157.63
   4           $1,215.51
   5           $1,276.28
   6           $1,340.10
   7           $1,407.10
   8           $1,477.46
   9           $1,551.33
  10           $1,628.89
```

Fig. 6.6 | Compound-interest calculations with `for`. (Part 2 of 2.)

6.4 Examples Using the for Statement (Cont.)

- Format item {0,20} indicates that the value output should be displayed with a **field width** of 20.
  - To indicate that output should be **left justified**, use a negative field width.

5.11 Compound Assignment Operators (Cont.)

- Figure 5.14 explains the arithmetic compound assignment operators.

Assignment operator	Sample expression	Explanation	Assigns
Assume: int c = 3, d = 5, e = 4, f = 6, g = 12;			
+=	c += 7	c = c + 7	10 to c
-=	d -= 4	d = d - 4	1 to d
*=	e *= 5	e = e * 5	20 to e
/=	f /= 3	f = f / 3	2 to f
%=	g %= 9	g = g % 9	3 to g

Fig. 5.14 | Arithmetic compound assignment operators.

5.12 Increment and Decrement Operators

- C# provides operators for adding or subtracting 1 from a numeric variable (Fig. 5.15).
  - The unary **increment operator**, ++
  - The unary **decrement operator**, **--**.

Operator	Called	Sample expression	Explanation
++	prefix increment	++a	Increments a by 1, then uses the new value of a in the expression.
++	postfix increment	a++	Uses the current value of a, then increments a by 1.
--	prefix decrement	--b	Decrements b by 1, then uses the new value of b.
--	postfix decrement	b--	Uses the current value of b, then decrements b by 1.

Fig. 5.15 | Increment and decrement operators.

5.13 Simple Types

- The table in Appendix B, Simple Types, lists the 13 **simple types** in C#.
- C# requires all variables to have a type.
- Instance variables of types `char`, `byte`, `sbyte`, `short`, `ushort`, `int`, `uint`, `long`, `ulong`, `float`, `double`, and `decimal` are all given the value `0` by default.
- Instance variables of type `bool` are given the value `false` by default.

6.6 `switch` Multiple-Selection Statement

- The **`switch` multiple-selection** statement performs different actions based on the value of an expression.
- Each action is associated with the value of a **constant integral expression** or a **constant string expression** that the expression may assume.

6.8 Logical Operators

- The **&&** (**conditional AND**) operator works as follows:

```
if ( gender == "F" && age >= 65 )
   ++seniorFemales;
```

- The combined condition is true if and only if *both* simple conditions are true.

- The **||** (**conditional OR**) operator, as in the following application segment:

```
if ( ( semesterAverage >= 90 ) || ( finalExam >= 90 ) )
    Console.WriteLine ( "Student grade is A" );
```

Logical Negation (!) Operator

- The **!** (**logical negation**) operator enables you to “reverse” the meaning of a condition.

```
if ( ! ( grade == sentinelValue ) )
   Console.WriteLine( "The next grade is {0}", grade );
```

6.8 Logical Operators (Cont.)

Boolean Logical AND (&) and Boolean Logical OR (|) Operators

- The **boolean logical AND** (&) and **boolean logical inclusive OR** (|) operators do not perform short-circuit evaluation.
- This is useful if the right operand has a required **side effect**. For example:

