CS313D: ADVANCED PROGRAMMING LANGUAGE

Computer Science Department

Lecture 3: C# language basics

Lecture Contents

- C# basics
  - Conditions
  - Loops
  - Methods
  - Arrays

Conditions and branching

Relational operators

=	Equal to	==	x + 7 == 2*y	x + 7 = 2y
≠	Not equal to	!=	ans != 'n'	*ans* ≠ *'n'*
<	Less than	<	count < m + 3	*count* < m + 3
≤	Less than or equal to	<=	time <= limit	*time* ≤ *limit*
>	Greater than	>	time > limit	*time* > *limit*
≥	Greater than or equal to	>=	age >= 21	*age* ≥ 21

Logical Operators

AND

Exp_1	*Exp_2*	*Exp_1* && *Exp_2*
true	true	true
true	false	false
false	true	false
false	false	false

OR

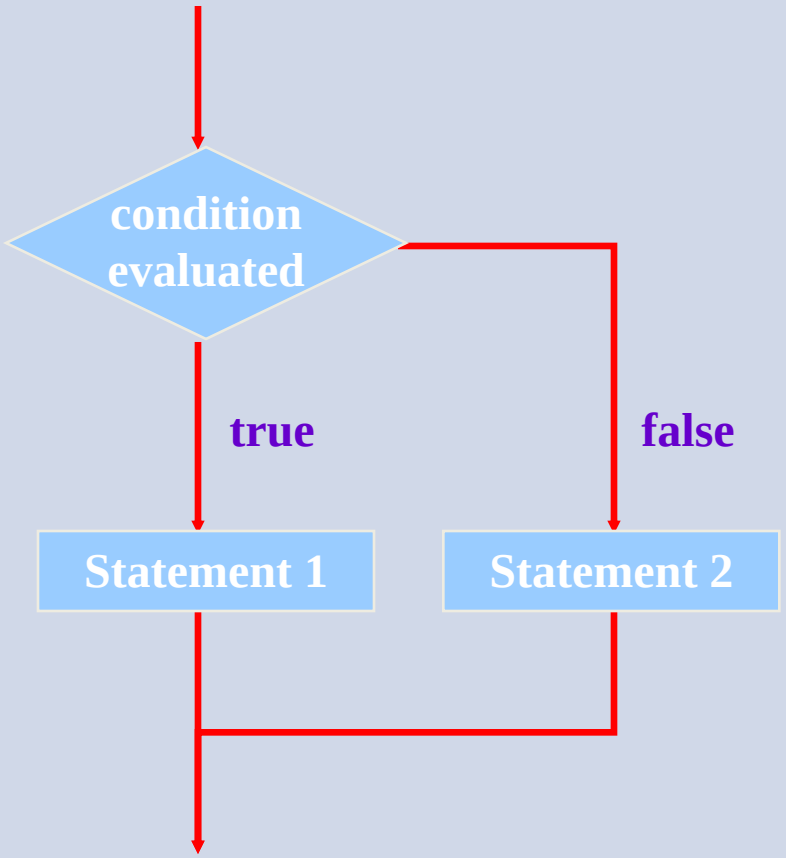
Exp_1	*Exp_2*	*Exp_1* || *Exp_2*
true	true	true
true	false	true
false	true	true
false	false	false

NOT

Exp	!(*Exp*)
true	false
false	true

Conditional Statements

- A *conditional statement* lets us choose which statement will be executed next
- Conditional statements give us the power to make basic decisions
- Java's conditional statements:
  - the *`if` and `if-else` statements*
  - the conditional operator
  - the *`switch` statement*



Logic of an `if-else` statement

- Several statements can be grouped together into a *block statement*
- A block is delimited by braces ({ … })

The `if` Statement

- The *`if` statement* has the following syntax:

The condition must be a *boolean expression*.
e.g., a boolean variable, a == b, a <= b.
It must evaluate to either true or false.

`if` is a Java reserved word

```
if ( condition )
    statement1;
else
    statement2;
```

If the condition is true, this statement is executed.

If the condition is false, this statement is executed.

