COIS1020H: Programming for Computing Systems

Chapter 8
Advanced Method Concepts

Introduction to Recursion

- We have been calling other methods from a method.
- It's also possible for a method to call itself.
- A method that calls itself is a *recursive method*.

- Each time it calls itself, the cycle is repeated endlessly.
- Like a loop (more on that later), a recursive method must have some way to control the number of times it repeats.

RecursionDemo.cs

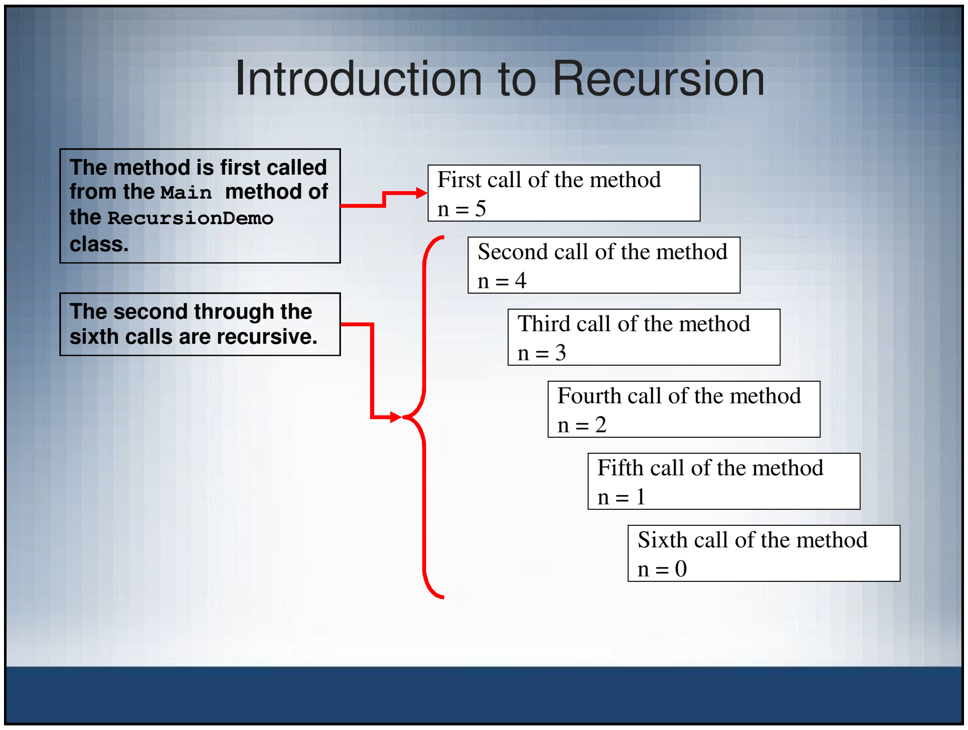
```
// This class demonstrates the Recursive.message method
using System;

public class RecursionDemo
{
    public static void Main()
    {
        Message(5);

        Console.ReadLine();
    }
```

RecursionDemo.cs (cont'd)

```
    // Message is a recursive method, which displays a message n times.
    public static void Message(int n)
    {
        if (n > 0)
        {
            Console.WriteLine("This is a recursive method.");
            Message(n - 1);
        }
    }
}
```

Output:

```
This is a recursive method
This is a recursive method
This is a recursive method
This is a recursive method
This is a recursive method
```



Introduction to Recursion

The method is first called from the `Main` method of the `RecursionDemo` class.

The second through the sixth calls are recursive.

