



Chapter 08 – Advance Method Concepts:

Objectives

- Learn about parameter types and review mandatory value parameters
- Use `ref` parameters, `out` parameters, and parameter arrays
- Overload methods
- Learn how to avoid ambiguous methods
- Use optional parameters



Understanding Parameter Types

- **Mandatory parameter**
 - An argument for it is required in every method call
- Four types of mandatory parameters:
 - Value parameters
 - Declared without any modifiers
 - Reference parameters
 - Declared with the `ref` modifier
 - Output parameters
 - Declared with the `out` modifier
 - Parameter arrays
 - Declared with the `params` modifier



Using Mandatory Value Parameters

- **Value parameter**
 - The method receives a copy of the value passed to it
 - The copy is stored at a different memory address than the actual parameter
- Changes to value parameters never affect the original argument in the calling method

Using Mandatory Value Parameters (cont'd.)

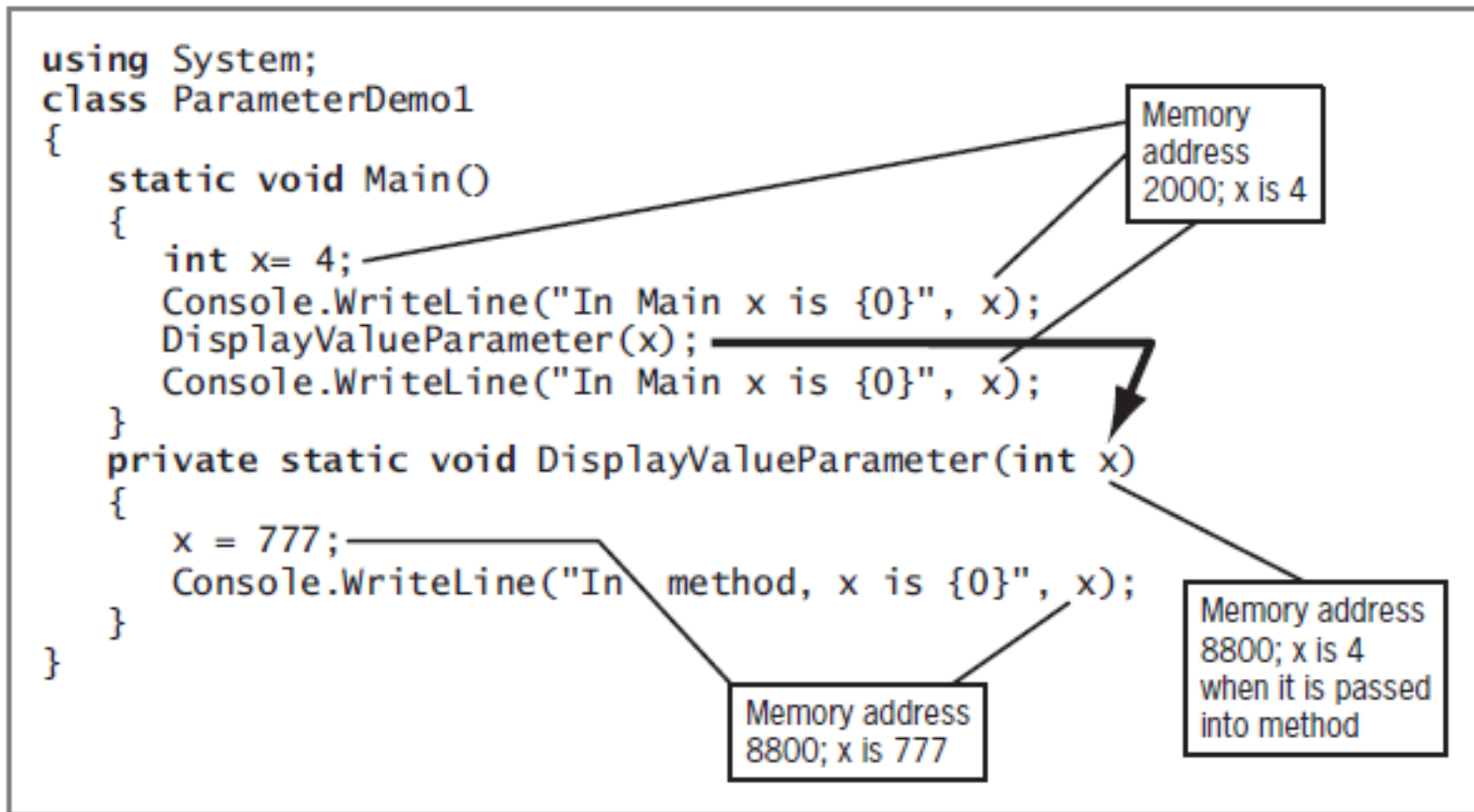
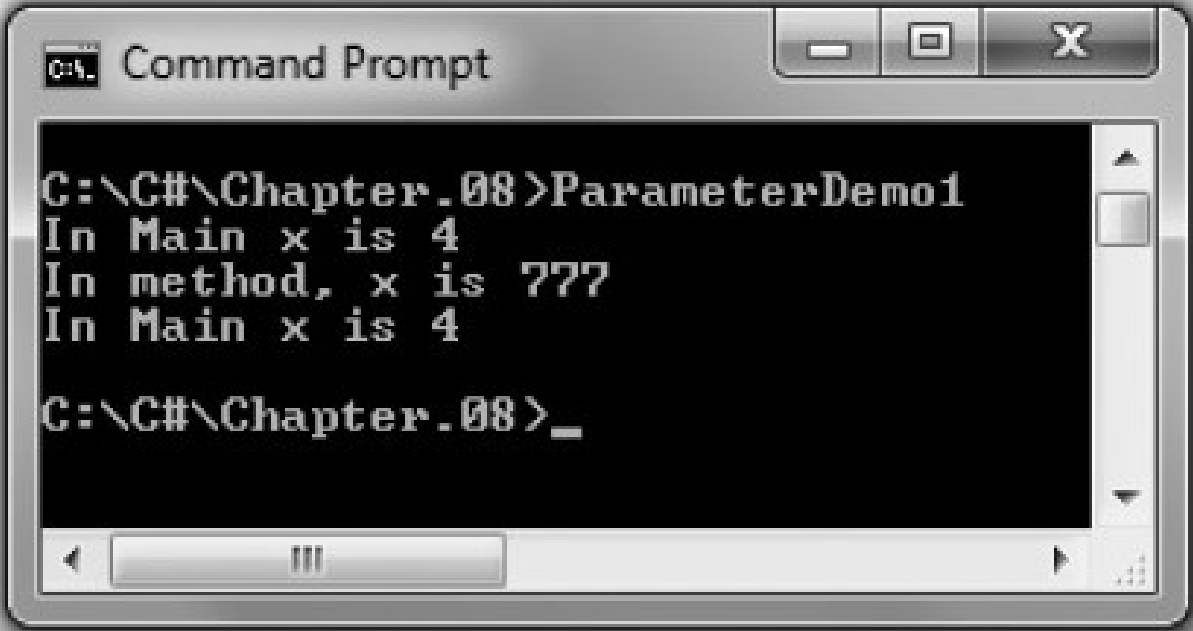