Conditional Operator

- Also called "ternary operator"
  - Allows embedded conditional in expression
  - Essentially "shorthand if-else" operator
  - Example:

```
if (n1 > n2)
      max = n1;
else
      max = n2;
```

  - Can be written:

```
max = (n1 > n2) ? n1 : n2;
```

    - "?" and ":" form this "ternary" operator

switch

multiple-selection statement

a constant integral expression of type: byte, short, int or char.

switch Statement

SYNTAX

```
switch (Controlling_Expression)
{
    case Constant_1:
        Statement_Sequence_1
        break;
    case Constant_2:
        Statement_Sequence_2
        break;
                .
                .
                .
    case Constant_n:
        Statement_Sequence_n
        break;
    default:
        Default_Statement_Sequence
}
```

case **labels**

break statement

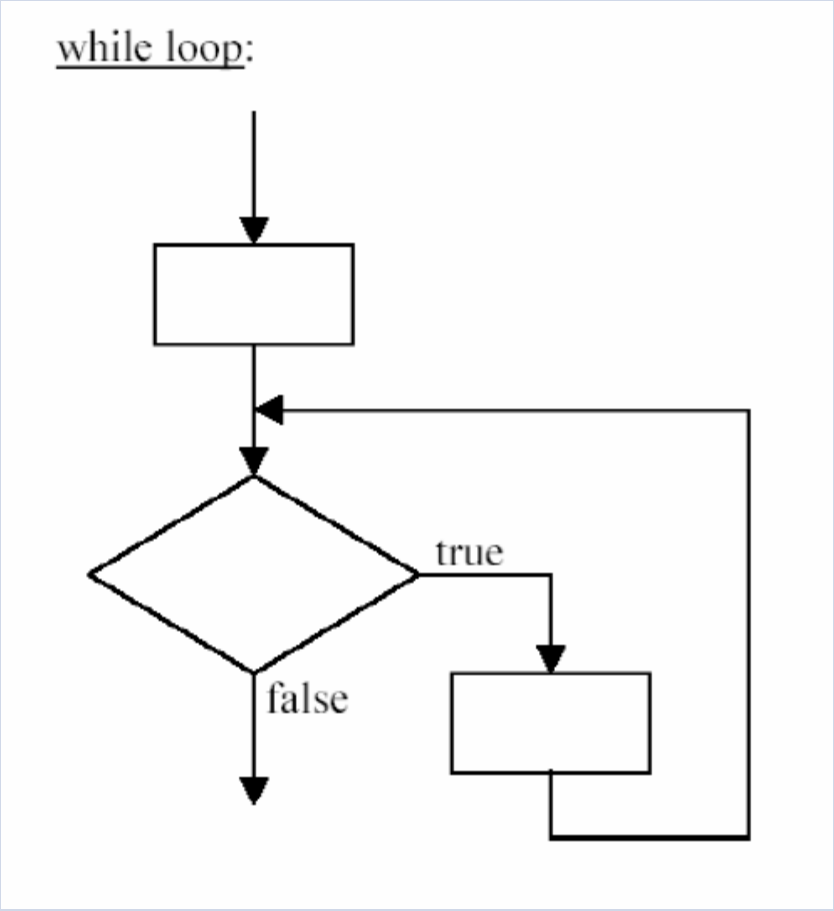
optional default **case**

11

loops

Chapters 5 & 6

Dr. Amal Khalifa, Spr 2015



