

7

Methods: A Deeper Look

© 2006 Pearson Education, Inc. All rights reserved.

OBJECTIVES

In this chapter you will learn:

- How static methods and variables are associated with an entire class rather than specific instances of the class.
- How to use random-number generation to implement game-playing applications.
- To understand how the visibility of declarations is limited to specific regions of applications.
- What method overloading is and how to create overloaded methods.
- The differences between passing method arguments by value and by reference.

© 2006 Pearson Education, Inc. All rights reserved.

7.2 Packaging Code in C#

• Three common ways of packaging code:

- Methods
- Classes
- Namespaces

• Benefits of using methods:

- Divide and conquer; small and simple pieces
- Able to reuse software
- Easier to maintain and debug
- Hide details of implementation

© 2006 Pearson Education, Inc. All rights reserved.

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.

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

© 2006 Pearson Education, Inc. All rights reserved.

7.3 static Methods, static Variables and Class Math

• static method (or class method)

- Applies to the class as a whole instead of a specific object of the class
- Call a static method by using the method call: `ClassName.methodName(arguments)`
- All methods of the `Math` class are static
 - Example: `Math.Sqrt(900.0)`

© 2006 Pearson Education, Inc. All rights reserved.

Method	Description	Example
<code>Abs(x)</code>	absolute value of x	<code>Abs(23.7)</code> is 23.7 <code>Abs(0.0)</code> is 0.0 <code>Abs(-23.7)</code> is 23.7
<code>Ceiling(x)</code>	rounds x to the smallest integer not less than x	<code>Ceiling(9.5)</code> is 10.0 <code>Ceiling(-9.8)</code> is -9.0
<code>Cos(x)</code>	trigonometric cosine of x (x in radians)	<code>Cos(0.0)</code> is 1.0
<code>Exp(x)</code>	exponential method e^x	<code>Exp(2.0)</code> is 7.38906 <code>Exp(9.2)</code> is 9.0
<code>Floor(x)</code>	rounds x to the largest integer not greater than x	<code>Floor(-9.8)</code> is -10.0 <code>Floor(Math.E)</code> is 1.0
<code>Log(x)</code>	natural logarithm of x (base e)	<code>Log(Math.E * Math.E)</code> is 2.0
<code>Max(x, y)</code>	larger value of x and y	<code>Max(0.5, 12.7)</code> is 12.7 <code>Max(-2.3, -12.7)</code> is -2.3
<code>Min(x, y)</code>	smaller value of x and y	<code>Min(0.5, 12.7)</code> is 0.5 <code>Min(-2.3, -12.7)</code> is -12.7
<code>Pow(x, y)</code>	x raised to the power y (i.e., x^y)	<code>Pow(2.0, 7.0)</code> is 128.0 <code>Pow(0.5, 0.5)</code> is 3.0
<code>Sin(x)</code>	trigonometric sine of x (x in radians)	<code>Sin(0.0)</code> is 0.0
<code>Sqrt(x)</code>	square root of x	<code>Sqrt(900.0)</code> is 30.0
<code>Tan(x)</code>	trigonometric tangent of x (x in radians)	<code>Tan(0.0)</code> is 0.0

Fig. 7.2 | Math class methods.

© 2006 Pearson Education, Inc. All rights reserved.

7.5 Notes on Declaring and Using Methods

- Three ways to call a method:
 - Use a method name by itself to call another method of the same class
 - Use a variable containing a reference to an object, followed by a dot (.) and the method name to call a method of the referenced object
 - Use the class name and a dot (.) to call a static method of a class
- static methods cannot call non-static methods of the same class directly

© 2006 Pearson Education, Inc. All rights reserved.

7.5 Notes on Declaring and Using Methods (Cont.)

- Three ways to return control to the calling statement:
 - If method does not return a result:
 - Program flow reaches the method-ending right brace
 - Program executes the return statement
 - If method does return a result:
 - Program executes the statement `return expression;`
 - `expression` is first evaluated and then its value is returned to the caller

© 2006 Pearson Education, Inc. All rights reserved.