Figure 8-1 Program calling a method with a value parameter

Using Mandatory Value Parameters (cont'd.)



```
C:\C#\Chapter.08>ParameterDemo1
In Main x is 4
In method, x is 777
In Main x is 4

C:\C#\Chapter.08>_
```

Figure 8-2 Output of the ParameterDemo1 program

Using Reference Parameters, Output Parameters, and Parameter Arrays

- **Reference parameters and output parameters**
 - Have memory addresses that are passed to a method, allowing the method to alter the original variables
- **Differences**
 - Reference parameters need to contain a value before calling the method
 - Output parameters do not need to contain a value
- Reference and output parameters act as **aliases** for the same memory location occupied by the original passed variable

Using a `ref` Parameter

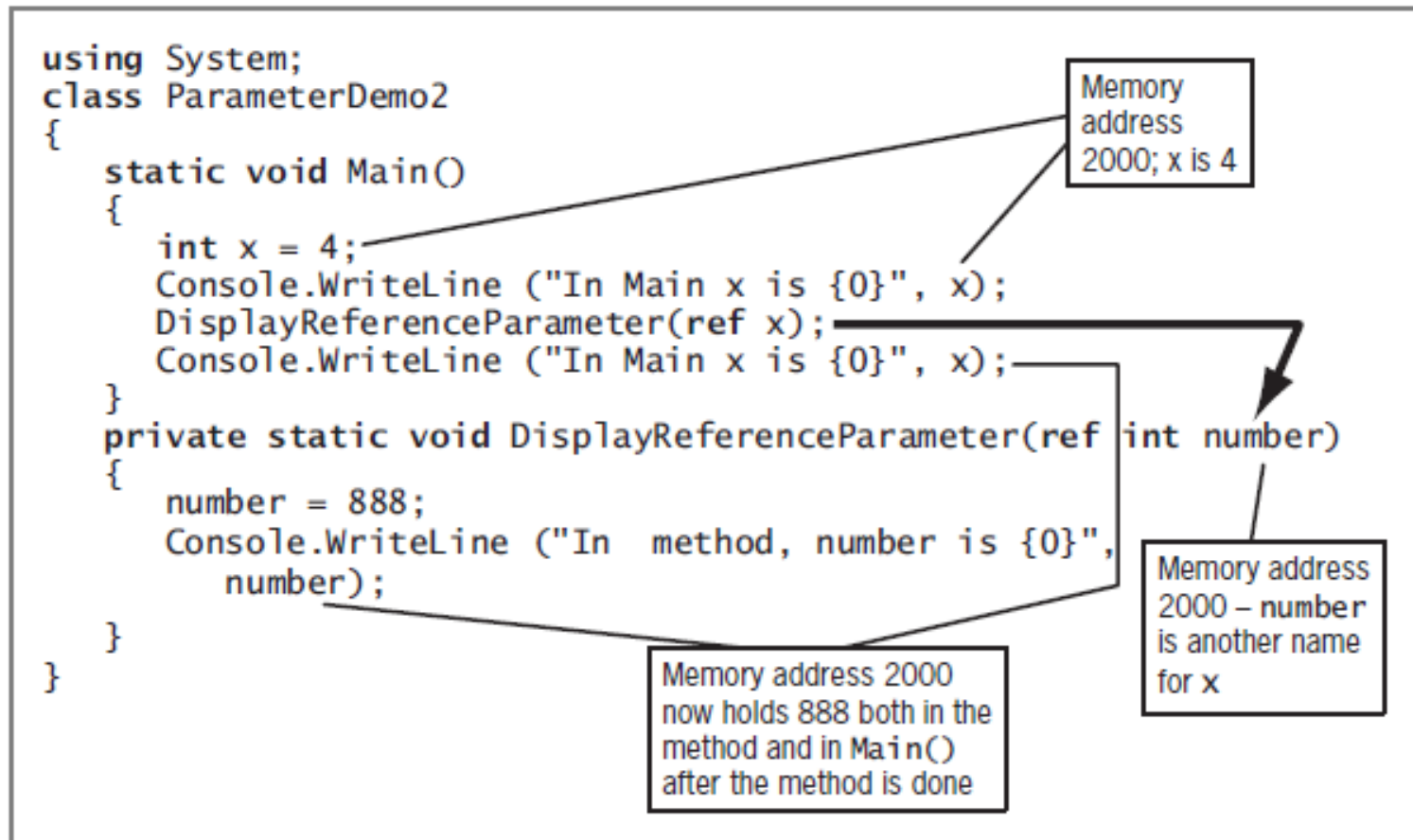
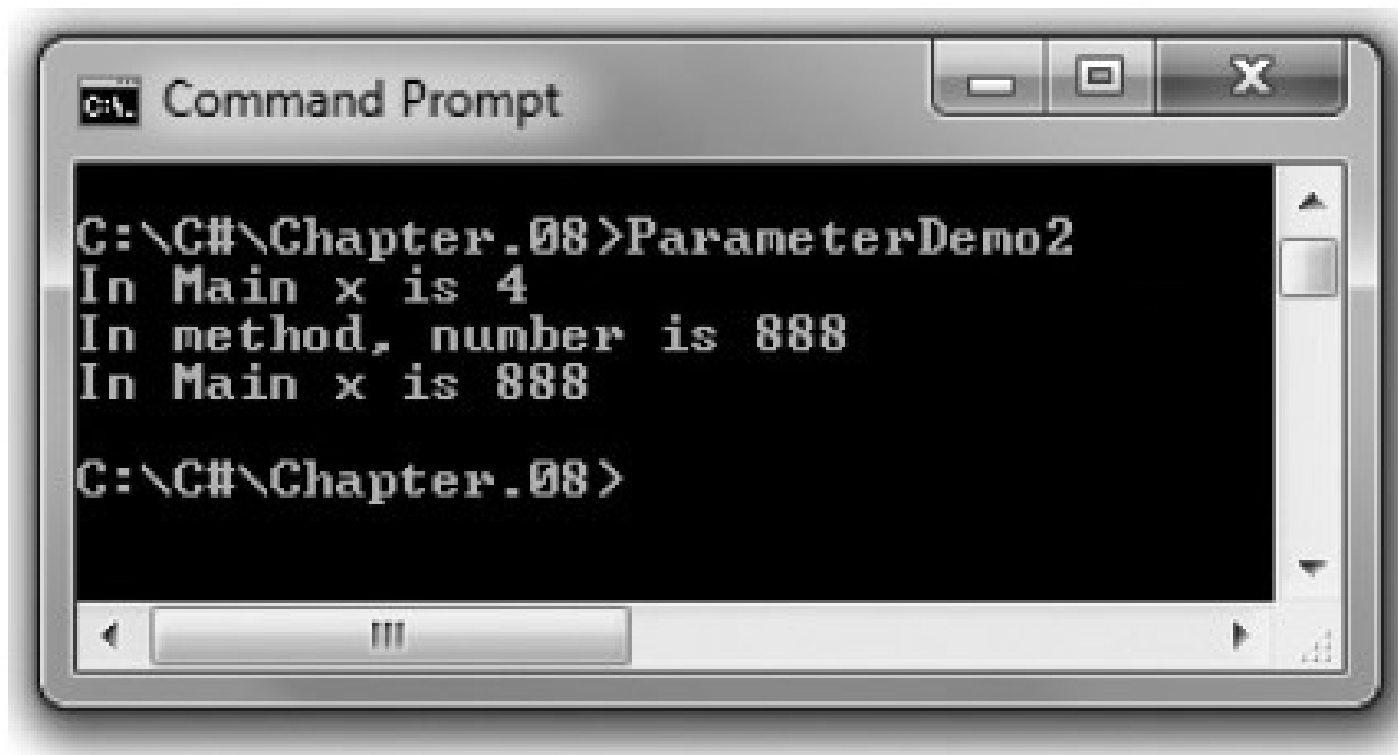


Figure 8-3 Program calling a method with a reference parameter

Using a `ref` Parameter (cont'd.)



```
C:\C#\Chapter.08>ParameterDemo2
In Main x is 4
In method, number is 888
In Main x is 888

C:\C#\Chapter.08>
```

The screenshot shows a Windows Command Prompt window titled "Command Prompt". The window has a standard Windows interface with minimize, maximize, and close buttons in the top right corner. The command prompt shows the current directory as "C:\C#\Chapter.08". The user has entered the command "ParameterDemo2", and the program has executed, displaying the following output: "In Main x is 4", "In method, number is 888", and "In Main x is 888". The prompt then returns to "C:\C#\Chapter.08>".