sentinel value

While loops

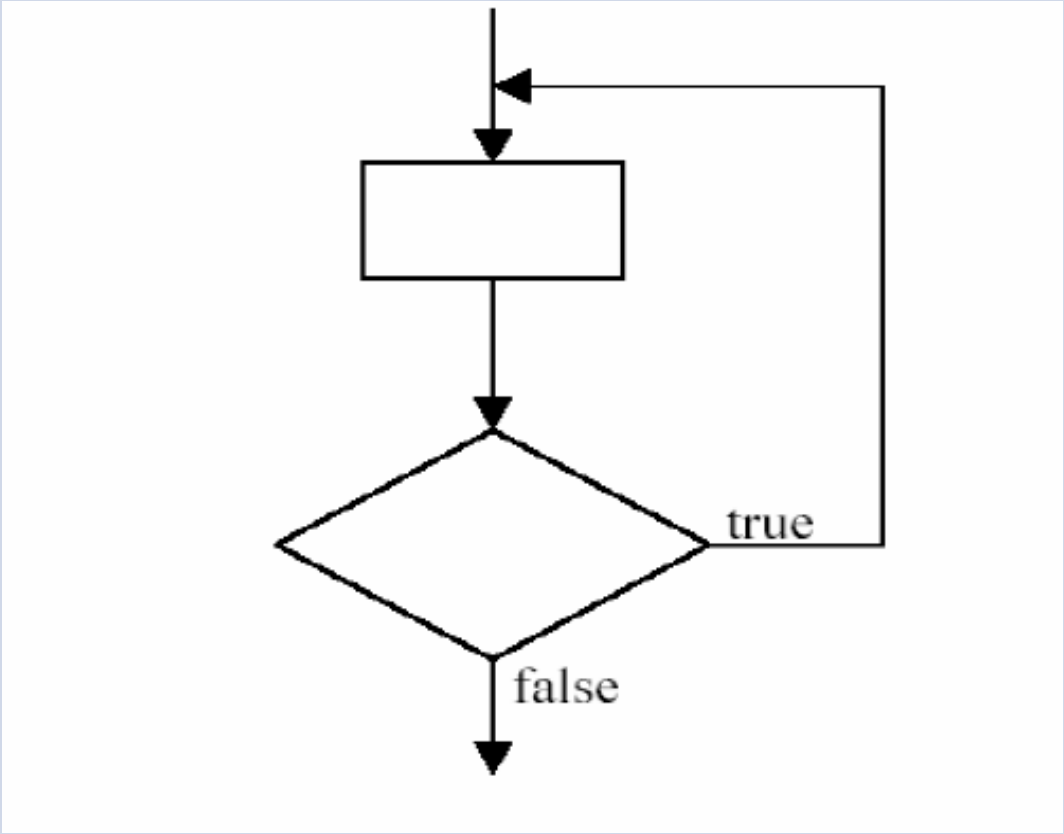
```
while ( condition )
    statement;
```

Example


```
1  // Fig. 6.1: WhileCounter.cs
2  // Counter-controlled repetition with the while repetition statement.
3  using System;
4
5  public class WhileCounter
6  {
7     public static void Main( string[] args )
8     {
9        int counter = 1; // declare and initialize control variable
10
11       while ( counter <= 10 ) // loop-continuation condition
12       {
13          Console.Write( "{0}  ", counter );
14          ++counter; // increment control variable
15       } // end while
16
17       Console.WriteLine(); // output a newline
18    } // end Main
19 } // end class WhileCounter
```

```
1  2  3  4  5  6  7  8  9  10
```

Fig. 6.1 | Counter-controlled repetition with the `while` repetition statement.



do loops

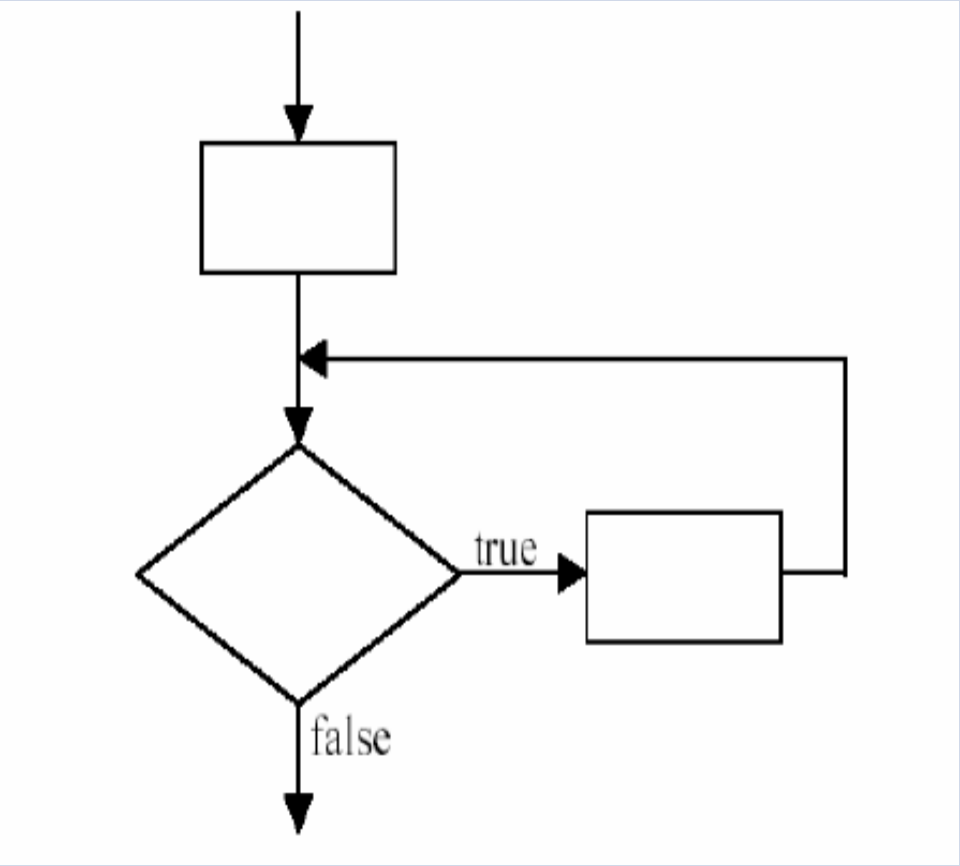
```
do
{
    statement;
}
while ( condition );
```