First call of the method
n = 5

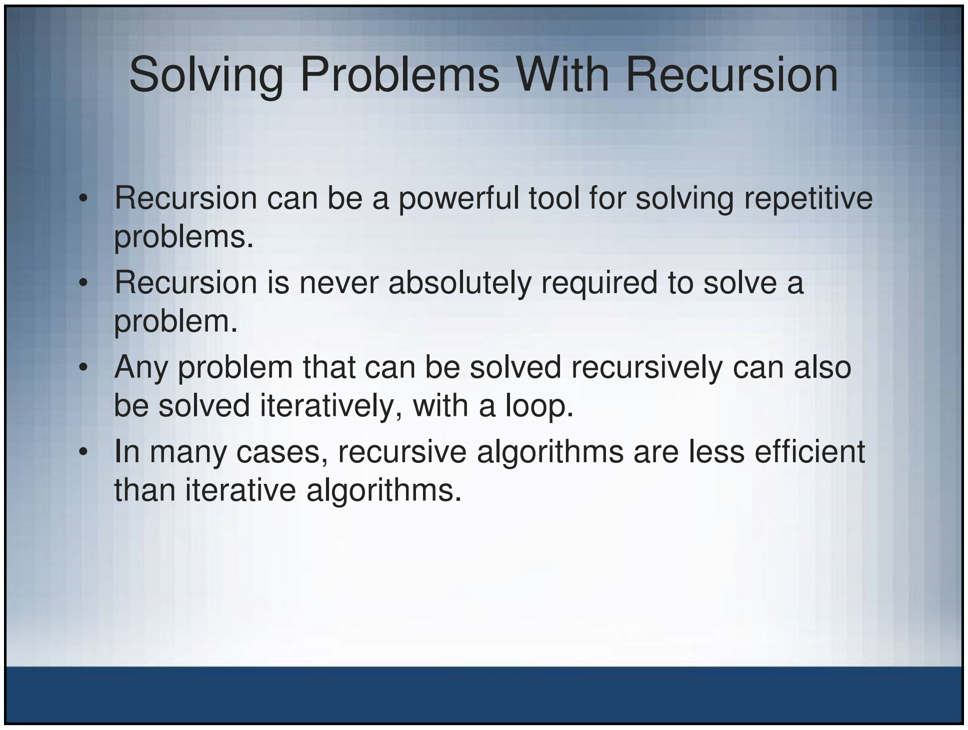
Second call of the method
n = 4

Third call of the method
n = 3

Fourth call of the method
n = 2

Fifth call of the method
n = 1

Sixth call of the method
n = 0



Solving Problems With Recursion

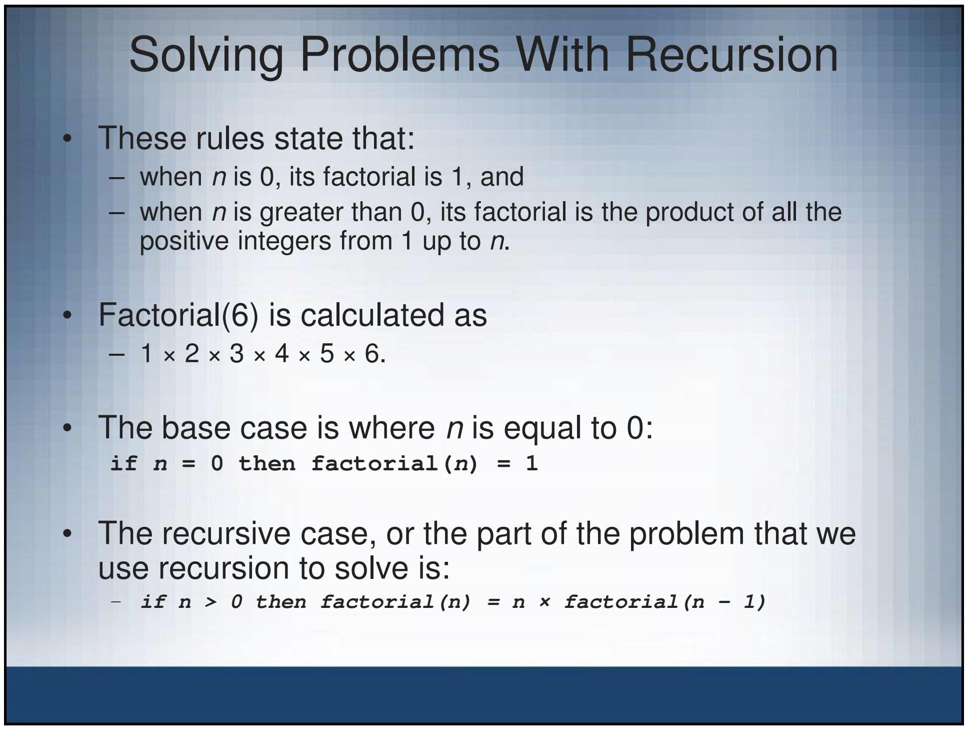
- Recursion can be a powerful tool for solving repetitive problems.
- Recursion is never absolutely required to solve a problem.
- Any problem that can be solved recursively can also be solved iteratively, with a loop.
- In many cases, recursive algorithms are less efficient than iterative algorithms.

Solving Problems With Recursion

- Recursion works like this:
  - A base case is established.
    - If matched, the method solves it and returns.
  - If the base case cannot be solved now:
    - the method reduces it to a smaller problem (recursive case) and calls itself to solve the smaller problem.
- By reducing the problem with each recursive call, the base case will eventually be reached and the recursion will stop.
- In mathematics, the notation *n*! represents the factorial of the number *n*.

Solving Problems With Recursion

- The factorial of a nonnegative number can be defined by the following rules:
  - If $n = 0$ then $n! = 1$
  - If $n > 0$ then $n! = 1 \times 2 \times 3 \times \ldots \times n$
- Let's replace the notation *n*! with factorial(*n*), which looks a bit more like computer code, and rewrite these rules as:
  - If $n = 0$ then factorial(n) = 1
  - If $n > 0$ then factorial(n) = $1 \times 2 \times 3 \times \ldots \times n$



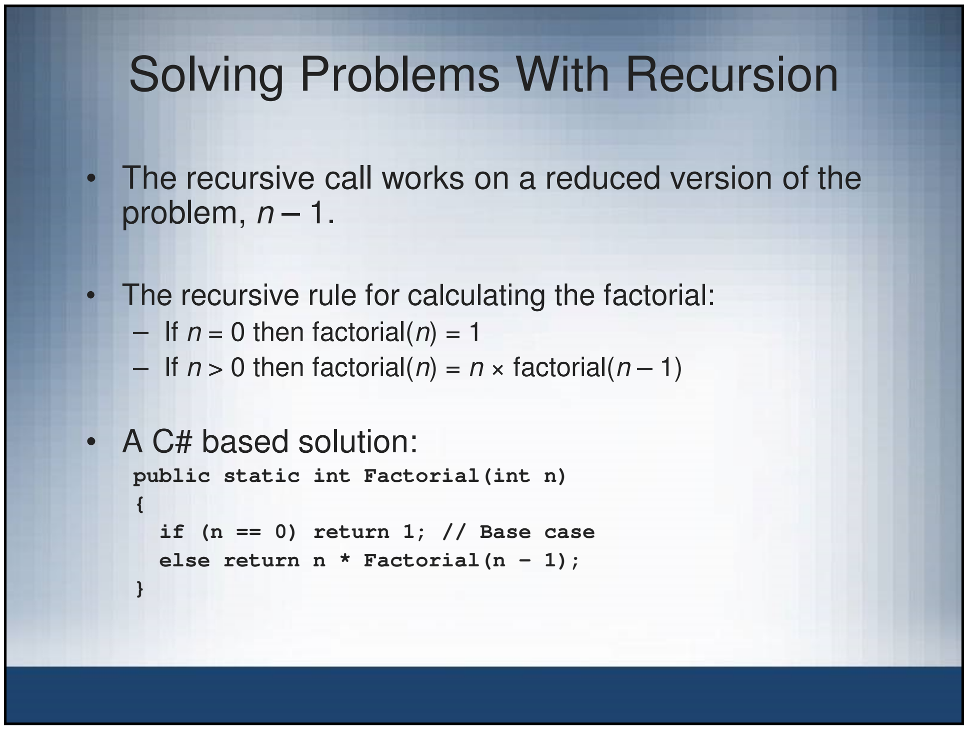
Solving Problems With Recursion

- These rules state that:
  - when *n* is 0, its factorial is 1, and
  - when *n* is greater than 0, its factorial is the product of all the positive integers from 1 up to *n*.
- Factorial(6) is calculated as
  - $1 \times 2 \times 3 \times 4 \times 5 \times 6$.
- The base case is where *n* is equal to 0:

```
if n = 0 then factorial(n) = 1
```

- The recursive case, or the part of the problem that we use recursion to solve is:

```
– if n > 0 then factorial(n) = n × factorial(n – 1)
```



Solving Problems With Recursion

- The recursive call works on a reduced version of the problem, $n - 1$.
- The recursive rule for calculating the factorial:
  - If $n = 0$ then factorial(n) = 1
  - If $n > 0$ then factorial(n) = $n \times$ factorial($n - 1$)
- A C# based solution:

```
public static int Factorial(int n)
{
  if (n == 0) return 1; // Base case
  else return n * Factorial(n – 1);
}
```

FactorialDemo.cs

```
using System;
public class FactorialDemo
{
    public static void Main()
    {
        long number;

        Console.Write("Enter a nonnegative integer: ");
        number = Convert.ToInt32(Console.ReadLine());

        Console.WriteLine("{0}! is {1}", number, Factorial(number));

        Console.ReadLine();
    }
```