Figure 8-4 Output of the `ParameterDemo2` program

Using an out Parameter

```
using System;
class InputMethodDemo
{
    static void Main()
    {
        int first, second;
        InputMethod(out first, out second);
        Console.WriteLine("After InputMethod first is {0}", first);
        Console.WriteLine("and second is {0}", second);
    }
    private static void InputMethod(out int one, out int two)
    {
        string s1, s2;
        Console.Write("Enter first integer ");
        s1 = Console.ReadLine();
        Console.Write("Enter second integer ");
        s2 = Console.ReadLine();
        one = Convert.ToInt32(s1);
        two = Convert.ToInt32(s2);
    }
}
```

Notice the keyword out.

Notice the keyword out.

Figure 8-5 The InputMethodDemo program

Using an out Parameter (cont'd.)



```
Command Prompt

C:\C#\Chapter.08>InputMethodDemo
Enter first integer 23
Enter second integer 45
After InputMethod first is 23
and second is 45

C:\C#\Chapter.08>
```

Figure 8-6 Output of the `InputMethodDemo` program



Using Reference and Output Parameters

- The advantage of using reference and output parameters:
 - A method can change multiple variables
- The disadvantage of using reference and output parameters:
 - They allow multiple methods to have access to the same data, weakening the “black box” paradigm



Using the TryParse () Methods

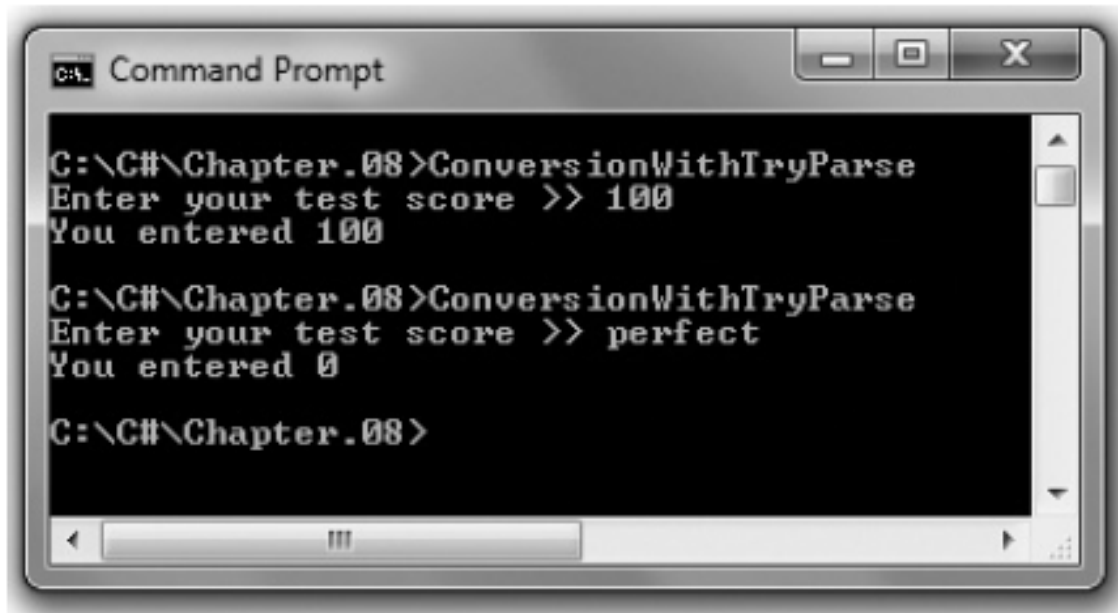
- The TryParse () methods are an alternative to the Convert () and Parse () methods from Chapter 2
- TryParse () requires two parameters:
 - The value to be converted
 - An out parameter to hold the result
- TryParse () returns a Boolean value

Using the TryParse () Methods (cont'd.)

```
using System;
class ConversionWithTryParse
{
    static void Main()
    {
        string entryString;
        int score;
        Console.Write("Enter your test score >> ");
        entryString = Console.ReadLine();
        int.TryParse(entryString, out score);
        Console.WriteLine("You entered {0}", score);
    }
}
```

Figure 8-9 The ConversionWithTryParse program

Using the TryParse () Methods (cont'd.)



```
C:\C#\Chapter.08>ConversionWithTryParse
Enter your test score >> 100
You entered 100

C:\C#\Chapter.08>ConversionWithTryParse
Enter your test score >> perfect
You entered 0

C:\C#\Chapter.08>
```

