7

Methods: A Deeper Look

Procedural Note

In the lab, it may be necessary to give the full path to the S: drive when creating a new project

\\ignis\sharespace\dar5p\My Documents\Visual Studio 2008\Projects

7.12 Method Overloading

- Methods of the same name can be declared in the same class, or **overloaded**, as long as they have different sets of parameters.
- When an **overloaded method** is called, the C# compiler selects the appropriate method by examining the number, types and order of the arguments in the call.
- Method overloading is used to create several methods with the same name that perform the same tasks, but on different types or numbers of arguments.

Declaring Overloaded Methods

Class `MethodOverload` (Fig. 7.13) includes two overloaded versions of a method called `Square`.

MethodOverload.cs

(1 of 2)

```
// Fig. 7.13: MethodOverload.cs
// Overloaded method declarations.
using System;

public class MethodOverload
{
   // test overloaded square methods
   public void TestOverloadedMethods()
   {
      Console.WriteLine( "Square of integer 7 is {0}", Square( 7 ) );
      Console.WriteLine( "Square of double 7.5 is {0}", Square( 7.5 ) );
   } // end method TestOverloadedMethods

   // square method with int argument
   public int Square( int intValue )
```

Overloaded version of the method that operates on an integer.

Overloaded version of the method that operates on a double.

Fig. 7.13 | Overloaded method declarations. (Part 1 of 2.)

MethodOverload.cs

(2 of 2)

```
   {
      Console.WriteLine( "Called square with int argument: {0}",
         intValue );
      return intValue * intValue;
   } // end method Square with int argument

   // square method with double argument
   public doub Square( double doubleValue )
   {
      Console.WriteLine( "Called square with double argument: {0}",
         doubleValue );
      return doubleValue * doubleValue;
   } // end method Square with double argument
} // end class MethodOverload
```

Overloaded version of the method that operates on a double.

Fig. 7.13 | Overloaded method declarations. (Part 2 of 2.)

- Class `MethodOverloadTest` (Fig. 7.14) tests class `MethodOverload`.

MethodOverload Test.cs

```
// Fig. 7.14: MethodOverloadTest.cs
// Application to test class MethodOverload.
public class MethodOverloadTest
{
  public static void Main( string[] args )
  {
    MethodOverload methodOverload = new MethodOverload();
    methodOverload.TestOverloadedMethods();
  } // end Main
} // 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
```

Fig. 7.14 | Application to test class `MethodOverload`.

Distinguishing Between Overloaded Methods

- The compiler distinguishes overloaded methods by their **signature**—a combination of the method's name and the number, types and order of its parameters.

Return Types of Overloaded Methods

- Method calls cannot be distinguished by return type.
- The application in Fig. 7.15 illustrates the compiler errors generated when two methods have the same signature but different return types.

MethodOverload.cs

(1 of 3)

7.13 Recursion

- A **recursive method** is a method that calls itself.
- A recursive method is capable of solving only the **base case(s)**.
- Each method call divides the problem into two conceptual pieces: a piece that the method knows how to do and a **recursive call**, or **recursion step** that solves a smaller problem.
- A sequence of returns ensues until the original method call returns the result to the caller.

7.13 Recursion Examples

- Factorial
- Fibonacci

- Figure 7.17 uses recursion to calculate and display the factorials of the integers from 0 to 10.

FactorialTest.cs

(1 of 2)

```
// Fig. 7.17: FactorialTest.cs
// Recursive Factorial method.
using System;

public class FactorialTest
{
   public static void Main( string[] args )
   {
      // calculate the factorials of 0 through 10
      for ( long counter = 0; counter <= 10; counter++ )
         Console.WriteLine( "{0}! = {1}",
            counter, Factorial( counter ) );
   } // end Main

   // recursive declaration of method Factorial
```

Fig. 7.17 | Recursive `Factorial` method. (Part 1 of 2.)

FactorialTest.cs

(2 of 2)

```
   public static long Factorial( long number )
   {
      // base case
      if ( number <= 1 )
         return 1;
      // recursion step
      else
         return number * Factorial( number - 1 );
   } // end method Factorial
} // end class FactorialTest
```

First, test to determine whether the terminating condition is `true`.

The recursive call solves a slightly simpler problem than the original calculation.

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

Fig. 7.17 | Recursive `Factorial` method. (Part 2 of 2.)

7.13 Recursion (Cont.)

- First, test to determine whether the terminating condition is `true`.
- The recursive call solves a slightly simpler problem than the original calculation.

Common Programming Error 7.11

Either omitting the base case or writing the recursion step incorrectly so that it does not converge on the base case will cause infinite recursion, eventually exhausting memory. This error is analogous to the problem of an infinite loop in an iterative (nonrecursive) solution.

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

- Two ways to pass arguments to functions in many programming languages are **pass-by-value** and **pass-by-reference**.
- When an argument is passed by value (the default in C#), a *copy* of its value is made and passed to the called function.
- When an argument is passed by reference, the caller gives the method the ability to access and modify the caller’s original variable.

7.14 Passing Arguments: Pass-by-Value vs. Pass-by-Reference (Cont.)

- Passing a reference-type variable passes the method a copy of the actual reference that refers to the object.
  - The reference itself is passed by value, but the method can still use the reference it receives to modify the original object in memory.
- A `return` statement returns a copy of the value stored in a value-type variable or a copy of the reference stored in a reference-type variable.
- In effect, objects are always passed by reference.

