Advanced Programming

Chapter 7:
Methods: A Deeper Look

Dr. Ehsan Khadangi

Some slides are borrowed from Dr. Shahriar Bizhani

Reference

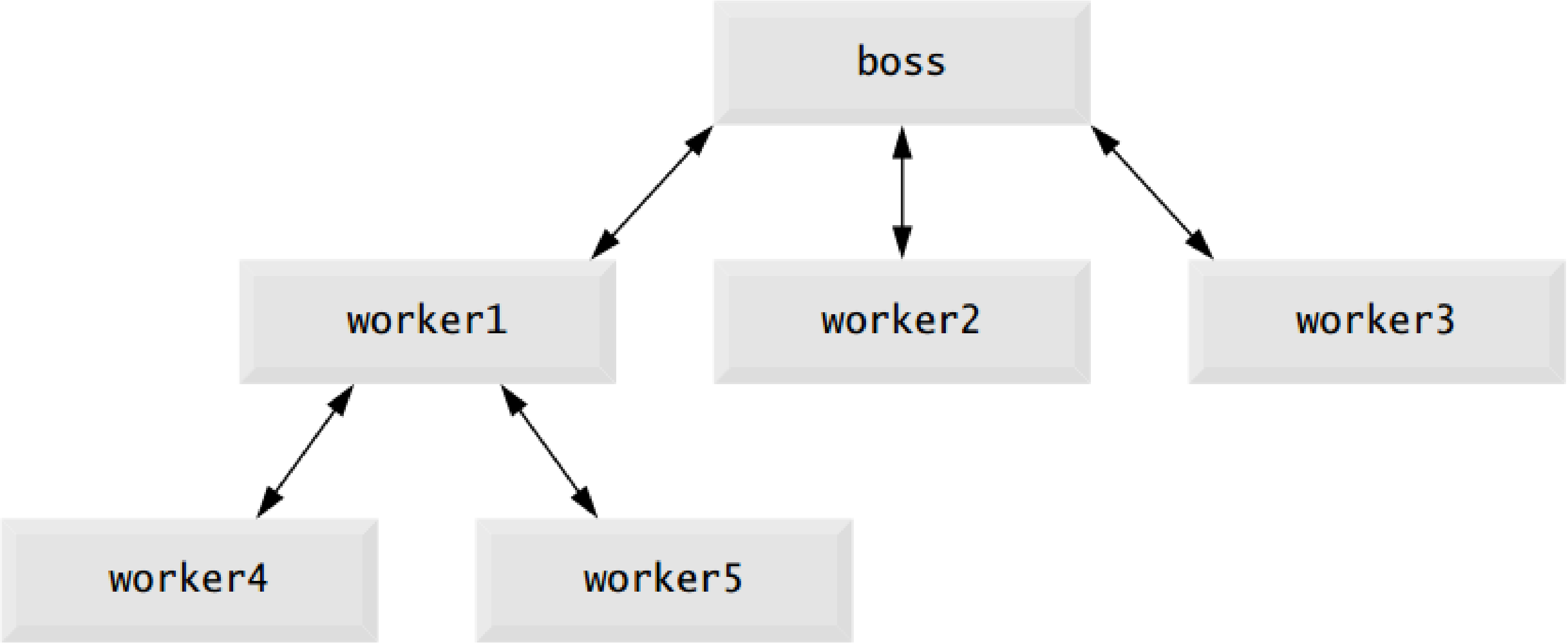
- **Visual C# 2012 How to Program**, Paul Deitel & Harvey Deitel, 5th Edition, Prentice Hall.

Introduction

- ❑ modularize an app by separating its tasks into units

- ❑ **Reasons for using Methods:**
  - ➢ Divide and conquer
  - ➢ Reusability
    - ✓ You can use classes/methods **without knowledge** of **how they work**
  - ➢ Less repetition
    - ✓ Methods can be called several times

Method call-and-return structure



Static methods

❑ Using methods:

- **Generally:**
  - ✓ call a method of the *same* object:
    - – *MethodName*(*argument1*, *arument2*, …)
  - ✓ call a method of another class:
    - – ***ObjectName***.*MethodName*(*argument1*, *arument2*, …)

- **Static methods:**
  - ✓ performs a task that does not depend on specific object.
  - ✓ Call a static method of a class:
    ClassName.*MethodName*(*argument1*, *arument2*, …)