```
( birthday == true ) | ( ++age >= 65 )
```

- This ensures that the condition `++age >= 65` will be evaluated.

Error-Prevention Tip 6.5

For clarity, avoid expressions with side effects in conditions. The side effects may look clever, but they can make it harder to understand code and can lead to subtle logic errors.

6.8 Logical Operators (Cont.)

Boolean Logical Exclusive OR (^)

- A complex condition containing the **boolean logical exclusive OR** (^) operator (also called the **logical XOR operator**) is `true` *if and only if one of its operands is* `true` *and the other is* `false`.
- Figure 6.16 is a truth table for the boolean logical exclusive OR operator (^).

expression1	expression2	expression1 ^ expression2
false	false	false
false	true	true
true	false	true
true	true	false

Fig. 6.16 | ^ (boolean logical exclusive OR) operator truth table.

6.8 Logical Operators (Cont.)

- Figure 6.19 shows the C# operators from highest precedence to lowest.

Operators	Associativity	Type
. new ++ *(postfix)* --*(postfix)*	left to right	highest precedence
++ -- + - ! *(type)*	right to left	unary prefix
* / %	left to right	multiplicative
+ -	left to right	additive
< <= > >=	left to right	relational
== !=	left to right	equality

Fig. 6.19 | Precedence/associativity of the operators discussed so far. (Part 1 of 2.)

7

Methods: A Deeper Look

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**.
- 2 Rationales for the use of methods
  - Manageability of design
  - Software Reuse
    - "Good programmers code, great programmers reuse."

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).

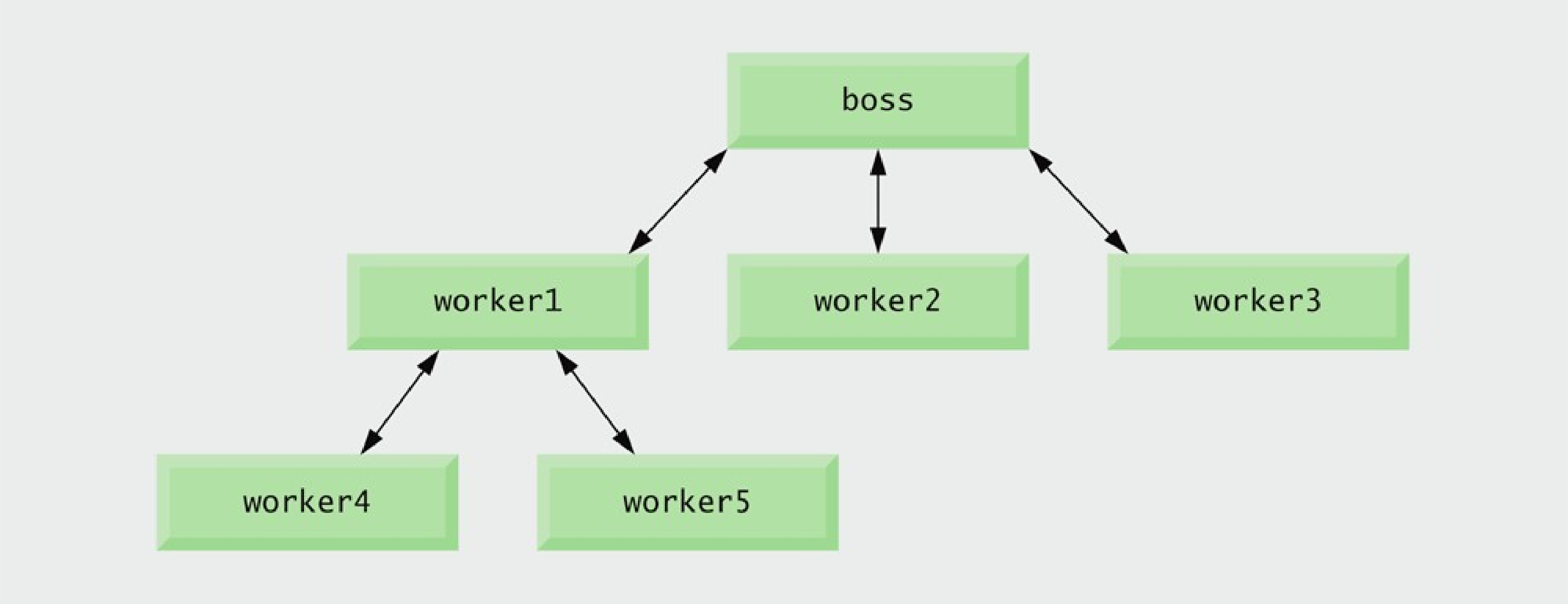


Fig. 7.1 | Hierarchical boss-method/worker-method relationship. (Part 1 of 2.)

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.
  - Static methods are different because they do not depend on any particular object instance
- 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 (from System namespace) 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.

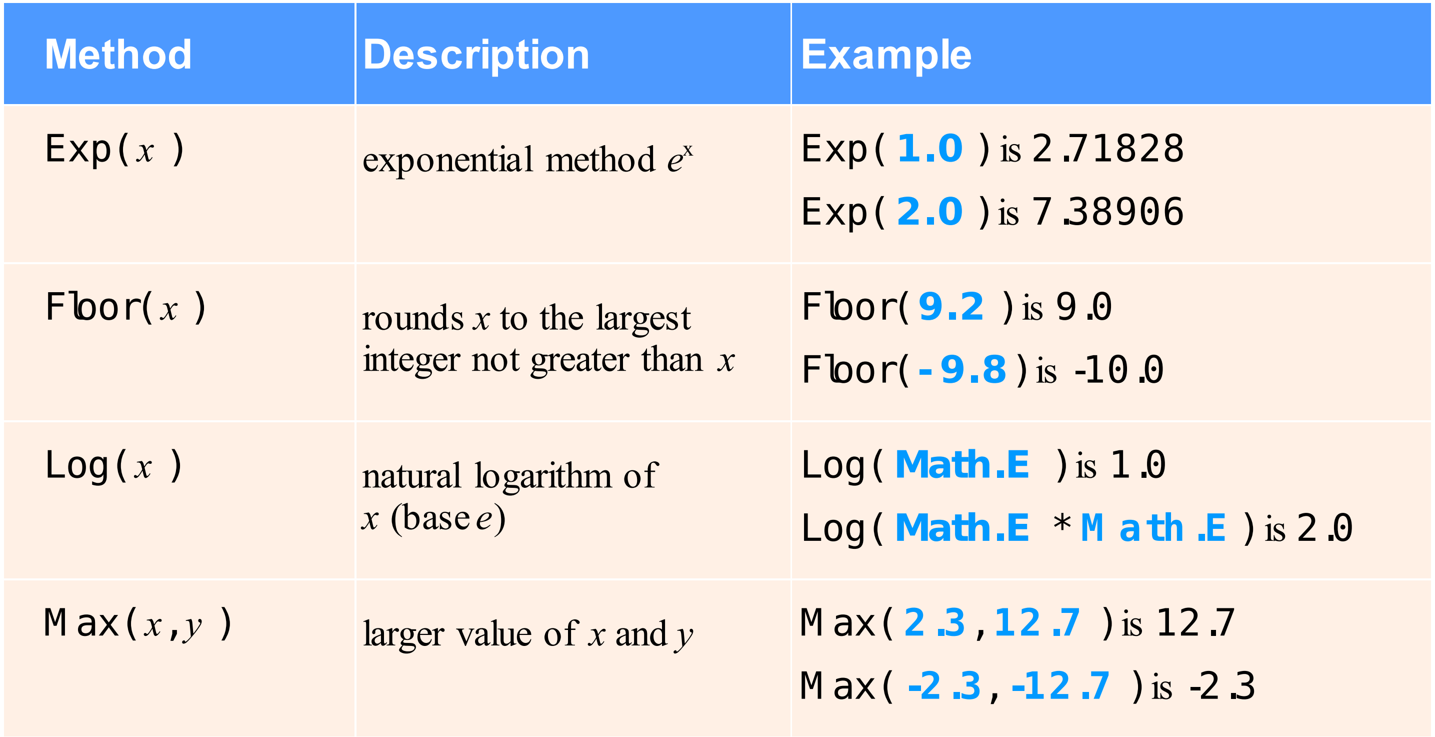
7.3 static Methods, static Variables and Class Math (Cont.)

- 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
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.
  - Pi ~3.14 (ratio of circumference to diameter)
  - E ~2.72 (base value for natural log)
- 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.

- 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.

Outline

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

`MaximumFinder.cs`

(1 of 3)