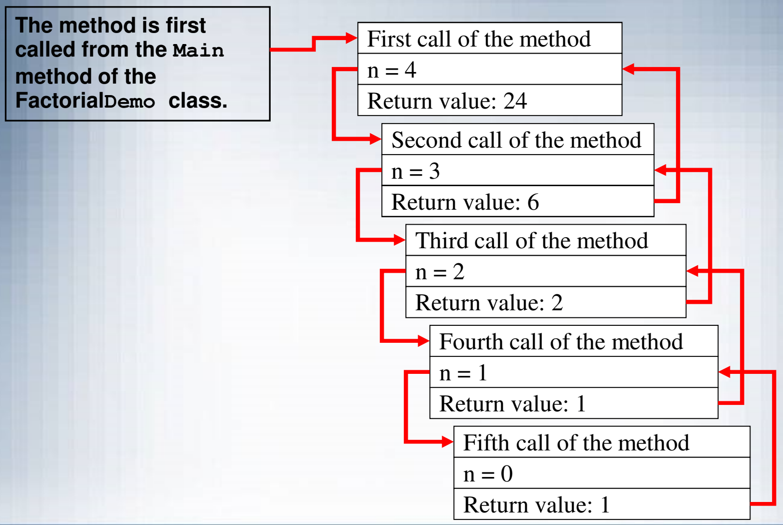
FactorialDemo.cs (cont'd)

```
    // The factorial method uses recursion to calculate
    // the factorial of its argument, which is assumed to be a nonnegative number.
    //   **param n The number to use in the calculation.
    //   **return The factorial of n.
    public static long Factorial(long n)
    {
      if (n == 0)
         return 1;                          // base case
      else
         return n * Factorial(n - 1);      //recursive case
    }
}
```

Output:

```
Enter a nonnegative integer: 4
4! is 24
```

Solving Problems With Recursion



Understanding Parameter Types

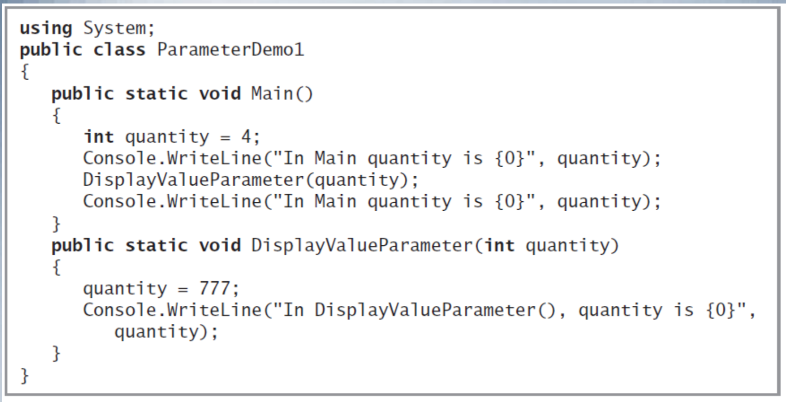
- **Mandatory parameter**
  - Argument for it is required in every method call
- Four types of mandatory parameters
  - Value parameters
    - Declared without any modifiers
  - Reference parameters
    - Declared with the `ref` modifier
  - Output parameters
    - Declared with the `out` modifier
  - Parameter arrays (not important for this course)
    - Declared with the `params` modifier

14

Using Mandatory Value Parameters

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

Using Value Parameters (cont'd.)



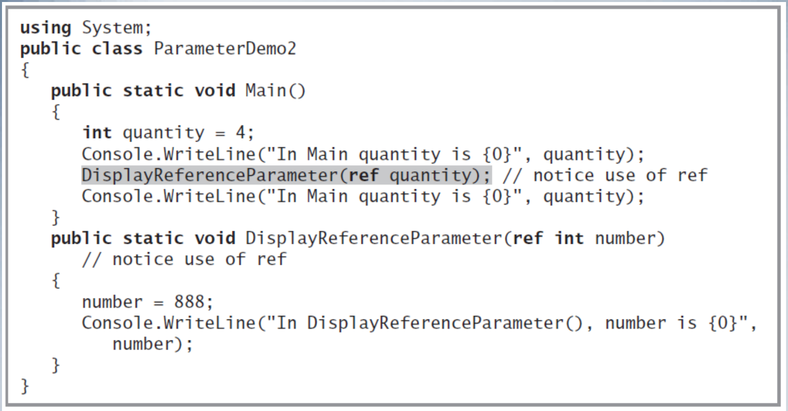
```
using System;
public class ParameterDemo1
{
   public static void Main()
   {
      int quantity = 4;
      Console.WriteLine("In Main quantity is {0}", quantity);
      DisplayValueParameter(quantity);
      Console.WriteLine("In Main quantity is {0}", quantity);
   }
   public static void DisplayValueParameter(int quantity)
   {
      quantity = 777;
      Console.WriteLine("In DisplayValueParameter(), quantity is {0}",
         quantity);
   }
}
```

Figure 8-1 Program calling method with a value parameter

Using Reference and Output Parameters

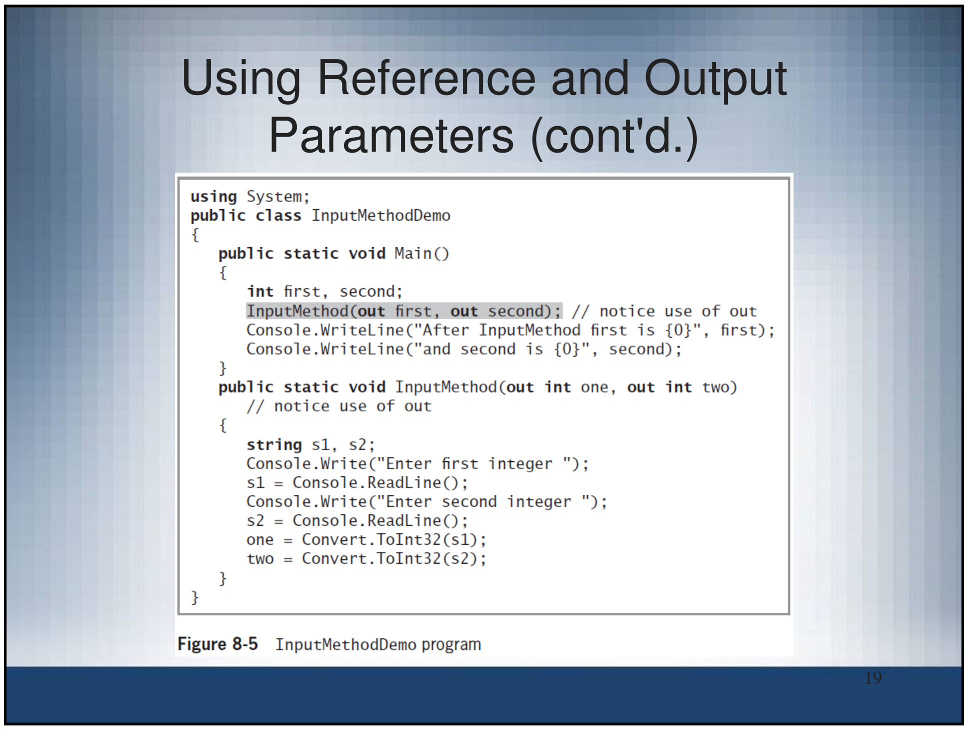
- **Reference** and **output** parameters
  - Have memory addresses that are passed to a method, allowing it to alter the original variables
- Differences
  - Reference parameters need to contain a value before calling the method
  - Output parameters do not need to contain a value
- Reference and output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