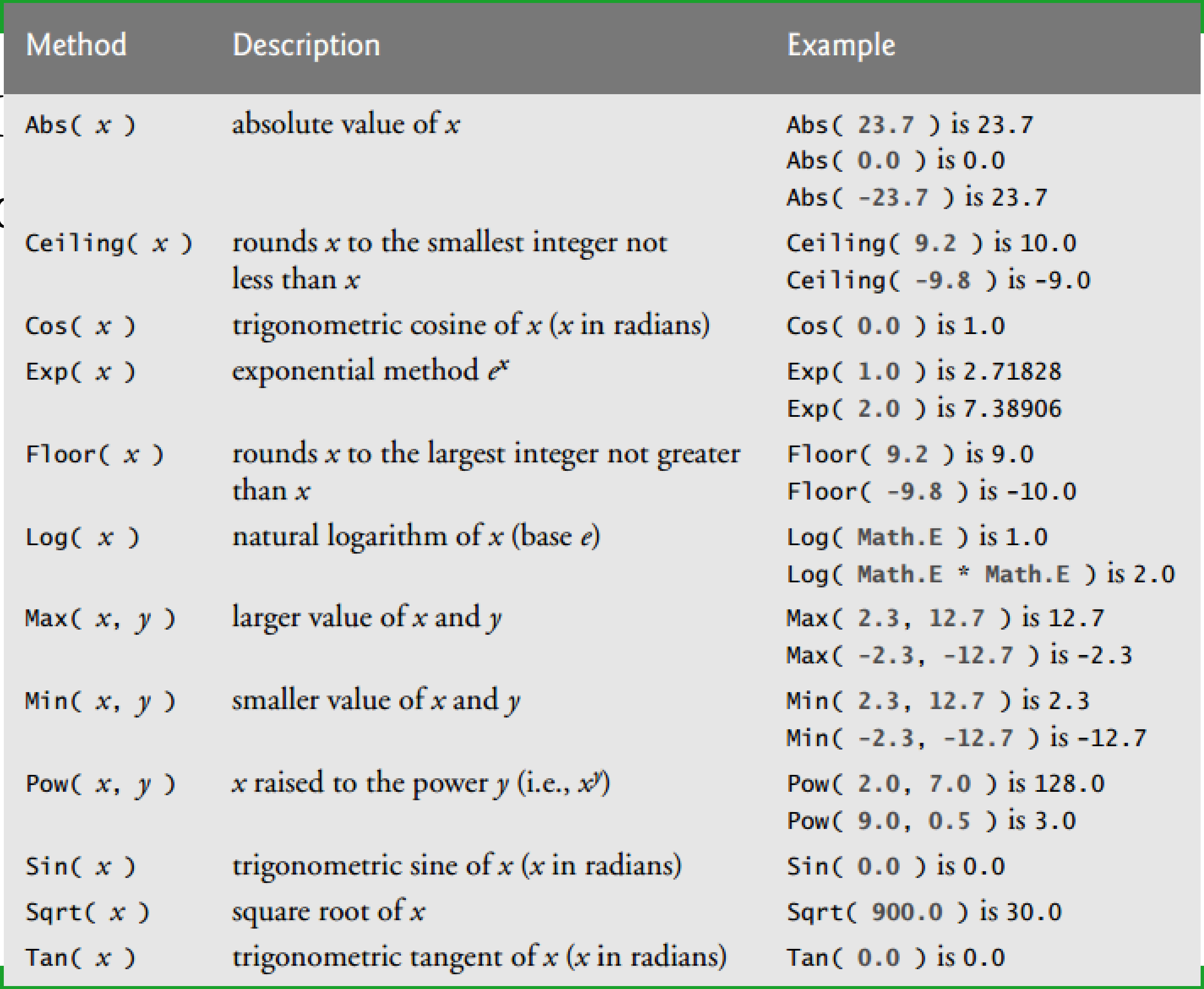
Static members

The Math Class

Methods: A Deeper look

□ M

□ Co



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`

`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`

`Min( x, y )`	smaller value of *x* and *y*	`Min( 2.3, 12.7 )` is `2.3`
		`Min( -2.3, -12.7 )` is `-12.7`

`Pow( x, y )`	*x* raised to the power *y* (i.e., x^y)	`Pow( 2.0, 7.0 )` is `128.0`
		`Pow( 9.0, 0.5 )` is `3.0`

`Sin( x )`	trigonometric sine of *x* (*x* in radians)	`Sin( 0.0 )` is `0.0`
`Sqrt( x )`	square root of *x*	`Sqrt( 900.0 )` is `30.0`
`Tan( x )`	trigonometric tangent of *x* (*x* in radians)	`Tan( 0.0 )` is `0.0`

Static variables

❑ Definition:

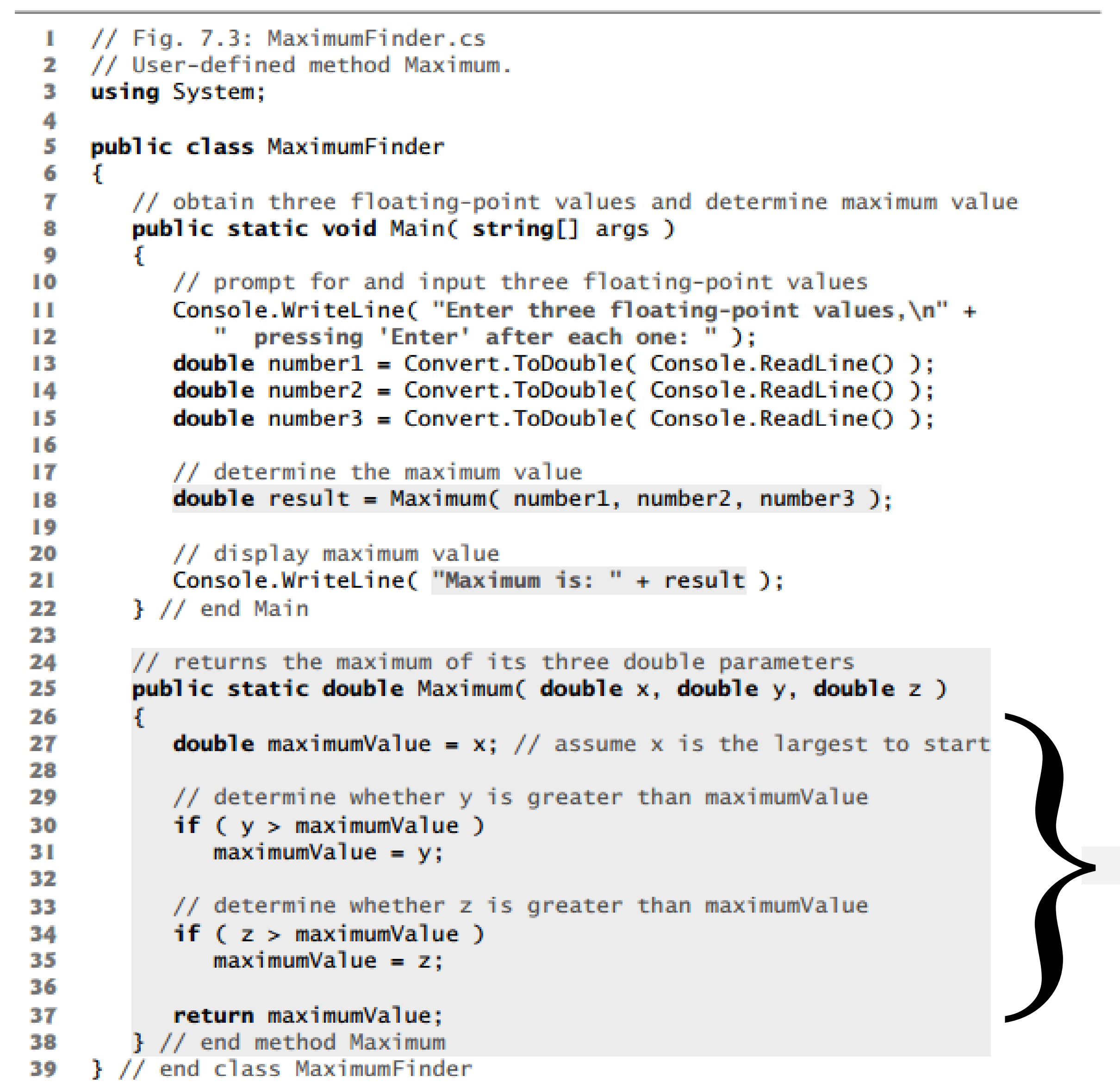
- ➢ Public static const double Pi = 3.14159265358979323846
- ➢ Public static const double E = 2.7182818284590452354

❑ Use:

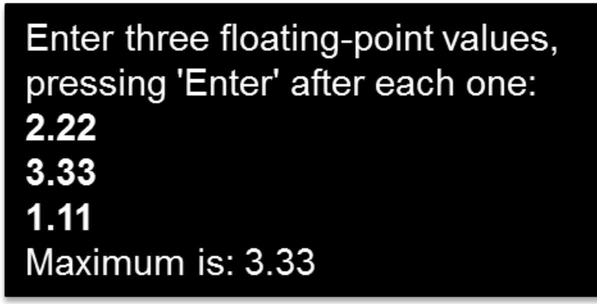
- ➢ Math.PI
- ➢ Math.E

Static members

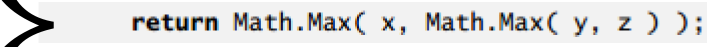
Methods: A Deeper look



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

public class MaximumFinder
{
   // obtain three floating-point values and determine maximum value
   public static void Main( string[] args )
   {
      // 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() );