Example


```
1  // Fig. 6.7: DoWhileTest.cs
2  // do...while repetition statement.
3  using System;
4
5  public class DoWhileTest
6  {
7     public static void Main( string[] args )
8     {
9        int counter = 1; // initialize counter
10
11       do
12       {
13          Console.Write( "{0}  ", counter );
14          ++counter;
15       } while ( counter <= 10 ); // end do...while
16
17       Console.WriteLine(); // outputs a newline
18    } // end Main
19 } // end class DoWhileTest
```

```
1  2  3  4  5  6  7  8  9  10
```

Fig. 6.7 | do...while repetition statement.



for loops

```
for ( initialization ; condition ; increment )
    statement;
```

Example

```
1  // Fig. 6.2: ForCounter.cs
2  // Counter-controlled repetition with the for repetition statement.
3  using System;
4
5  public class ForCounter
6  {
7     public static void Main( string[] args )
8     {
9        // for statement header includes initialization,
10       // loop-continuation condition and increment
11       for ( int counter = 1; counter <= 10; ++counter )
12          Console.Write( "{0}  ", counter );
13
14       Console.WriteLine(); // output a newline
15    } // end Main
16 } // end class ForCounter
```

```
1  2  3  4  5  6  7  8  9  10
```

Fig. 6.2 | Counter-controlled repetition with the `for` repetition statement.

break and Continue

- break;
  - Forces loop to exit immediately.
  - Execution continues with the first statement after the control statement.
- continue;
  - Skips rest of loop body
  - In while and do…while statements, the program evaluates the loop-continuation test immediately after the continue statement executes.
  - In a for statement, the increment expression executes, then the program evaluates the loop-continuation test.

19

Methods

Chapter 7

Software Engineering Observation 7.1

Don't try to "reinvent the wheel." When possible, reuse Framework Class Library classes and methods (`msdn.microsoft.com/en-us/library/ms229335.aspx`). This reduces app development time, avoids introducing programming errors and contributes to good app performance.

Packaging Code in C#

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

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

Fig. 7.2 | `Math` class methods. (Part 1 of 2.)

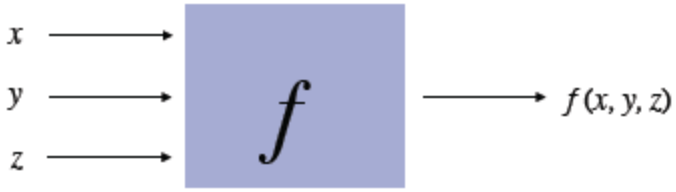
Method	Description	Example
`Pow( x, y )`	x raised to the power y (i.e., x^y)	`Pow( 2.0, 7.0 )` is `128.0` `Pow( 9.0, 0.5 )` is `3.0`
`Sin( x )`	trigonometric sine of x (x in radians)	`Sin( 0.0 )` is `0.0`
`Sqrt( x )`	square root of x	`Sqrt( 900.0 )` is `30.0`
`Tan( x )`	trigonometric tangent of x (x in radians)	`Tan( 0.0 )` is `0.0`