Figure 8-10 Two typical executions of the ConversionWithTryParse program



Using Parameter Arrays

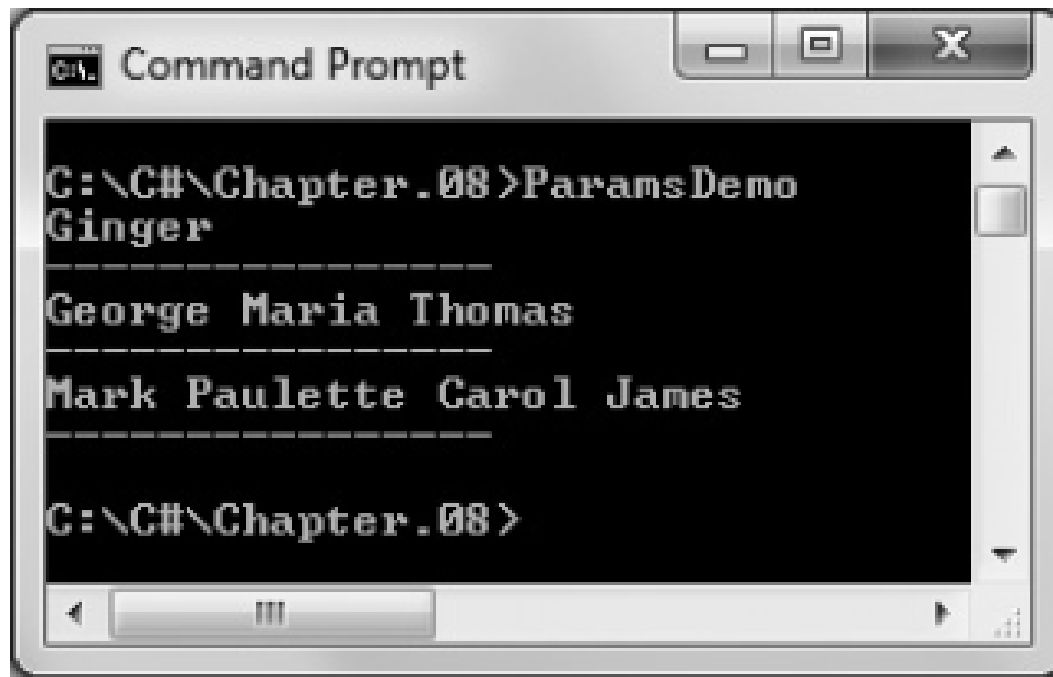
- **Parameter array**
 - A local array declared within the method header by using the keyword `params`
 - Used when you don't know how many arguments of the same type you might eventually send to a method
- No additional parameters are permitted after the `params` keyword
- Only one `params` keyword is permitted in a method declaration

Using Parameter Arrays (cont'd.)

```
using System;
class ParamsDemo
{
    static void Main()
    {
        string[] names = {"Mark", "Paulette", "Carol", "James"};
        DisplayStrings("Ginger");
        DisplayStrings("George", "Maria", "Thomas");
        DisplayStrings(names);
    }
    private static void DisplayStrings(params string[] people)
    {
        foreach(string person in people)
            Console.Write("{0} ", person);
        Console.WriteLine("\n-----");
    }
}
```

Figure 8-11 The ParamsDemo program

Using Parameter Arrays (cont'd.)



```
C:\C#\Chapter.08>ParamsDemo
Ginger
-----
George Maria Thomas
-----
Mark Paulette Carol James
-----
C:\C#\Chapter.08>
```

The screenshot shows a Windows Command Prompt window titled "Command Prompt". The command prompt is at the directory "C:\C#\Chapter.08". The user has entered the command "ParamsDemo". The program has executed and produced the following output: "Ginger", followed by a line of dashes, then "George Maria Thomas", followed by another line of dashes, then "Mark Paulette Carol James", followed by a third line of dashes. The prompt "C:\C#\Chapter.08>" is shown again at the bottom, indicating the program has finished execution.

Figure 8-12 Output of the ParamsDemo program



Overloading Methods

- **Overloading**
 - Involves using one term to indicate diverse meanings
- When you overload a C# method:
 - You write multiple methods with a shared name
 - The compiler understands your meaning based on the arguments you use with the method
- Methods are overloaded correctly when they have the same identifier but different parameter lists



Understanding Overload Resolution

- **Overload resolution**
 - Used by C# to determine which method to execute when a method call could execute multiple overloaded methods
- **Applicable methods**
 - A set of methods that can accept a call with a specific list of arguments
- **Betterness rules**
 - Rules that determine which method version to call
 - Similar to the implicit data type conversion rules

Understanding Overload Resolution (cont'd.)

Data Type	Conversions Are Better in This Order
byte	short, ushort, int, uint, long, ulong, float, double, decimal
sbyte	short, int, long, float, double, decimal
short	int, long, float, double, decimal
ushort	int, uint, long, ulong, float, double, decimal
int	long, float, double, decimal
uint	long, ulong, float, double, decimal
long	float, double, decimal
ulong	float, double, decimal
float	double
char	ushort, int, uint, long, ulong, float, double, decimal