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

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

   // returns the maximum of its three double parameters
   public static 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 )
         maximumValue = y;

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

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



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



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

String

Methods: A Deeper look

- ❑ *Assembling Strings with String Concatenation*
  - ➢ *+ and += operators*
  - ➢ *+ creates a new string object*

- ❑ *Anything Can Be Converted to a string*
  - ➢ If a bool is concatenated with a string, the bool is converted to the string "True" or "False“

```
bool   b = true;
string  s = “Test is”
WriteLine( s+b );
```

```
Test is True
```

  - ➢ All objects have a **ToString** method that returns a string of it
  - ➢ When an object is concatenated with a string, ToString method is called *implicitly*

Method Call

- ❑ A static method can
  - ➢ **call *only* other static methods** of the same class directly (i.e., using the method name by itself)
  - ➢ **can use *only* static variables** in the same class directly.
- ❑ To access the class’s non-static members, a static method must use a reference to an object of the class.
- ❑ Why?
  - ➢ How would the method know which object’s variables to manipulate

Why is Main static?

```
public static void Main(string args[])
```

❑ Main is Entry Point of the program

❑ During app *startup*, *no objects* of the class have been created

❑ The **Main** method must be called to begin program execution.

❑ **static Main** allows calling Main without creating an object.

Methods: A Deeper look

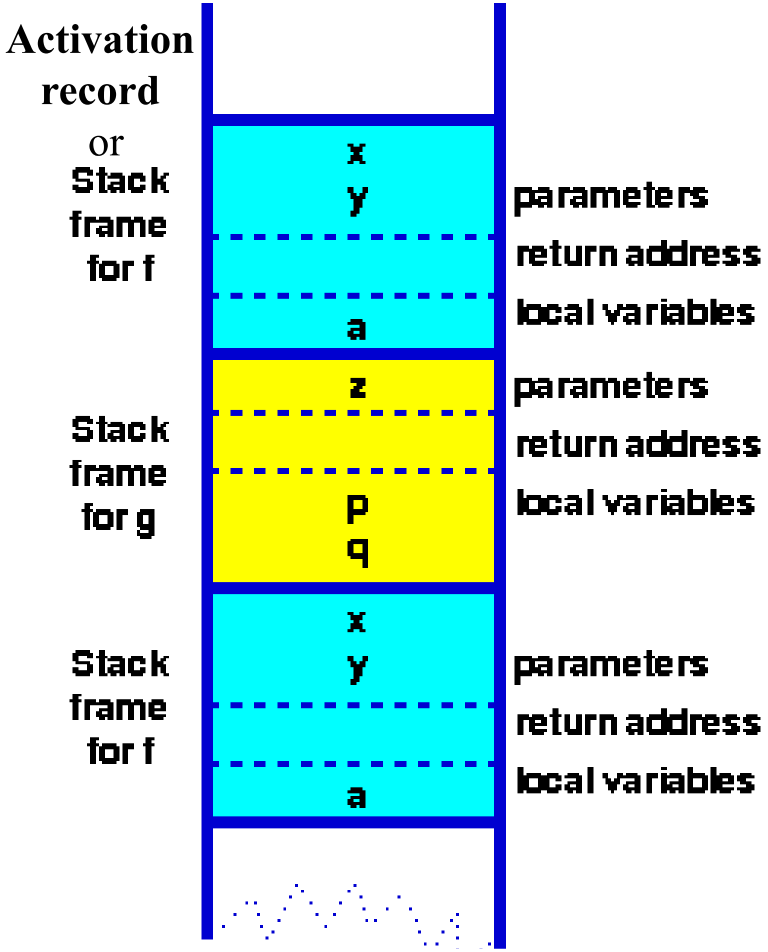
7.6 Method-Call Stack

- ❑ Stack data structure
  - ➢ Last in first out (LIFO)
  - ➢ **Push**: insert a new item on top
  - ➢ **Pop**: remove an item from top
- ❑ When an app calls a method, the called method must know how to return to its caller
- ❑ The return address of the calling method is *pushed* onto the **program-execution stack** (**method-call stack**).
- ❑ Stack overflow

7.6 Method-Call Stack

- ❑ An example of methods f & g

```
Claas X{
     f( int x, int y) {
        int a;
        if (cond )
             return …;
        a = …;
        return g( a );
        }

     g( int z ) {
        int p, q;
        p = …. ; q = …. ;
        return f(p,q);
        }
}
```