Common Programming Error 7.4

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

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

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

© 2006 Pearson Education, Inc. All rights reserved.

7.7 Argument Promotion and Casting

- Argument promotion
 - C# will promote a method call argument to match its corresponding method parameter according to the promotion rules
 - Values in an expression are promoted to the “highest” type in the expression (a temporary copy of the value is made)
 - Converting values to lower types results in a compilation error, unless the programmer explicitly forces the conversion to occur
 - Some information may be lost
 - Place the desired data type in parentheses before the value
 - Example: `(int) 4.5`
 - Result: 4

© 2006 Pearson Education, Inc. All rights reserved.

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

© 2006 Pearson Education, Inc. All rights reserved.

7.9 Case Study: Random-Number Generation

- Random-number generation
 - Class `Random`
 - Random's `Next` method generates a random `int` value in the range 0 to +2,147,483,646, inclusive
 - Scaling: With one `int` argument, `Next` returns a value from 0 up to, but not including, the argument's value
 - Shifting: With two `int` argument, `Next` returns a value from the lower `int` up to, but not including, the higher `int`
 - Equal likelihood
 - Class `Random` from `System` namespace
 - Is seeded with the current time of day to generate different sequences of numbers each time the program executes

© 2006 Pearson Education, Inc. All rights reserved.

13

Outline

```

1 // Fig. 7.7: 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.WriteLine( "{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

```

RandomIntegers.cs (1 of 2)

Create a Random object

Generate a random die roll

© 2006 Pearson Education, Inc. All rights reserved.

14

Outline

```

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

```

RandomIntegers.cs (2 of 2)

Two different sets of results containing integers in the range 1-6

© 2006 Pearson Education, Inc. All rights reserved.

15

Outline

```

1 // Fig. 7.8: 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    }
26 }

```

RollDie.cs (1 of 3)

Create a Random object

Declare frequency counters

Iterate 6000 times

Generate a random die roll

© 2006 Pearson Education, Inc. All rights reserved.

16

Outline

```

24 // determine roll value 1-6 and increment appropriate counter
25 switch ( face )
26 {
27     case 1:
28         frequency1++; // increment the 1s counter
29         break;
30     case 2:
31         frequency2++; // increment the 2s counter
32         break;
33     case 3:
34         frequency3++; // increment the 3s counter
35         break;
36     case 4:
37         frequency4++; // increment the 4s counter
38         break;
39     case 5:
40         frequency5++; // increment the 5s counter
41         break;
42     case 6:
43         frequency6++; // increment the 6s counter
44         break;
45 } // end switch
46 } // end for

```

RollDie.cs (2 of 3)

switch based on the die roll to determine frequency

© 2006 Pearson Education, Inc. All rights reserved.

17

Outline

```

48 Console.WriteLine( "Face\tFrequency" ); // output headers
49 Console.WriteLine( "\t{0}\t{1}\t{2}\t{3}\t{4}\t{5}\t{6}",
50     frequency1, frequency2, frequency3, frequency4,
51     frequency5, frequency6 );
52 } // end Main
53 // end class RollDie

```

RollDie.cs (3 of 3)

Display die roll frequencies

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	1022
6	1010

© 2006 Pearson Education, Inc. All rights reserved.

18

7.11 Scope of Declarations

- Basic scope rules
 - Scope of a parameter declaration is the body of the method in which appears
 - Scope of a local-variable declaration is from the point of declaration to the end of that block
 - Scope of a local-variable declaration in the initialization section of a `for` header is the rest of the `for` header and the body of the `for` statement
 - Scope of a method or field of a class is the entire body of the class
 - Shadowing: A local variable or parameter in a method that has the same name as a field of a class is hidden until the block terminates execution

© 2006 Pearson Education, Inc. All rights reserved.

19

Outline

Scope.cs (1 of 2)

```

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 = 5;
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 = 1; // method's local variable x hides instance variable x
15
16        Console.WriteLine("local x in method Begin is {0}", x);
17
18        // UseLocalVariable has its own local x
19        UseLocalVariable();
20
21        // UseInstanceVariable uses class Scope's instance variable x
22        UseInstanceVariable();
23
24        // UseLocalVariable reinitializes its own local x
25        UseLocalVariable();
26
27        // class Scope's instance variable x retains its value
28        UseInstanceVariable();
29    }
30 }

```

Shadows field x

Display value of local variable x

© 2006 Pearson Education, Inc. All rights reserved.