Table 8-1 Betterness rules for data type conversion



Avoiding Ambiguous Methods

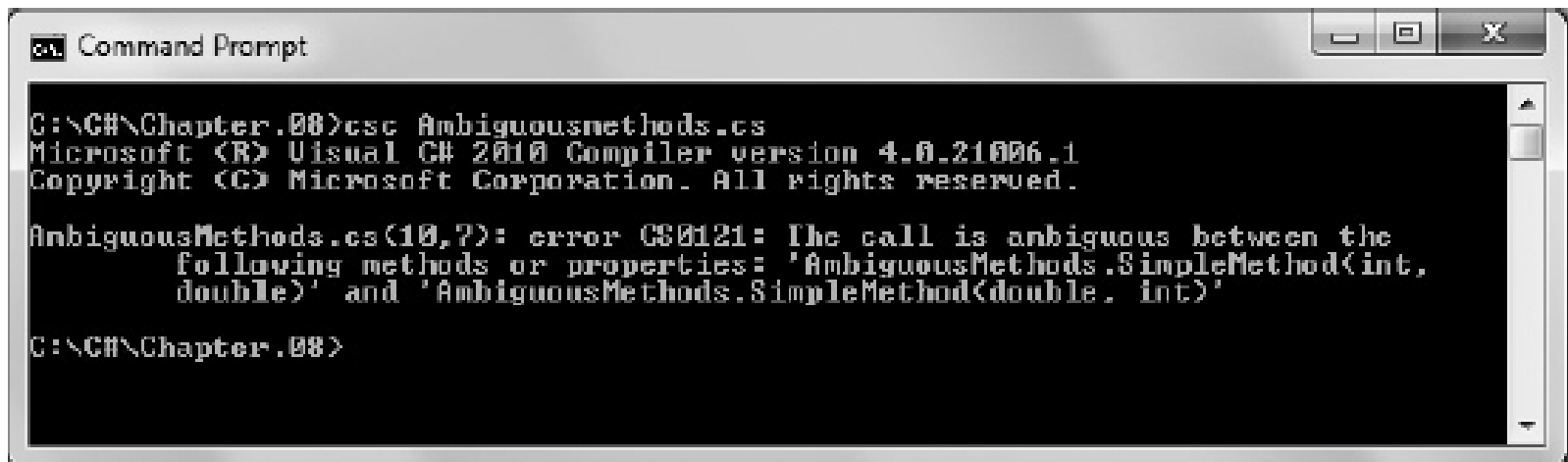
- **Ambiguous** method
 - A situation in which the compiler cannot determine which method to use
 - Occurs when you overload methods
- Methods with identical names that have identical parameter lists but different return types are not overloaded

Avoiding Ambiguous Methods (cont'd.)

```
using System;
class AmbiguousMethods
{
    static void Main()
    {
        int iNum = 20;
        double dNum = 4.5;
        SimpleMethod(iNum, dNum); // calls first version
        SimpleMethod(dNum, iNum); // calls second version
        SimpleMethod(iNum, iNum); // error! Call is ambiguous.
    }
    private static void SimpleMethod(int i, double d)
    {
        Console.WriteLine("Method receives int and double");
    }
    private static void SimpleMethod(double d, int i)
    {
        Console.WriteLine("Method receives double and int");
    }
}
```

Figure 8-21 Program containing an ambiguous method call

Avoiding Ambiguous Methods (cont'd.)



```
Command Prompt

C:\CH\Chapter.00>csc AmbiguousMethods.cs
Microsoft (R) Visual C# 2010 Compiler version 4.0.21006.1
Copyright (C) Microsoft Corporation. All rights reserved.

AmbiguousMethods.cs(10,7): error CS0121: The call is ambiguous between the
    following methods or properties: 'AmbiguousMethods.SimpleMethod(int,
    double)' and 'AmbiguousMethods.SimpleMethod(double, int)'

C:\CH\Chapter.00>
```

Figure 8-22 Error message generated by an ambiguous method call



Using Optional Parameters

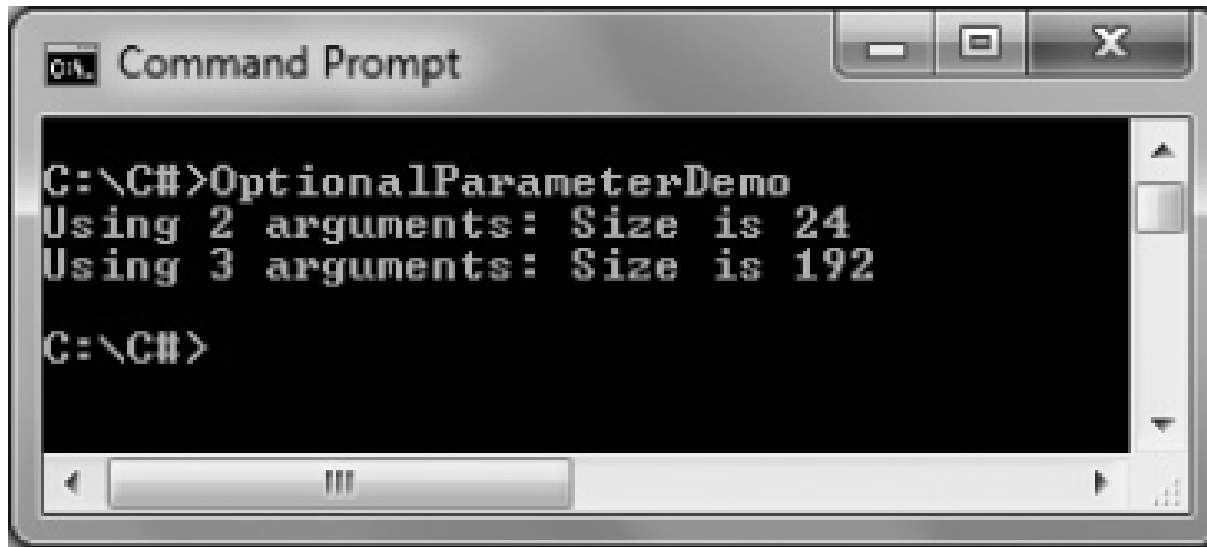
- **Optional parameter**
 - One for which a default value is automatically supplied
- Make a parameter optional by providing a value for it in the method declaration
 - Only value parameters can be given default values
- Any optional parameters in a parameter list must follow all mandatory parameters