Method call stack

7.7 Argument Promotion and Casting

Methods: A Deeper look

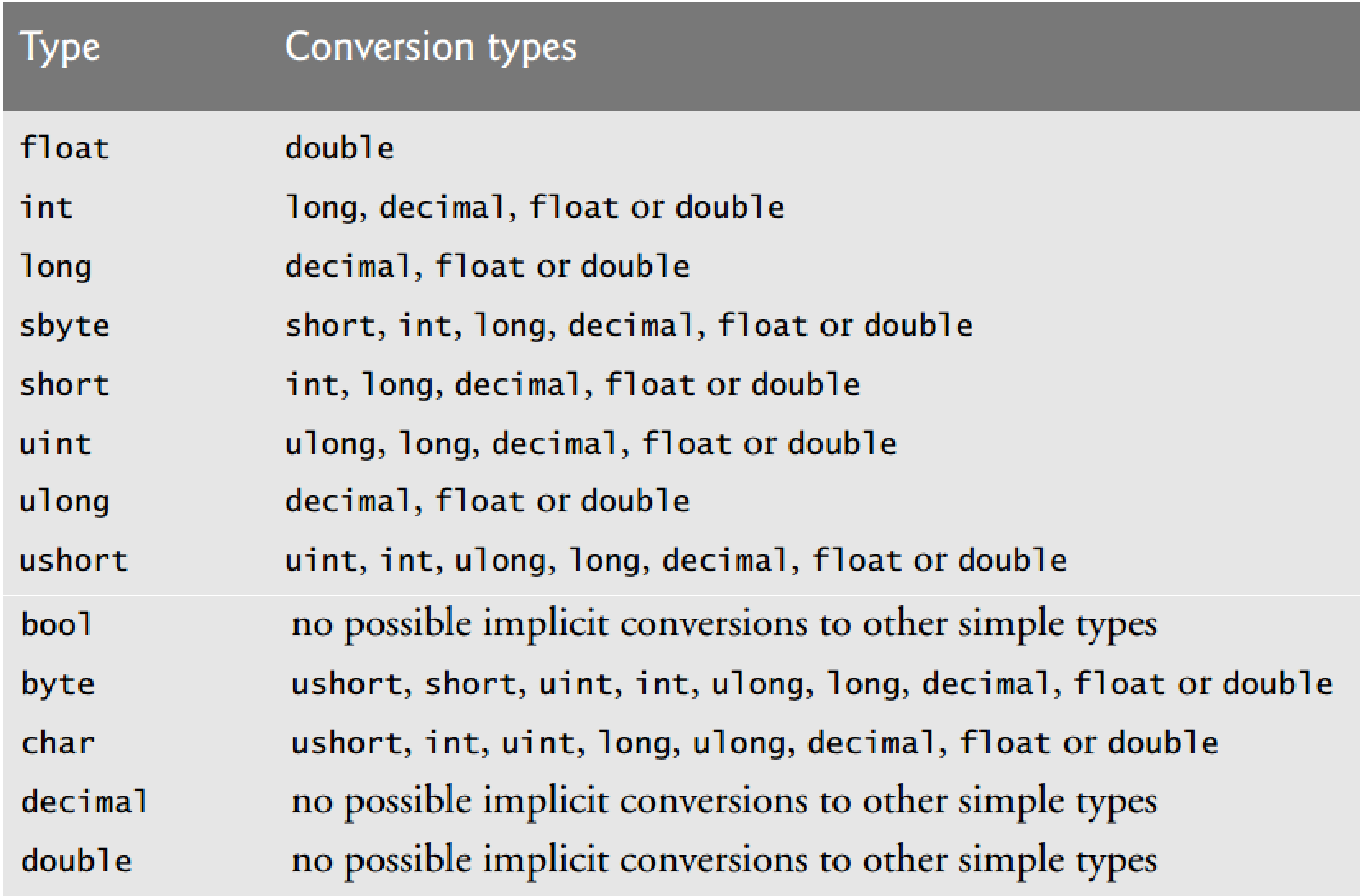
- **Implicit Conversion**
  - Object is converted to a needed type implicitly
  - Only done if complier knows no data will be lost
- **Explicit Conversion**
  - Object is manually converted: e.g. `Square((int) 4.0)`
  - Required if there could be a loss of data:
- **Argument promotion**: implicitly converting an argument's value to the type that the method expects (if possible)
- E.g.

```
Console.WriteLine( Math.Sqrt( 4 ) );
```

convert the int value 4 to the double value 4.0 before passing the value

Promotion Rules

Methods: A Deeper look



Type	Conversion 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
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
decimal	no possible implicit conversions to other simple types
double	no possible implicit conversions to other simple types

7.8 The.net framework class library

- ❑ Do not reinvent the wheel
- ❑ Information hiding
- ❑ Dynamic help
- ❑ .Net framework class library documentation
  - ➢ https://msdn.microsoft.com/en-us/library/mt472912(v=vs.110).aspx
- ❑ .Net API browser