20

Outline

Scope.cs (2 of 2)

```

31 // end method Begin
32
33 // create and initialize local variable x during each call
34 public void UseLocalVariable()
35 {
36     int x = 10; // initialized each time UseLocalVariable is called
37
38     Console.WriteLine(
39         "local 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( "instance 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

```

Shadows field x

Display value of local variable x

Display value of field x

© 2006 Pearson Education, Inc. All rights reserved.

21

Outline

ScopeTest.cs

```

1 // Fig. 7.12: ScopeTest.cs
2 // Application to test class Scope.
3 public class ScopeTest
4 {
5     // application starting point
6     public static void Main( string[] args )
7     {
8         Scope testScope = new Scope();
9         testScope.Begin();
10    } // end Main
11 } // end class ScopeTest

```

```

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

```

© 2006 Pearson Education, Inc. All rights reserved.

22

7.12 Method Overloading

- Method Overloading
 - Multiple methods with the same name, but different types, number or order of parameters in their parameter lists
 - Compiler decides which method is being called by matching the method call's argument list to one of the overloaded methods' parameter lists
 - A method's name and number, type and order of its parameters form its signature
 - Differences in return type are irrelevant in method overloading
 - Overloaded methods can have different return types
 - Methods with different return types but the same signature cause a compilation error

© 2006 Pearson Education, Inc. All rights reserved.

23

Outline

MethodOverload.cs

```

1 // Fig. 7.13: MethodOverload.cs
2 // Overloaded method declarations.
3 using System;
4
5 public class MethodOverload
6 {
7     // test overloaded square methods
8     public void TestOverloadedMethods()
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 method TestOverloadedMethods
13
14    // square method with int argument
15    public 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 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

```

Correctly calls the "Square of int" method

Correctly calls the "Square of double" method

Declaring the "Square of int" method

Declaring the "Square of double" method

© 2006 Pearson Education, Inc. All rights reserved.

24

Outline

MethodOverloadTest.cs

```

1 // Fig. 7.14: MethodOverloadTest.cs
2 // Application to test class MethodOverload.
3 public class MethodOverloadTest
4 {
5     public static void Main( string[] args )
6     {
7         MethodOverload methodOverload = new MethodOverload();
8         methodOverload.TestOverloadedMethods();
9     } // end Main
10 } // 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

```

© 2006 Pearson Education, Inc. All rights reserved.

25

```

1 // Fig. 7.10: 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    }
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    }
18 } // end class MethodOverloadError
  
```

Outline

MethodOverload.cs

Same method signature

Compilation error

© 2006 Pearson Education, Inc. All rights reserved.

26

Common Programming Error 7.10

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

© 2006 Pearson Education, Inc. All rights reserved.

27

7.13 Recursion

- Recursion
 - A method that calls itself
 - Break the problem you want to solve into pieces
 - Recursive method consists of:
 - Base case(s)
 - Solving simple problem
 - Recursive call(s)
 - Also known as recursion step(s)
 - Normally includes a return statement

© 2006 Pearson Education, Inc. All rights reserved.

28

Fig. 7.16 | Recursive evaluation of 5!.

© 2006 Pearson Education, Inc. All rights reserved.

29

```

1 // Fig. 7.17: FactorialTest.cs
2 // Recursive Factorial method.
3 using System;
4
5 public class FactorialTest
6 {
7     public static void Main( string[] args )
8     {
9         // calculate the factorials of 0 through 10
10        for ( long counter = 0; counter <= 10; counter++ )
11            Console.WriteLine( "(0) = {1}",
12                counter, Factorial( counter ) );
13    } // end method Main
  
```

Outline

FactorialTest.cs

(1 of 2)

Call recursive method

© 2006 Pearson Education, Inc. All rights reserved.