Using Optional Parameters (cont'd.)

```
using System;
class OptionalParameterDemo
{
    static void Main()
    {
        Console.Write("Using 2 arguments: ");
        DisplaySize(4, 6);
        Console.Write("Using 3 arguments: ");
        DisplaySize(4, 6, 8);
    }
    private static void DisplaySize(int length, int width, int height = 1)
    {
        int area = length * width * height;
        Console.WriteLine("Size is {0}", area);
    }
}
```

Figure 8-23 The OptionalParameterDemo class

Using Optional Parameters (cont'd.)



```
C:\C#>OptionalParameterDemo
Using 2 arguments: Size is 24
Using 3 arguments: Size is 192
C:\C#>
```

Figure 8-24 Execution of the `OptionalParameterDemo` program

Using Optional Parameters (cont'd.)

Method Declaration	Explanation
<code>private static void M1(int a, int b, int c, int d = 10)</code>	Valid. The first three parameters are mandatory and the last one is optional.
<code>private static void M2(int a, int b = 3, int c)</code>	Invalid. Because b has a default value, c must also have one.
<code>private static void M3(int a = 3, int b = 4, int c = 5)</code>	Valid. All parameters are optional.
<code>private static void M4(int a, int b, int c)</code>	Valid. All parameters are mandatory.
<code>private static void M5(int a = 4, int b, int c = 8)</code>	Invalid. Because a has a default value, both b and c must have default values.

Table 8-2 Examples of valid and invalid optional parameter method declarations



Leaving Out Unnamed Arguments

- When calling a method with optional parameters and you are using unnamed arguments, leave out any arguments to the right of the last one you use
- Example method:

```
private static void Method1(int a, char b,  
int c = 22, double d = 33.2)
```

Call to Method1()	Explanation
<code>Method1(1, 'A', 3, 4.4);</code>	Valid. The four arguments are assigned to the four parameters.
<code>Method1(1, 'K', 9);</code>	Valid. The three arguments are assigned to a, b, and c in the method, and the default value of 33.2 is used for d.
<code>Method1(5, 'D');</code>	Valid. The two arguments are assigned to a and b in the method, and the default values of 22 and 33.2 are used for c and d, respectively.
<code>Method1(1);</code>	Invalid. <code>Method1()</code> requires at least two arguments for the first two parameters.
<code>Method1();</code>	Invalid. <code>Method1()</code> requires at least two arguments for the first two parameters.
<code>Method1(3, 18.5);</code>	Invalid. The first argument, 3, can be assigned to a, but the second argument must be type char.
<code>Method1(4, 'R', 55.5);</code>	Invalid. The first argument, 4, can be assigned to a, and the second argument, 'R', can be assigned to b, but the third argument must be type int. When arguments are unnamed, you cannot "skip" parameter c, use its default value, and assign 55.5 to parameter d.

Table 8-3 Examples of legal and illegal calls to `Method1()`



Using Named Arguments

- Leave out optional arguments in a method call if you pass the remaining arguments by name
- Named arguments appear in any order
 - But must appear after all the unnamed arguments have been listed
- Name an argument using its parameter name and a colon before the value
- Example method:

```
private static void Method1(int a, char b,  
int c = 22, double d = 33.2)
```

Call to Method2()	Explanation
<code>Method2(1, 'A');</code>	Valid. The two arguments are assigned to a and b in the method, and the default values of 22 and 33.2 are used for c and d, respectively.
<code>Method2(2, 'E', 3);</code>	Valid. The three arguments are assigned to a, b, and c. The default value 33.2 is used for d.
<code>Method2(2, 'E', c : 3);</code>	Valid. This call is identical to the one above.
<code>Method2(1, 'K', d : 88.8);</code>	Valid. The first two arguments are assigned to a and b. The default value 22 is used for c. The named value 88.8 is used for d.
<code>Method2(d : 2.1, b : 'A', c : 88, a: 12);</code>	Valid. All the arguments are assigned to parameters whether they are listed in order or not.
<code>Method2(5, 'S', d : 7.4, c: 9);</code>	Valid. The first two arguments are assigned to a and b. Even though the values for c and d are not listed in order, they are assigned correctly.
<code>Method2(d : 11.1, 6, 'P');</code>	Invalid. This call contains an unnamed argument after the first named argument. Named arguments must appear after all unnamed arguments.