Using Reference and Output Parameters (cont'd.)



```
using System;
public class ParameterDemo2
{
   public static void Main()
   {
      int quantity = 4;
      Console.WriteLine("In Main quantity is {0}", quantity);
      DisplayReferenceParameter(ref quantity); // notice use of ref
      Console.WriteLine("In Main quantity is {0}", quantity);
   }
   public static void DisplayReferenceParameter(ref int number)
      // notice use of ref
   {
      number = 888;
      Console.WriteLine("In DisplayReferenceParameter(), number is {0}",
         number);
   }
}
```

Figure 8-3 Program calling method with a reference parameter

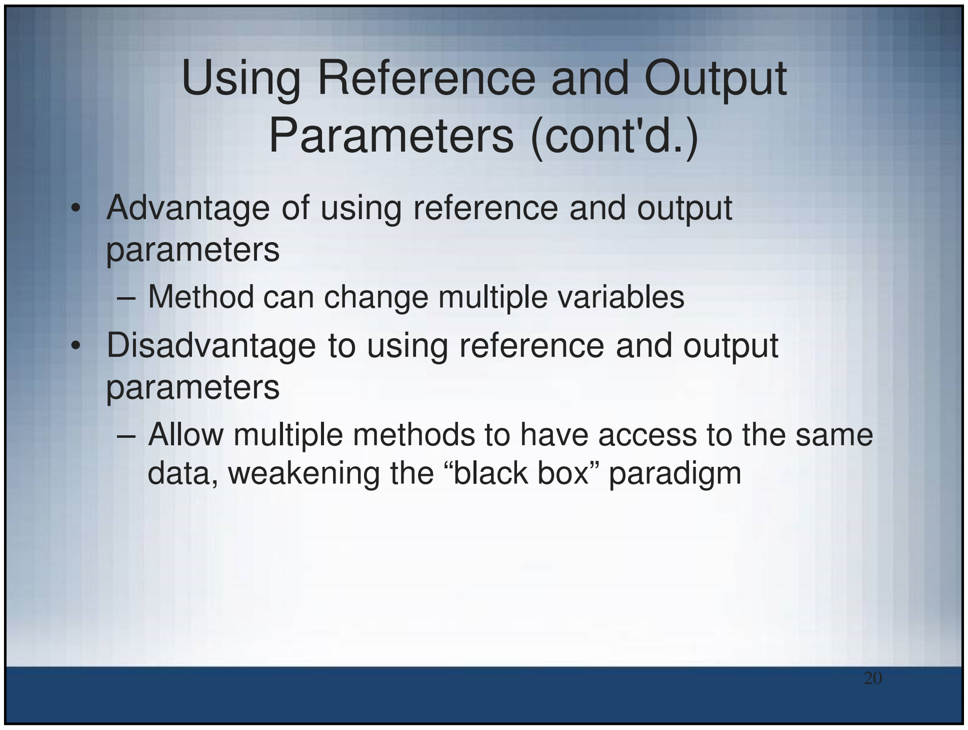


Using Reference and Output Parameters (cont'd.)

```
using System;
public class InputMethodDemo
{
   public static void Main()
   {
      int first, second;
      InputMethod(out first, out second); // notice use of out
      Console.WriteLine("After InputMethod first is {0}", first);
      Console.WriteLine("and second is {0}", second);
   }
   public static void InputMethod(out int one, out int two)
      // notice use of out
   {
      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);
   }
}
```

Figure 8-5 InputMethodDemo program

19



Using Reference and Output Parameters (cont'd.)

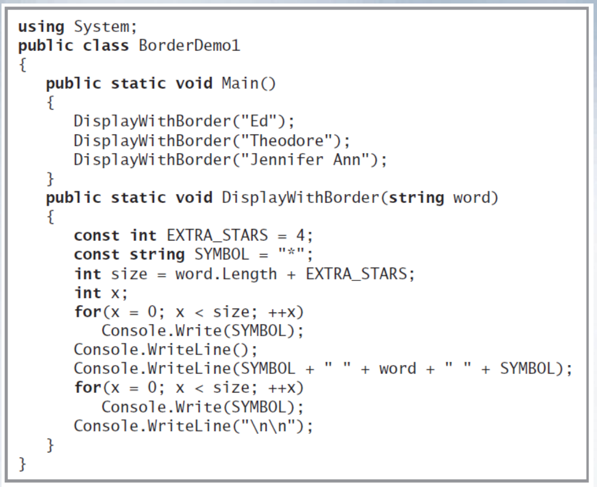
- Advantage of using reference and output parameters
  – Method can change multiple variables
- Disadvantage to using reference and output parameters
  – Allow multiple methods to have access to the same data, weakening the “black box” paradigm

20

Overloading Methods

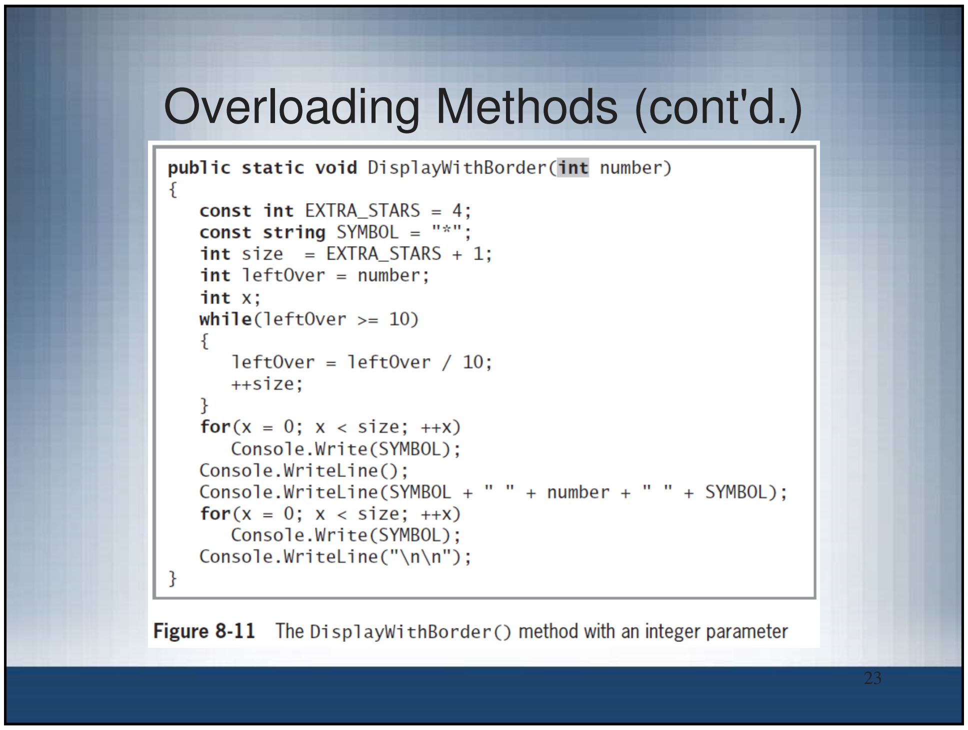
- **Overloading**
  - Involves using one term to indicate diverse meanings
- When you overload a C# method:
  - You write multiple methods with a shared name
  - Compiler understands your meaning based on the arguments you use with the method
- Methods are overloaded correctly when:
  - They have the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