7.9 Random-Number Generation

Methods: A Deeper look

- Within namespace **`System`**
- Can create random byte, int and double values
- Truly random?
  - ✓ The numbers are generated using an equations with a *seed*
    - – The *seed* is usually the exact time of day
- Creating a Random Number Generator Object

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

- Generating a Random Integer

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

  - ✓ Returns a number from 0 to **`Int32.MaxValue`**
    - – Int32.MaxValue = 2,147,483,647

7.9 Random-Number Generation

❑ Scaling & Shifting

```
randomNumbers.Next(6);
```

- ➢ Returns a value from 0 up to but not including *6*

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

or

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

- ➢ Returns a number between *1* and up to but not including *7*
- ➢ Generate random value from a set of values
  e.g. from the sequence 2, 5, 8, 11 and 14

```
number = 2 + 3 * randomNumbers.Next( 5 );
```

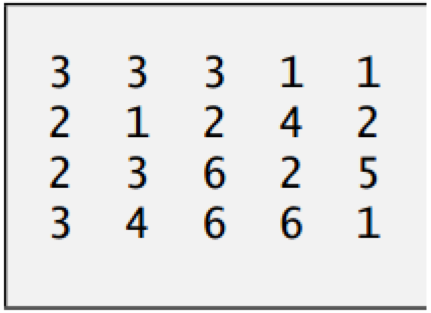
- ➢ Generalization:

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

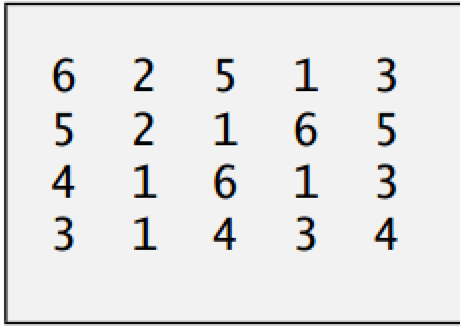
Fig. 7.6

Methods: A Deeper look

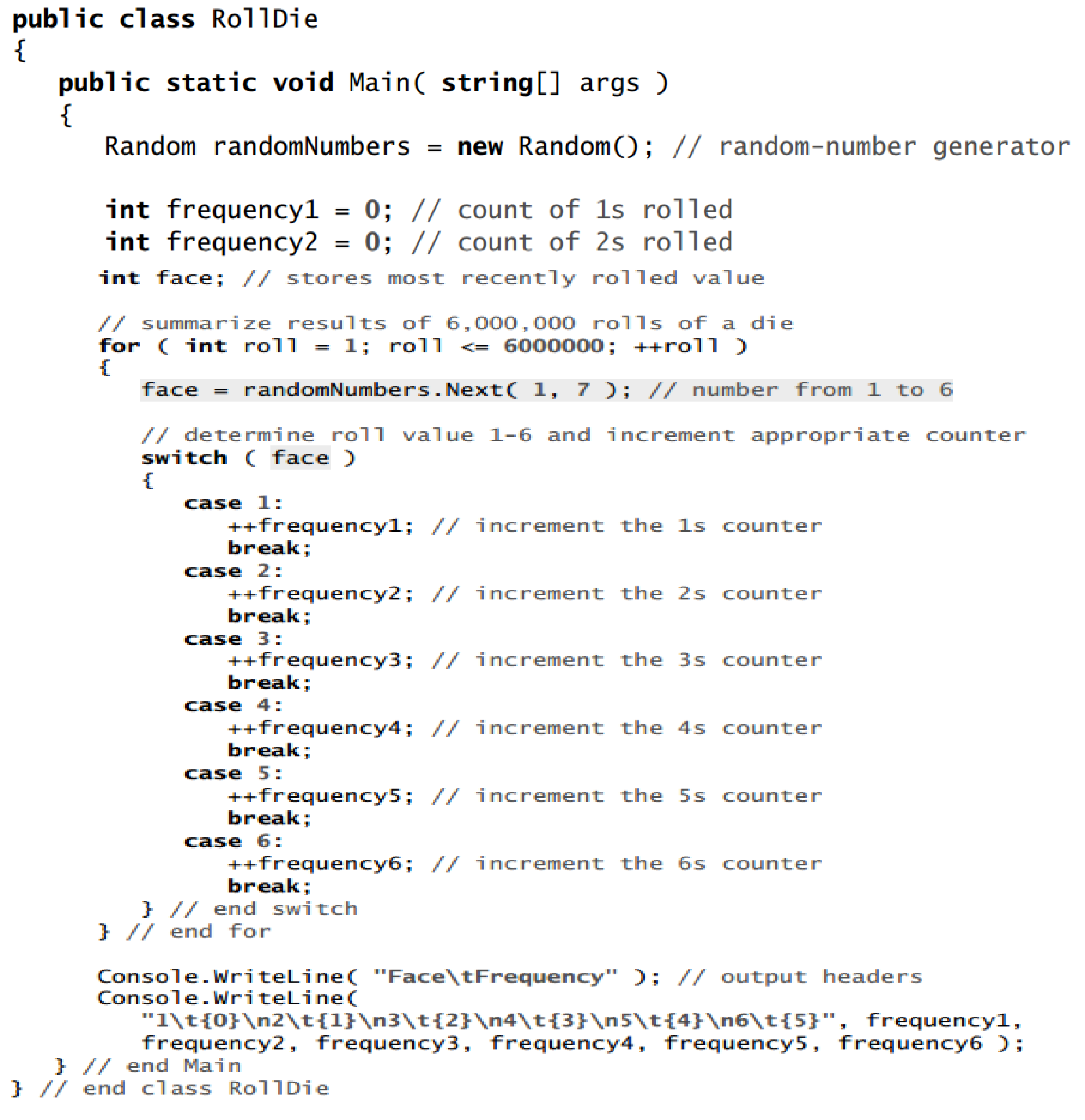
```
   public static void Main( string[] args )
   {
      Random randomNumbers = new Random(); // random-number generator
      int face; // stores each random integer generated

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



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

      // summarize results of 6,000,000 rolls of a die
      for ( int roll = 1; roll <= 6000000; ++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;
            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
```