Table 8-4 Examples of legal and illegal calls to Method2()



Advantages to Using Named Arguments

- Added flexibility with method calls
 - Named arguments can be coded in any order as long as they follow all the unnamed arguments
- Can result in fewer overloaded methods
- Can aid in self-documentation

Advantages to Using Named Arguments (cont'd.)

Overloaded implementations of Closing()

```
private static void Closing()
{
    Console.WriteLine("Sincerely,");
    Console.WriteLine("James O'Hara");
}
private static void Closing(string name)
{
    Console.WriteLine("Sincerely,");
    Console.WriteLine(name);
}
```

Single implementation of Closing() with optional parameter

```
private static void Closing(string name = "James O'Hara")
{
    Console.WriteLine("Sincerely,");
    Console.WriteLine(name);
}
```

Figure 8-25 Two ways to implement Closing() to accept a name parameter or not



Disadvantages to Using Named Arguments

- Implementation hiding is compromised
- Changes to the called method could cause problems and require changes to the calling method or program

Disadvantages to Using Named Arguments (cont'd.)

```
private static double ComputeGross(double hours, double rate, out double bonus)
{
    double gross = hours * rate;
    if(hours >= 40)
        bonus = 100;
    else
        bonus = 50;
    return gross;
}
private static double ComputeTotalPay(double gross, double bonus)
{
    double total = gross + bonus;
    return total;
}
```


Figure 8-26 Two payroll program methods

Disadvantages to Using Named Arguments (cont'd.)

```
static void Main()
{
    double hours = 40;
    double rate = 10.00;
    double bonus = 0;
    double totalPay;
    totalPay = ComputeTotalPay(ComputeGross(hours, rate, out bonus),
        bonus);
    Console.WriteLine("Total pay is {0}", totalPay);
}
```

Figure 8-27 A Main() method that calls ComputeTotalPay() using positional arguments

Disadvantages to Using Named Arguments (cont'd.)



```
Command Prompt
C:\C#\Chapter.08>Payroll1
Total pay is 500
C:\C#\Chapter.08>
```

Figure 8-28 Execution of the `Main()` method in Figure 8-27

Disadvantages to Using Named Arguments (cont'd.)

```
static void Main()
{
    double hours = 40;
    double rate = 10.00;
    double bonus = 0;
    double totalPay;
    totalPay = ComputeTotalPay(bonus: bonus,
                              gross: ComputeGross(hours, rate, out bonus));
    Console.WriteLine("Total pay is {0}", totalPay);
}
```

Figure 8-29 A `Main()` method that calls `ComputeTotalPay()` using named arguments

Disadvantages to Using Named Arguments (cont'd.)

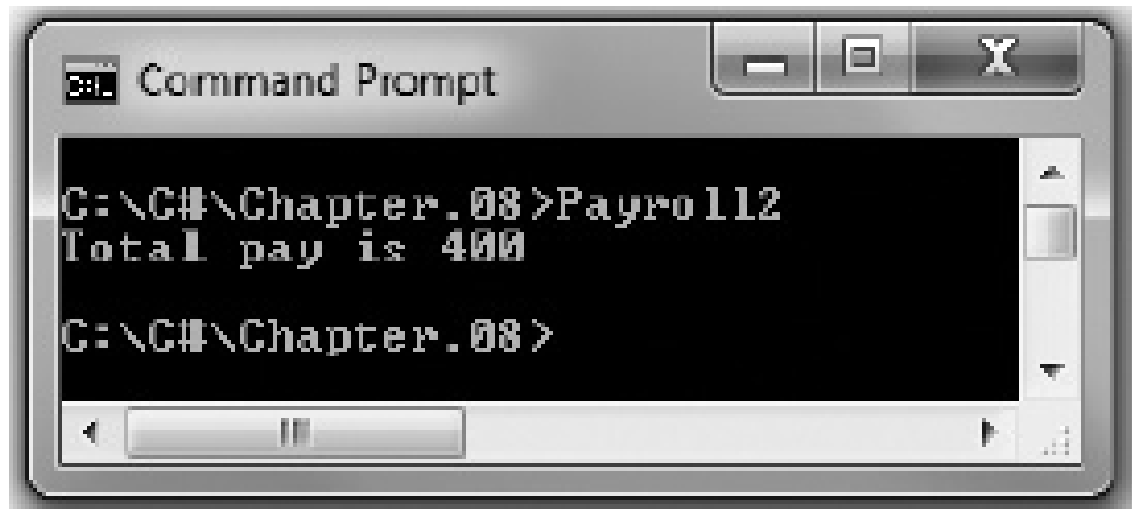


Figure 8-30 Execution of the `Main()` method in Figure 8-29



Overload Resolution with Named and Optional Arguments

- Named and optional arguments affect overload resolution
 - Rules for betterness on argument conversions are applied only for arguments that are given explicitly
- If two signatures are equally good, the signature that does not omit optional parameters is considered better



Summary

- Method parameters can be mandatory or optional
- Types of formal parameters:
 - Value parameters
 - Reference parameters
 - Output parameters
 - Parameter arrays
- When you overload a C# method, you write multiple methods with a shared name but different argument lists



Summary (cont'd.)

- When you overload a method, you run the risk of creating an ambiguous situation
- An optional parameter to a method is one for which a default value is automatically supplied if you do not explicitly send one as an argument