Overloading Methods (cont'd.)



```
using System;
public class BorderDemo1
{
   public static void Main()
   {
      DisplayWithBorder("Ed");
      DisplayWithBorder("Theodore");
      DisplayWithBorder("Jennifer Ann");
   }
   public static void DisplayWithBorder(string word)
   {
      const int EXTRA_STARS = 4;
      const string SYMBOL = "*";
      int size = word.Length + EXTRA_STARS;
      int x;
      for(x = 0; x < size; ++x)
         Console.Write(SYMBOL);
      Console.WriteLine();
      Console.WriteLine(SYMBOL + " " + word + " " + SYMBOL);
      for(x = 0; x < size; ++x)
         Console.Write(SYMBOL);
      Console.WriteLine("\n\n");
   }
}
```

Figure 8-9 The BorderDemo1 program

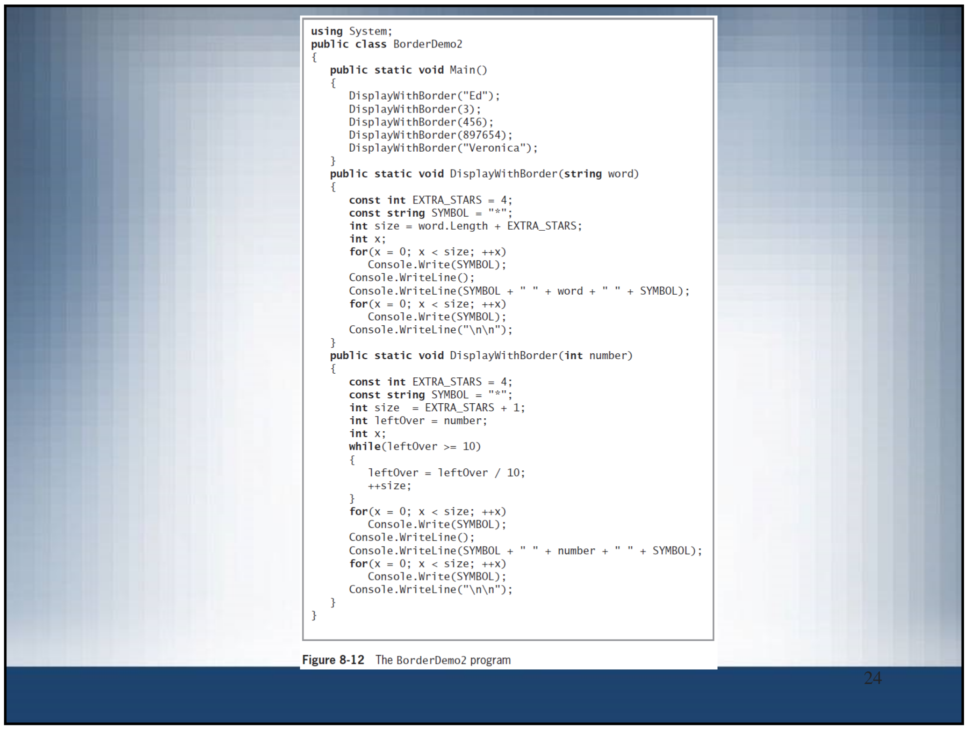


Overloading Methods (cont'd.)

```
public static void DisplayWithBorder(int number)
{
   const int EXTRA_STARS = 4;
   const string SYMBOL = "*";
   int size  = EXTRA_STARS + 1;
   int leftOver = number;
   int x;
   while(leftOver >= 10)
   {
      leftOver = leftOver / 10;
      ++size;
   }
   for(x = 0; x < size; ++x)
      Console.Write(SYMBOL);
   Console.WriteLine();
   Console.WriteLine(SYMBOL + " " + number + " " + SYMBOL);
   for(x = 0; x < size; ++x)
      Console.Write(SYMBOL);
   Console.WriteLine("\n\n");
}
```

Figure 8-11 The DisplayWithBorder() method with an integer parameter

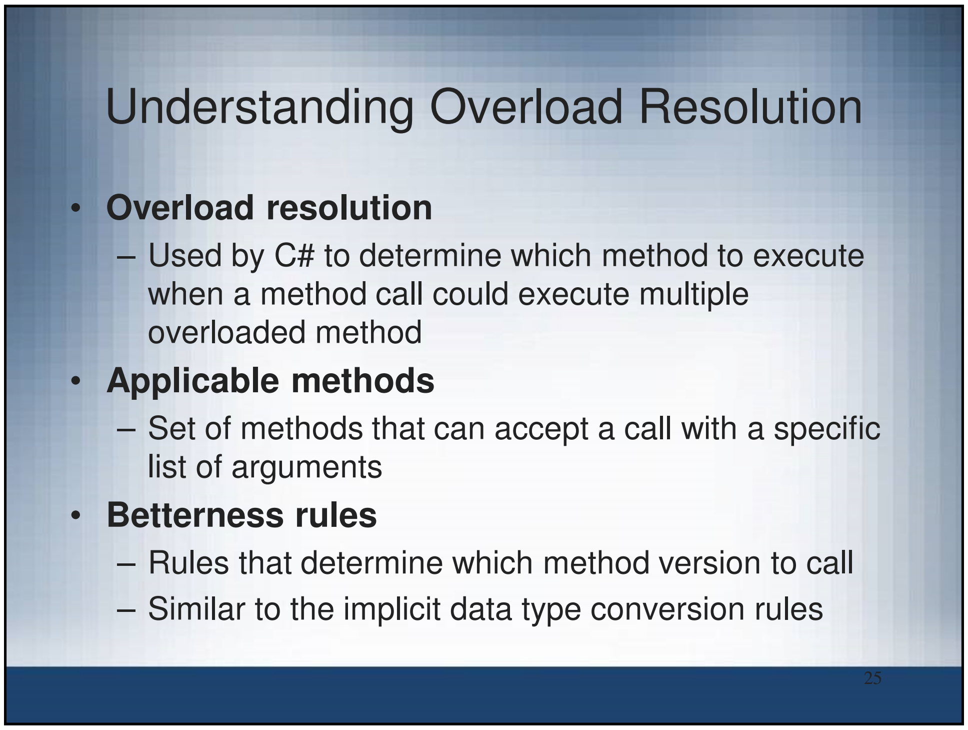
23



```
using System;
public class BorderDemo2
{
   public static void Main()
   {
      DisplayWithBorder("Ed");
      DisplayWithBorder(3);
      DisplayWithBorder(456);
      DisplayWithBorder(897654);
      DisplayWithBorder("Veronica");
   }
   public static void DisplayWithBorder(string word)
   {
      const int EXTRA_STARS = 4;
      const string SYMBOL = "*";
      int size = word.Length + EXTRA_STARS;
      int x;
      for(x = 0; x < size; ++x)
         Console.Write(SYMBOL);
      Console.WriteLine();
      Console.WriteLine(SYMBOL + " " + word + " " + SYMBOL);
      for(x = 0; x < size; ++x)
         Console.Write(SYMBOL);
      Console.WriteLine("\n\n");
   }
   public static void DisplayWithBorder(int number)
   {
      const int EXTRA_STARS = 4;
      const string SYMBOL = "*";
      int size  = EXTRA_STARS + 1;
      int leftOver = number;
      int x;
      while(leftOver >= 10)
      {
         leftOver = leftOver / 10;
         ++size;
      }
      for(x = 0; x < size; ++x)
         Console.Write(SYMBOL);
      Console.WriteLine();
      Console.WriteLine(SYMBOL + " " + number + " " + SYMBOL);
      for(x = 0; x < size; ++x)
         Console.Write(SYMBOL);
      Console.WriteLine("\n\n");
   }
}
```