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

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

Methods

- A method:
  - groups a sequence of statement
  - takes input, performs actions, and produces output
  - In Java, each method is defined within specific class



Method Declaration: Header

- A method declaration begins with a *method header*

```
public class MyClass
{ …
   static  int    min ( int num1, int num2 )
```

properties

return type

method name

parameter list

The parameter list specifies the type and name of each parameter

The name of a parameter in the method declaration is called a formal argument

Method Declaration: Body

The header is followed by the *method body*:

```
class MyClass
{
    …
    static int min(int num1, int num2)
    {
        int minValue = num1 < num2 ? num1 : num2;
        return minValue;
    }

    …
 }
```

The `return` Statement

- The *return type* of a method indicates the type of value that the method sends back to the calling location
  - A method that does not return a value has a `void` return type

- The *return statement* specifies the value that will be returned
  - Its expression must conform to the return type

Calling a Method

- Each time a method is called, the values of the *actual arguments* in the invocation are assigned to the *formal arguments*

```
int  num = min(2, 3);
```

```
static int min (int num1, int num2)
{
    int minValue = (num1 < num2 ? num1 : num2);
    return minValue;
}
```

Example

```
1  // Fig. 7.3: MaximumFinder.cs
2  // User-defined method Maximum.
3  using System;
4
5  public class MaximumFinder
6  {
7     // obtain three floating-point values and determine maximum value
8     public static void Main( string[] args )
9     {
10       // prompt for and input three floating-point values
11       Console.WriteLine( "Enter three floating-point values,\n" +
12          "  pressing 'Enter' after each one: " );
13       double number1 = Convert.ToDouble( Console.ReadLine() );
14       double number2 = Convert.ToDouble( Console.ReadLine() );
15       double number3 = Convert.ToDouble( Console.ReadLine() );
16
17       // determine the maximum value
18       double result = Maximum( number1, number2, number3 );
19
20       // display maximum value
21       Console.WriteLine( "Maximum is: " + result );
22    } // end Main
23
```

Fig. 7.3 | User-defined method `Maximum`. (Part 1 of 2.)