```
// Fig٧.٣: 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 number١ = Convert.ToDouble( Console.ReadLine() );
     double number٢ = Convert.ToDouble( Console.ReadLine() );
     double number٣ = 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.)

MaximumFinder.cs

(2 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

(1 of 2)

```
// Fig7.4: 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 `Determine-Maximum` method to produce the application’s output

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

Outline

```
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
```

MaximumFinder Test.cs

(2 of 2)

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.)

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

7.6 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.)

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

MaximumFinderTest.cs

(1 of 2)

```
// Fig7.4: 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 `Determine-Maximum` method to produce the application’s output

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

7.8 The .NET Framework Class Library (Cont.)

- Some key Framework Class Library namespaces are described in Fig. 7.6.

Namespace	Description
`System.Windows.Forms`	Contains the classes required to create and manipulate GUIs.

`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 | Framework Class Library namespaces (a subset). (Part 1 of 2.)

7.8 The .NET Framework Class Library (Cont.)

Namespace	Description
`System.IO`	Contains classes that enable programs to input and output data.
`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 | Framework Class Library namespaces (a subset). (Part 2 of 2.)

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.

RandomIntegers.cs

(1 of 2)

```
// Fig7.7: 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.)

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.

7.10 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`

(1 of 4)

```
// Fig7.9: 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.)

`Craps.cs`

(3 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.)

```
      // display won or lost message
      if ( gameStatus == Status.WON )
         Console.WriteLine( "Player wins" );
      else
         Console.WriteLine( "Player loses" );
   } // end method Play

   // roll dice, calculate sum and display results
   public int RollDice()
   {
      // pick random die values
      int die1 = randomNumbers.Next( 1, 7 ); // first die roll
      int die2 = randomNumbers.Next( 1, 7 ); // second die roll

      int sum = die1 + die2; // sum of die values

      // display results of this roll
      Console.WriteLine( "Player rolled {0} + {1} = {2}",
         die1, die2, sum );
      return sum; // return sum of dice
   } // end method RollDice
} // 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.)

- 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).

CrapsTest.cs

(1 of 2)

```
// Fig7.10: 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  // Fig7.11: 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.)

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)

```
40         ++x; // modifies this method's local variable x
41         Console.WriteLine(
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

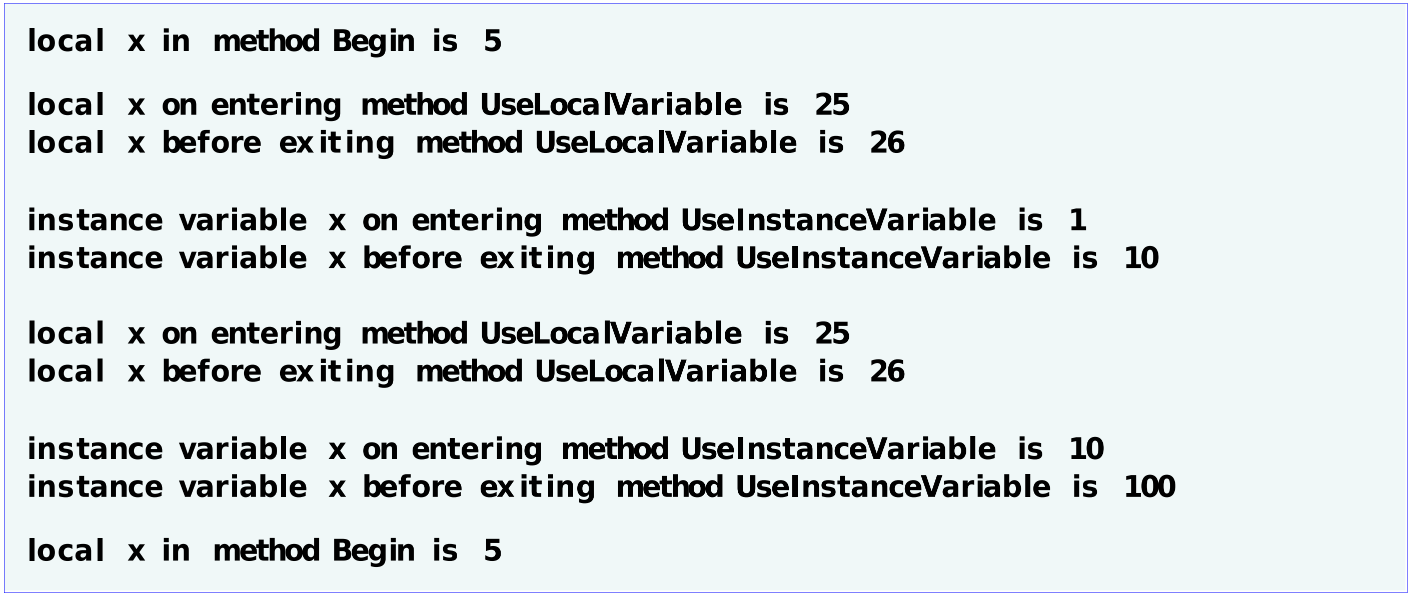
(1 of 2)

```
// Fig7.12: 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.)

ScopeTest.cs

(2 of 2)



```
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.)