Figure 8-12 The BorderDemo2 program

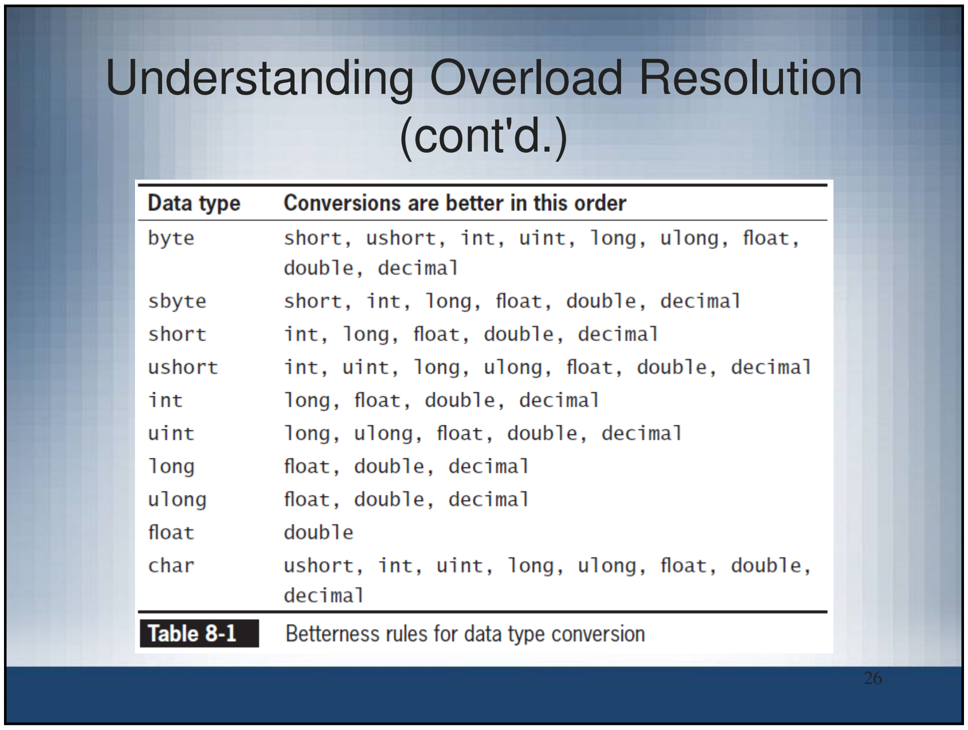
24



Understanding Overload Resolution

- **Overload resolution**
  - Used by C# to determine which method to execute when a method call could execute multiple overloaded method
- **Applicable methods**
  - 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

25



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

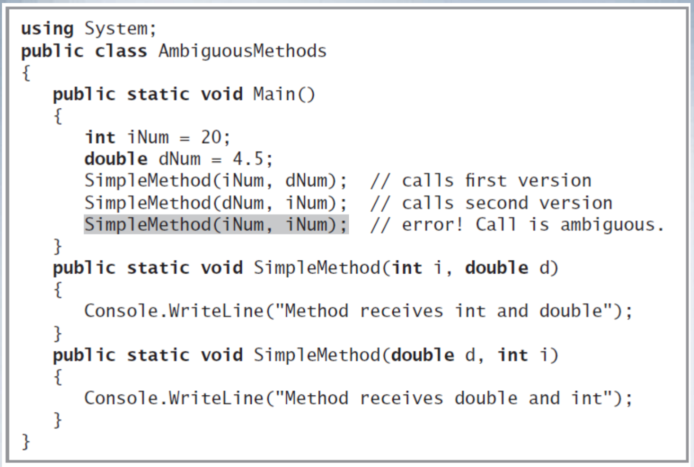
Table 8-1 Betterness rules for data type conversion

26

Avoiding Ambiguous Methods

- **Ambiguous method**
  - 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.)