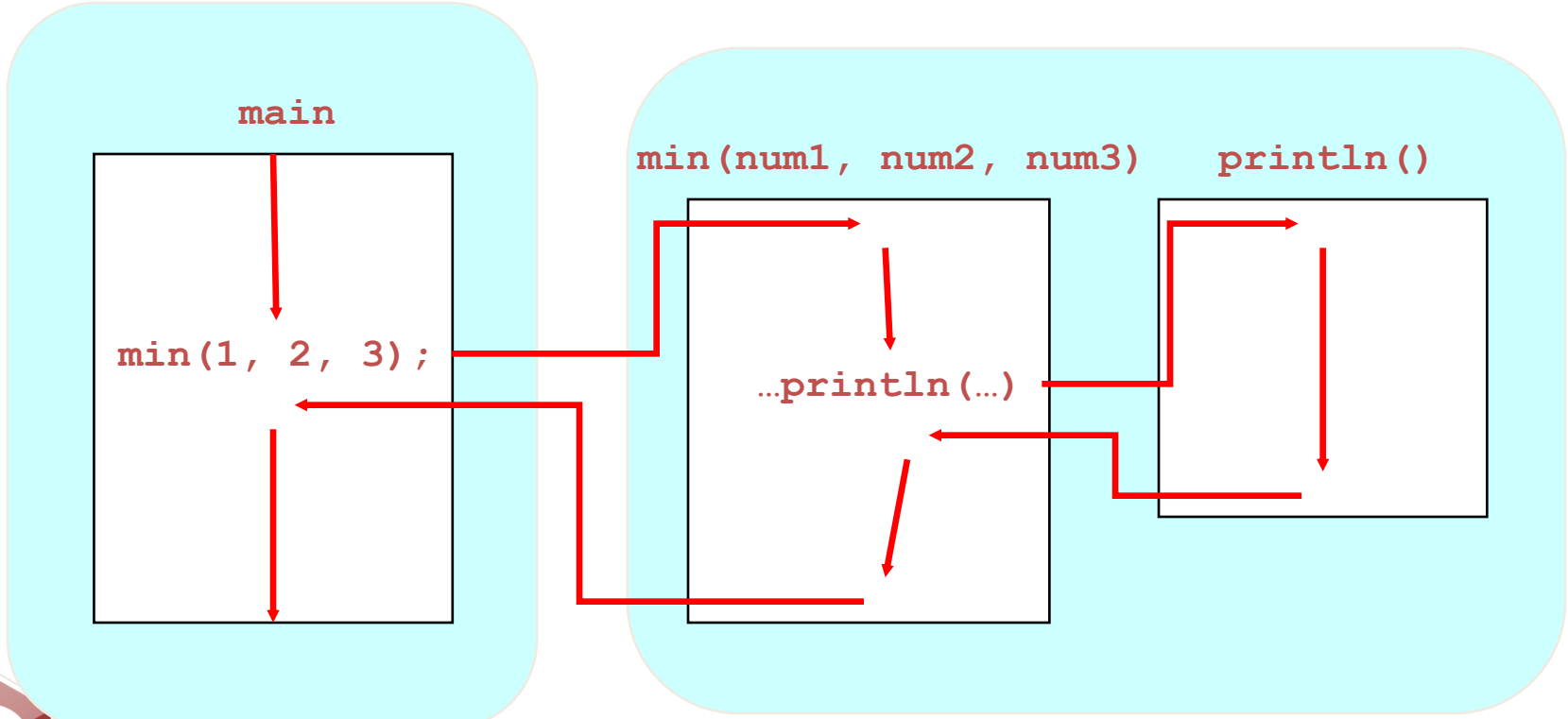
```
24      // returns the maximum of its three double parameters
25      public static double Maximum( double x, double y, double z )
26      {
27         double maximumValue = x; // assume x is the largest to start
28
29         // determine whether y is greater than maximumValue
30         if ( y > maximumValue )
31            maximumValue = y;
32
33         // determine whether z is greater than maximumValue
34         if ( z > maximumValue )
35            maximumValue = z;
36
37         return maximumValue;
38      } // end method Maximum
39   } // end class MaximumFinder
```

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

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

Method Call Stack

- A method can call another method, who can call another method, …



Method Overloading

- Methods of the same name declared in the same class
- Must have different sets of parameters (signatures).
- the compiler differentiates signatures by :
  - the *number* of parameters,
  - the *types* of the parameters and
  - the *order* of the parameter types in each signature.
- *Method calls cannot be distinguished by return type.*
  - Overloaded methods can have different return types if the methods have different parameter lists.

Example

```
1  // Fig. 7.10: 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
```

Fig. 7.10 | Overloaded method declarations. (Part 1 of 2.)

```
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. 7.10 | Overloaded method declarations. (Part 2 of 2.)

Common Programming Error 7.9

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

Error List

1 Error | 0 Warnings | 0 Messages | Search Error List

	Description	File	Line	Column	Project
1	Type 'MethodOverloadError' already defines a member called 'Square' with the same parameter types	MethodOverloadError.cs	14	25	MethodOverloadError

Fig. 7.11 | Overloaded methods with identical signatures cause compilation errors, even if return types are different. (Part 2 of 2.)

Error!!

Optional Parameters

- 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.
- Example:

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

  - 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.*

Common Programming Error 7.10

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

7.13 Optional Parameters (Cont.)

- 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 generates a compilation error because this method requires a minimum of one argument.
- The second is valid because one argument (10) is being passed—the optional exponentValue is not specified in the method call.

Example