Problem

- ❑ pseudo random number
- ❑ Random is not cryptographically secure random number generator
  - ➢ System.Security.Cryptography.RNGCryptoServiceProvider

Enumarations

- A type that consists of a set of named constants
- declaration
  - enum enum_name{ enumeration list };
  - enum Day {Sat, Sun, Mon, Tue, Wed, Thu, Fri};
- use initializers
  - enum Day {Sat=1, Sun, Mon, Tue, Wed, Thu, Fri};
- Underlying type
  - enum Day : Int16 {Sat=1, Sun, Mon, Tue, Wed, Thu, Fri};
- Fig. 7.8

7.11 Scope of Declarations

Scope Rules:

1. parameter declaration: the body of the method.
2. local-variable declaration: from the declaration point to the end of that block.
3. local-variable declaration in the `for` header: the body of the `for` statement and the other expressions in the header.
4. Method, property or field of a class: the entire body of the class.
   - non-static methods/properties of a class can use any of the class members (regardless of the declaration order)
   - static methods and properties can use any of the static members of the class.

- Fig. 7.9

7.11 Scope of Declarations

Scope Rules:

1. Method, property or field of a class: the entire body of the class.
   - ➢ non-static methods/properties of a class can use any of the class members (regardless of the declaration order)
   - ➢ static methods and properties can use any of the static members of the class.
2. Any block may contain variable declaration
   - ➢ Local variable with the same name as a field hides the field until block terminate

❑ Fig. 7.9

7.12 Method Overloading

- Methods with the same name
  - Can have the same name but need different arguments
    - Either in type or order of passed variables must be different
  - Usually perform the same task
    - On different data types
- Fig. 7.14

7.13 Optional Parameters

- ❑ Methods can have **optional parameters**:
  - *different number of arguments.*
- ❑ An optional parameter specifies a **default value** of the parameter
- ❑ Declaration:

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

- ❑ Call:

```
Power() // COMPILATION ERROR
Power(10)
Power(10, 3)
```

- ❑ All optional parameters must be placed to the end of parameter list
- ❑ Fig. 7.15

7.14 Named Parameters

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

- ❑ We can call it as follows:

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

- ❑ How can you specify only arguments for the hour and second?

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

```
t.SetTime( hour: 12, second: 22 ); // sets the time to 12:00:22
t.SetTime( second: 30, hour: 10 ); // sets the time to 10:00:30
```

7.15 Recursion

- ❑ Recursive methods
  - ➢ Methods that call themselves
    - ✓ Directly
    - ✓ Indirectly
      - – Call others methods which call it
  - ➢ Continually breaks problem down to simpler forms
  - ➢ Must converge in order to end recursion
  - ➢ Each method call remains open (unfinished)
    - ✓ Finishes each call and then finishes itself