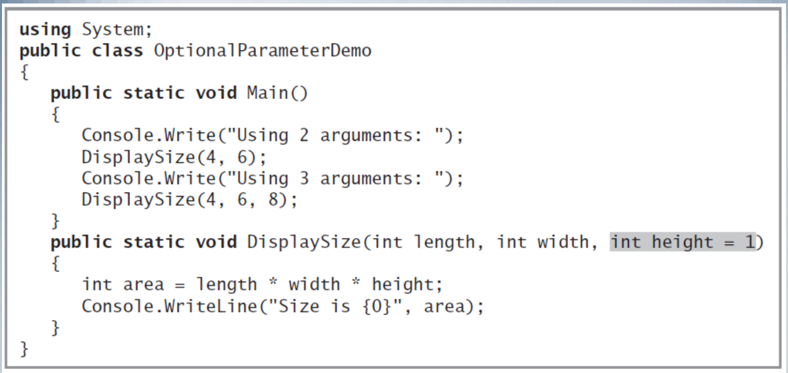
```
using System;
public class AmbiguousMethods
{
   public static void Main()
   {
      int iNum = 20;
      double dNum = 4.5;
      SimpleMethod(iNum, dNum);  // calls first version
      SimpleMethod(dNum, iNum);  // calls second version
      SimpleMethod(iNum, iNum);  // error! Call is ambiguous.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

Figure 8-15 Program containing 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

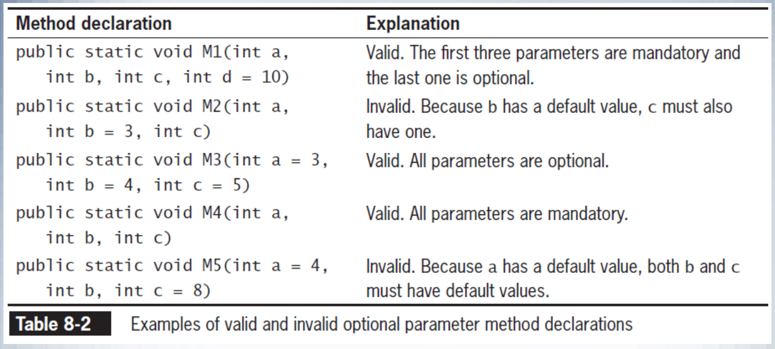
Using Optional Parameters (cont'd.)



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

Figure 8-17 The `OptionalParameterDemo` class

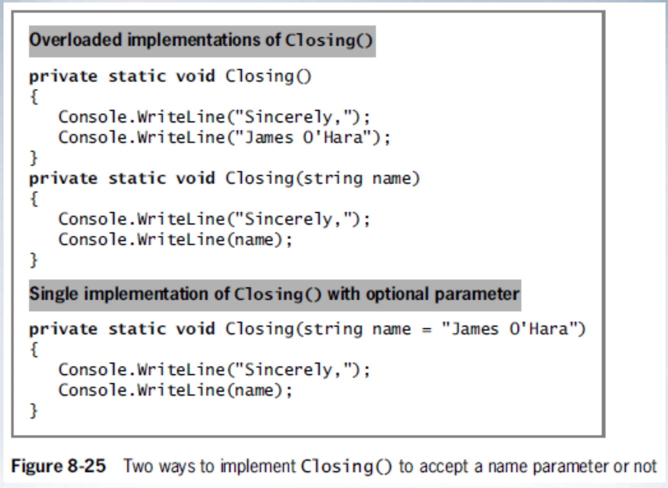
Using Optional Parameters (cont'd.)



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

Advantages to Using Optional Parameters



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

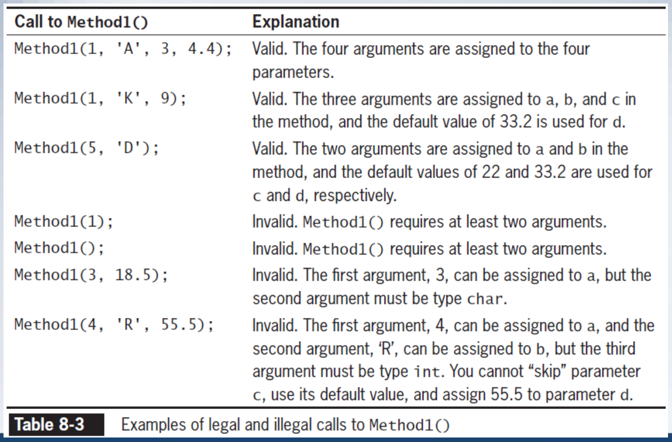
Leaving Out Unnamed Arguments

- 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

Leaving Out Unnamed Arguments

- Assume we have

public static void Method1(int a, char b, int c = 22, double d = 33,2)



Call to `Method1()`	Explanation
`Method1(1, 'A', 3, 4.4);`	Valid. The four arguments are assigned to the four parameters.
`Method1(1, 'K', 9);`	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`.
`Method1(5, 'D');`	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.
`Method1(1);`	Invalid. `Method1()` requires at least two arguments.
`Method1();`	Invalid. `Method1()` requires at least two arguments.
`Method1(3, 18.5);`	Invalid. The first argument, 3, can be assigned to `a`, but the second argument must be type `char`.
`Method1(4, 'R', 55.5);`	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`. 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

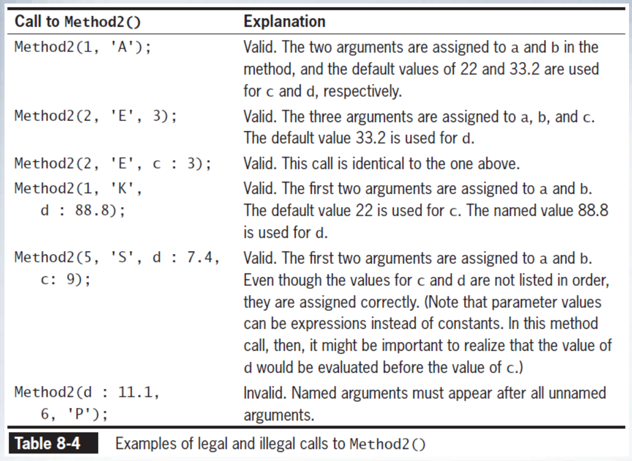
- You can leave out optional arguments in a method call if you pass the remaining arguments by name
- Named arguments appear in any order
  - But must appear after all the unnamed arguments have been listed
- Name an argument using its parameter name and a colon before the value

35

Using Named Arguments (cont'd.)

- Assume we have

  public static void Method2(int a, char b, int c = 22, double d = 33,2)



Call to `Method2()`	Explanation
`Method2(1, 'A');`	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.
`Method2(2, 'E', 3);`	Valid. The three arguments are assigned to a, b, and c. The default value 33.2 is used for d.
`Method2(2, 'E', c : 3);`	Valid. This call is identical to the one above.
`Method2(1, 'K', d : 88.8);`	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.
`Method2(5, 'S', d : 7.4, c: 9);`	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. (Note that parameter values can be expressions instead of constants. In this method call, then, it might be important to realize that the value of d would be evaluated before the value of c.)
`Method2(d : 11.1, 6, 'P');`	Invalid. Named arguments must appear after all unnamed arguments.

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

36

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

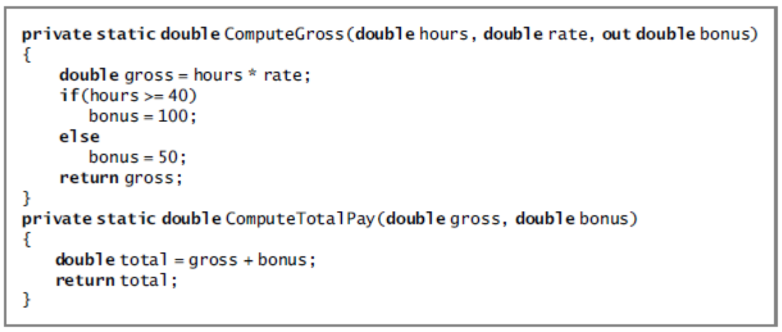
37

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

38

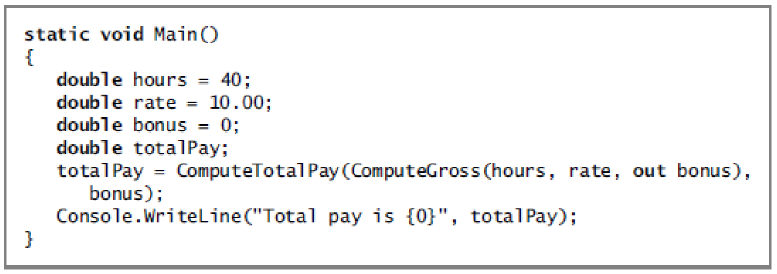
Disadvantages to Using Named Arguments (cont'd.)