```
1  // Fig. 7.12: Power.vb
2  // Optional argument demonstration with method Power.
3  using System;
4
5  class CalculatePowers
6  {
7     // call Power with and without optional arguments
8     public static void Main( string[] args )
9     {
10       Console.WriteLine( "Power(10) = {0}", Power( 10 ) ) ;
11       Console.WriteLine( "Power(2, 10) = {0}", Power( 2, 10 ) );
12    } // end Main
13
14    // use iteration to calculate power
15    public int Power( int baseValue, int exponentValue = 2 )
16    {
17       int result = 1; // initialize total
18
19       for ( int i = 1; i <= exponentValue; i++ )
20          result *= baseValue;
21
22       return result;
23    } // end method Power
24 } // end class CalculatePowers
```

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

Fig. 7.12 | Optional argument demonstration with method `Power`. (Part 1 of 2.)

Passing arguments to Methods

Pass-by-value

- also called call-by-value
- A copy of the argument's value is passed to the called method.
- The called method works exclusively with the copy → Changes to the called method's copy do not affect the original variable's value in the caller.

Pass-by-reference

- also called call-by-reference
- The called method can access the argument's value directly and modify that data, if necessary
- Improves performance by eliminating the need to copy

Passing arguments to Methods

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

Example

```
1  // Fig. 7.15: 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 static void Main( string[] args )
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\n" );
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're 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}\n", z );
```

Fig. 7.15 | Reference, output and value parameters. (Part 1 of 3.)

```
25
26          // pass y and z by value
27          Square( y );
28          Square( z );
29
30          // display values of y and z after they're passed to method Square
31          // to demonstrate that 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 Main
35
36       // uses reference parameter x to modify caller's variable
37       static 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       static 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       static void Square( int x )
53       {
54          x = x * x;
55       } // end method Square
56    } // end class ReferenceAndOutputParameters
```

```
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
```

Common Programming Error 7.12

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

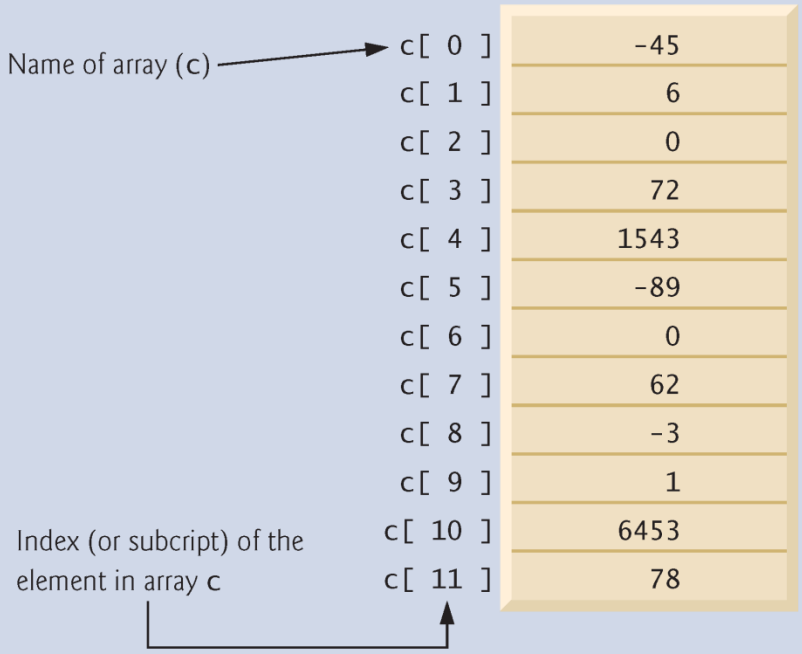
Be careful!!

Arrays & Strings

Chapter 8

What is an array?

- Array
  - data structures
  - Group of variables (called elements) containing values of the same type.
  - related data items of the same type.
  - fixed length once created.
  - Elements referred to using index or subscript.
  - In C#, Arrays are objects, so they’re considered reference types.
  - Every array object knows its own length and stores it in a `Length` instance variable.
  - Elements can be either primitive types or reference types (strings).



Array elements

An index must be a nonnegative integer → Can use an expression as an index

Every array object knows its own length and stores it in a `length` instance variable

7.3 Arrays in Java


```
int[] a;
int a[];
```

```
a = new int[5];
```

```
for(int i=0;i<5;i++)
    a[i] = i*i;
```

Example


```
Index   Value
    0       0
    1       0
    2       0
    3       0
    4       0
```