30

```

15 // recursive declaration of method Factorial
16 public static long Factorial( long number )
17 {
18     // base case
19     if ( number <= 1 )
20         return 1;
21     // recursion step
22     else
23         return number * Factorial( number - 1 );
24 } // end method Factorial
25 } // end class FactorialTest
  
```

Outline

FactorialTest.cs

(2 of 2)

static method header declaration

Base case: number <= 1 (ends recursion)

Recursive call

```

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

© 2006 Pearson Education, Inc. All rights reserved.

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

- **Argument pass by value**
 - Copy of value is made and passed into the function
 - Changes to the copy does not affect the original value (unless it is a reference type)
- **Argument pass by reference**
 - Gives the method the ability to access and modify original variable
 - Objects always passed by reference
 - To pass value by reference:
 - **ref**: For initialized variables
 - **out**: For uninitialized variables

© 2006 Pearson Education, Inc. All rights reserved.

Software Engineering Observation 7.5

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

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

© 2006 Pearson Education, Inc. All rights reserved.

```

1 // Fig. 7.18: ReferenceAndOutputParameters.cs
2 // Reference, output and value parameters.
3 using System;
4
5 class ReferenceAndOutputParameters
6 {
7     // call methods with reference, output and value parameters
8     public void DemonstrateReferenceAndOutputParameters()
9     {
10         int y = 5; // initialize y to 5
11         int z; // declares z, but does not initialize it
12
13         // display original values of y and z
14         Console.WriteLine("Original value of y: {0}", y);
15         Console.WriteLine("Original value of z: uninitialized");
16
17         // pass y and z by reference
18         SquareRef(ref y); // must use keyword ref
19         SquareOut(out z); // must use keyword out
20
21         // display values of y and z after they are modified by
22         // methods SquareRef and SquareOut, respectively
23         Console.WriteLine("Value of y after SquareRef: {0}", y);
24         Console.WriteLine("Value of z after SquareOut: {0}", z);
25
26         // pass y and z by value
27         Square(y);
28         Square(z);
    
```

Outline
ReferenceAndOutputParameters.cs
(1 of 2)

The declaration of local variables that are being passed by reference

Passes y with the ref keyword since the variable is initialized

Passes z with the out keyword since the variable is not initialized

Pass y and z by value

© 2006 Pearson Education, Inc. All rights reserved.

```

29
30 // display values of y and z after they are passed to method Square
31 // to demonstrate arguments passed by value are not modified
32 Console.WriteLine("Value of y after Square: {0}", y);
33 Console.WriteLine("Value of z after Square: {0}", z);
34 } // end method DemonstrateReferenceAndOutputParameters
35
36 // uses reference parameter x to modify caller's variable
37 void SquareRef(ref int x)
38 {
39     x = x * x; // squares value of caller's variable
40 } // end method SquareRef
41
42 // uses output parameter x to assign a value
43 // to an uninitialized variable
44 void SquareOut(out int x)
45 {
46     x = 6; // assigns a value to caller's variable
47     x = x * x; // squares value of caller's variable
48 } // end method SquareOut
49
50 // parameter x receives a copy of the value passed as an argument,
51 // so this method cannot modify the caller's variable
52 void Square(int x)
53 {
54     x = x * x;
55 } // end method Square
56 } // end class ReferenceAndOutputParameters
    
```

Outline
ReferenceAndOutputParameters.cs
(2 of 2)

Assignments are saved since arguments was passed by reference

Modifications are disregarded since the argument was passed by value

© 2006 Pearson Education, Inc. All rights reserved.

```

1 // Fig. 7.19: ReferenceAndOutputParametersTest.cs
2 // Application to test class ReferenceAndOutputParameters.
3 class ReferenceAndOutputParametersTest
4 {
5     static void Main( string[] args )
6     {
7         ReferenceAndOutputParameters test =
8             new ReferenceAndOutputParameters();
9         test.DemonstrateReferenceAndOutputParameters();
10    } // end Main
11 } // end class ReferenceAndOutputParametersTest
    
```

Outline
ReferenceAndOutputParametersTest.cs

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

© 2006 Pearson Education, Inc. All rights reserved.