```
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.)



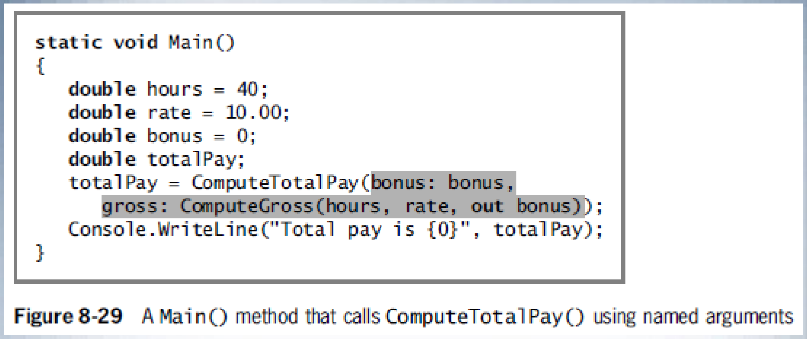
```
static void Main()
{
   double hours = 40;
   double rate = 10.00;
   double bonus = 0;
   double totalPay;
   totalPay = ComputeTotalPay(ComputeGross(hours, rate, out bonus),
      bonus);
   Console.WriteLine("Total pay is {0}", totalPay);
}
```

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

Output:

```
Total pay is $500.00
```

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

Output:
Total pay is $400.00

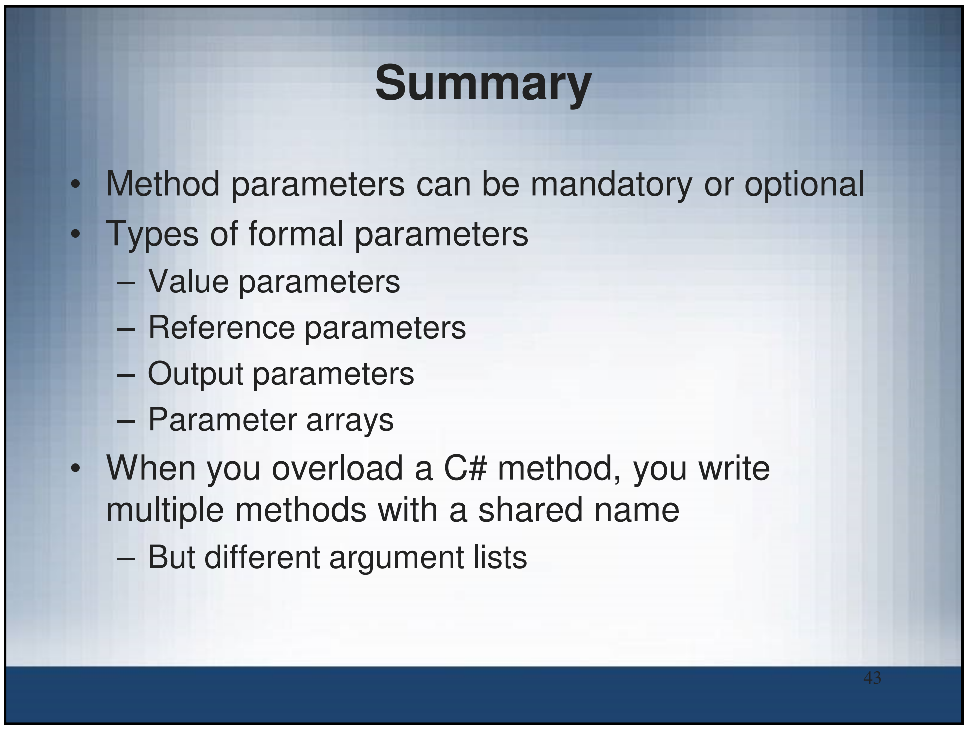
Overload Resolution with Named and Optional Arguments

- Named and optional arguments affect overload resolution
  - Rules for betterness on argument conversions are applied only for arguments that are given explicitly

```
public static void AMethod(int a, double b = 2.2)
public static void AMethod(int a, char b = 'H')
AMethod(12); // causes a problem
```

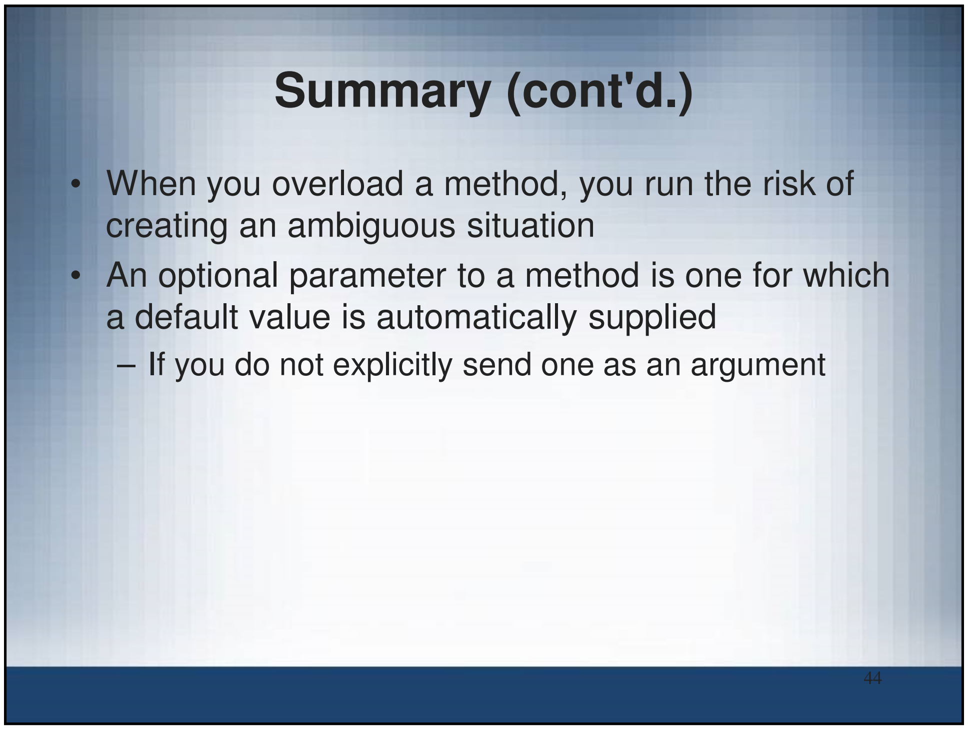
- If two signatures are equally good
  - Signature that does not omit optional parameters is considered better

```
public static void BMethod(int a)
public static void BMethod(int a, char b = 'H')
BMethod(12); // call the first version
```



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