```
1  // Fig. 8.2: InitArray.cs
2  // Creating an array.
3  using System;
4
5  public class InitArray
6  {
7     public static void Main( string[] args )
8     {
9        int[] array; // declare array named array
10
11       // create the space for array and initialize to default zeros
12       array = new int[ 5 ]; // 5 int elements
13
14       Console.WriteLine( "{0}{1,8}", "Index", "Value" ); // headings
15
16       // output each array element's value
17       for ( int counter = 0; counter < array.Length; ++counter )
18          Console.WriteLine( "{0,5}{1,8}", counter, array[ counter ] );
19    } // end Main
20 } // end class InitArray
```

Fig. 8.2 | Creating an array. (Part 1 of 2.)

Example:


```
1  // Fig. 8.3: InitArray.cs
2  // Initializing the elements of an array with an array initializer.
3  using System;
4
5  public class InitArray
6  {
7     public static void Main( string[] args )
8     {
9        // initializer list specifies the value for each element
10       int[] array = { 32, 27, 64, 18, 95, 14, 90, 70, 60, 37 };
11
12       Console.WriteLine( "{0}{1,8}", "Index", "Value" ); // headings
13
14       // output each array element's value
15       for ( int counter = 0; counter < array.Length; ++counter )
16          Console.WriteLine( "{0,5}{1,8}", counter, array[ counter ] );
17    } // end Main
18 } // end class InitArray
```

```
Index   Value
    0      32
    1      27
    2      64
    3      18
    4      95
    5      14
    6      90
    7      70
    8      60
    9      37
```

Fig. 8.3 | Initializing the elements of an array with an array initializer. (Part 1 of 2.)

Example

```
1  // Fig. 8.5: SumArray.cs
2  // Computing the sum of the elements of an array.
3  using System;
4
5  public class SumArray
6  {
7     public static void Main( string[] args )
8     {
9        int[] array = { 87, 68, 94, 100, 83, 78, 85, 91, 76, 87 };
10       int total = 0;
11
12       // add each element's value to total
13       for ( int counter = 0; counter < array.Length; ++counter )
14          total += array[ counter ];
15
16       Console.WriteLine( "Total of array elements: {0}", total );
17    } // end Main
18 } // end class SumArray
```

```
Total of array elements: 849
```

Fig. 8.5 | Computing the sum of the elements of an array.

Example

```
1  // Fig. 8.7: RollDie.cs
2  // Roll a six-sided die 6,000,000 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       int[] frequency = new int[ 7 ]; // array of frequency counters
11
12       // roll die 6,000,000 times; use die value as frequency index
13       for ( int roll = 1; roll <= 6000000; ++roll )
14          ++frequency[ randomNumbers.Next( 1, 7 ) ];
15
16       Console.WriteLine( "{0}{1,10}", "Face", "Frequency" );
17
18       // output each array element's value
19       for ( int face = 1; face < frequency.Length; ++face )
20          Console.WriteLine( "{0,4}{1,10}", face, frequency[ face ] );
21    } // end Main
22 } // end class RollDie
```

Fig. 8.7 | Roll a six-sided die 6,000,000 times. (Part 1 of 2.)

```
Face Frequency
   1    999924
   2   1000939
   3   1001249
   4    998454
   5   1000233
   6    999201
```

Fig. 8.7 | Roll a six-sided die 6,000,000 times. (Part 2 of 2.)

foreach Statement

- The **foreach statement** iterates through the elements of an entire array or collection.
- syntax

```
foreach( type identifier in arrayName )
    statement
```

  - *type* and *identifier* are the type and name (e.g., `int number`) of the **iteration variable**.
  - *arrayName* is the array through which to iterate.
- The type of the iteration variable must be consistent with the type of the elements in the array.
- The iteration variable represents successive values in the array on successive iterations of the `foreach` statement.

Example

```
1  // Fig. 8.12: ForEachTest.cs
2  // Using the foreach statement to total integers in an array.
3  using System;
4
5  public class ForEachTest
6  {
7     public static void Main( string[] args )
8     {
9        int[] array = { 87, 68, 94, 100, 83, 78, 85, 91, 76, 87 };
10       int total = 0;
11
12       // add each element's value to total
13       foreach ( int number in array )
14          total += number;
15
16       Console.WriteLine( "Total of array elements: {0}", total );
17    } // end Main
18 } // end class ForEachTest
```

```
Total of array elements: 849
```

Fig. 8.12 | Using the `foreach` statement to total integers in an array.

That's all …..