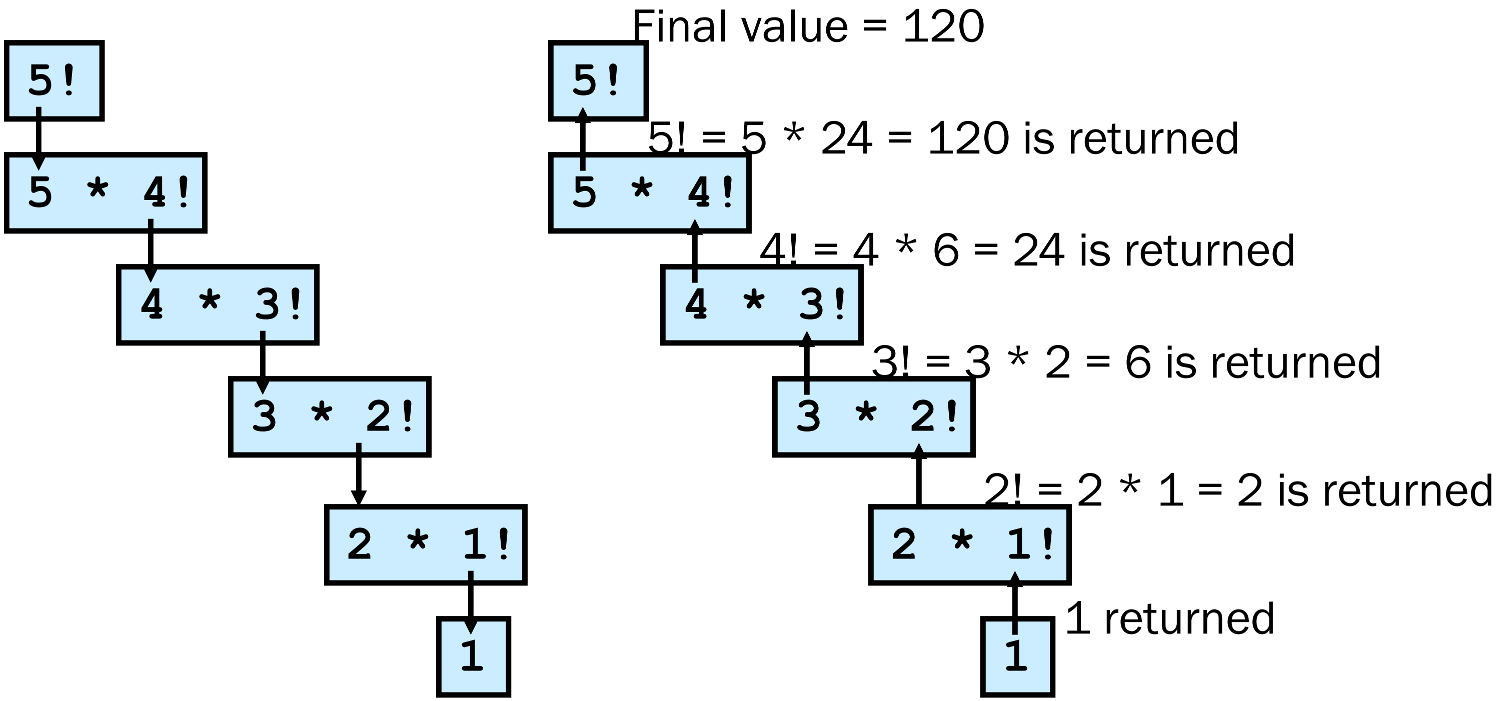
7.15 Recursion

Methods: A Deeper look

- ❑ Non-recursive: n * (n-1) * … * 1

```
factorial = 1;
for ( int counter = number; counter >= 1; --counter )
   factorial *= counter;
```

- ❑ Recursive: n! = n * (n – 1)!


5!
5 * 4!
4 * 3!
3 * 2!
2 * 1!
1
Final value = 120
5!
5! = 5 * 24 = 120 is returned
5 * 4!
4! = 4 * 6 = 24 is returned
4 * 3!
3! = 3 * 2 = 6 is returned
3 * 2!
2! = 2 * 1 = 2 is returned
2 * 1!
1 returned
1

(a) Procession of recursive calls.

(b) Values returned from each recursive call.

C# 6 Expression-Bodied Methods and Properties

Methods: A Deeper look

- Concise syntax for
  - Methods that contain only a return statement
  - Read-only properties with simple get accessor
  - Methods contain single statement
- Static int Cube(int x){

```
        return x*x*x;
}
static int Cube(int x) => x*x*x
public bool NoFault => State==“NY” || State==“NJ”
```

7.16 Pass-by-Value vs. Pass-by-Reference

Methods: A Deeper look

- **Passing by value**
  - Send a method a copy of the object (by default)
  - Changes to the copy, do *not* affect the original value
  - When returned are always returned by value
- **Passing by reference**
  - Send a method the actual reference point
    - ✓ Causes the variable to be changed throughout the program
  - The **`ref`** keyword specifies by reference
    - ✓ An uninitialized variable generates a compiler error.
  - The **`out`** keyword means a called method will initialize it
  - pass a reference-type variable by reference, allows you to modify it .
- Fig. 7.20