7.14 Passing Arguments: Pass-by-Value vs. Pass-by-Reference (Cont.)

- 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.

- Class `ReferenceAndOutputParameters` (Fig. 7.18) contains three methods that calculate the square of an integer.

`ReferenceAndOutputParameters.cs`

(1 of 4)

```
// Fig 7.18: ReferenceAndOutputParameters.cs
// Reference, output and value parameters.
using System;

class ReferenceAndOutputParameters
{
   // call methods with reference, output and value parameters
   public void DemonstrateReferenceAndOutputParameters()
   {
      int y = 5; // initialize y to 5
```

Fig. 7.18 | Reference, output and value parameters. (Part 1 of 4.)

ReferenceAndOutputParameters.cs

(2 of 4)

```
      int z; // declares z, but does not initialize it

      // display original values of y and z
      Console.WriteLine( "Original value of y: {0}", y );
      Console.WriteLine( "Original value of z: uninitialized\n" );

      // pass y and z by reference
      SquareRef( ref y ); // must use keyword ref
      SquareOut( out z ); // must use keyword out

      // display values of y and z after they are modified by
      // methods SquareRef and SquareOut, respectively
      Console.WriteLine( "Value of y after SquareRef: {0}", y );
```

When you pass a variable to a method with a reference parameter, you must precede the argument with the same keyword (`ref` or `out`) that was used to declare the reference parameter.

Fig. 7.18 | Reference, output and value parameters. (Part 2 of 4.)

ReferenceAndOutputParameters.cs

(3 of 4)

```
      Console.WriteLine( "Value of z after SquareOut: {0}\n", z );

      // pass y and z by value
      Square( y );
      Square( z );

      // display values of y and z after they are passed to method Square
      // to demonstrate that arguments passed by value are not modified
      Console.WriteLine( "Value of y after Square: {0}", y );
      Console.WriteLine( "Value of z after Square: {0}", z );
   } // end method DemonstrateReferenceAndOutputParameters

   // uses reference parameter x to modify caller's variable
   void SquareRef( ref int x )
   {
      x = x * x; // squares value of caller's variable
   } // end method SquareRef

```

Modify caller's x.

Fig. 7.18 | Reference, output and value parameters. (Part 3 of 4.)

ReferenceAndOutputParameters.cs

(4 of 4)

```

   // uses output parameter x to assign a value
   // to an uninitialized variable
   void SquareOut( out int x )
   {
      x = 6; // assigns a value to caller's variable
      x = x * x; // squares value of caller's variable
   } // end method SquareOut

   // parameter x receives a copy of the value passed as an argument,
   // so this method cannot modify the caller's variable
   void Square( int x )
   {
      x = x * x;
   } // end method Square
} // end class ReferenceAndOutputParameters
```

Assign a value to caller's uninitialized x.

Doesn't modify any caller variable.

Fig. 7.18 | Reference, output and value parameters. (Part 4 of 4.)

- Class `ReferenceAndOutputParametersTest` tests the `ReferenceAndOutputParameters` class.

ReferenceAnd OutputParamters Test.cs

```
// Fig7.19: ReferenceAndOutputParamtersTest.cs
// Application to test class ReferenceAndOutputParameters.
classReferenceAndOutputParamtersTest
{
  public static void Main( string[] args )
  {
    ReferenceAndOutputParameters test =
      new ReferenceAndOutputParameters();
    test.DemonstrateReferenceAndOutputParameters();
  } // end Main
} // end class ReferenceAndOutputParamtersTest
```

```
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
```

Fig. 7.19 | Application to test class `ReferenceAndOutputParameters`.

8

Arrays

8.1 Introduction

- **Data structures** are collections of related data items.
- **Arrays** are data structures consisting of related data items of the same type.
- Arrays are fixed-length entities—they remain the same length once they are created.

8.2 Arrays

- An array is a group of variables (called **elements**) containing values that all have the same type.
- Arrays are reference types—what we typically think of as an array is actually a reference to an array object.
- The elements of an array can be either value types or reference types.
- To refer to a particular element in an array, we specify the name of the reference to the array the element's position in the array, called the element's **index**.

8.2 Arrays (Cont.)

- **square brackets** (`[]`).
- Zero indexed
- An index must be a nonnegative integer and can be an expression.
- Every array’s length is stored in its `Length` property.
- Creating an array:

```
int[] c = new int[ 12 ];
```

8.3 Declaring and Creating Arrays (Cont.)

Difference from C++

- The number of elements can also be specified as an expression that is calculated at execution time.
- When an array is created, each element of the array receives a default value:
  - `0` for the numeric simple-type elements.
  - `false` for `bool` elements.
  - `null` for references.
- An application can create several arrays in a single declaration.
- For readability, it is better to write each array declaration in its own statement.

8.4 Examples Using Arrays

Using an Array Initializer

- An application can create an array and initialize its elements with an array initializer, a comma-separated list of expressions (called an initializer list) enclosed in braces.
- The array length is determined by the number of elements in the initializer list.
- A statement using an array initializer does not require `new` to create the. array object

- The application in Fig. 8.3 initializes an integer array with 10 values (line 10) and displays the array in tabular format.

InitArray.cs

(1 of 2)

```
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
```

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

InitArray.cs

(2 of 2)

```
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 2 of 2.)

- The code for displaying the array elements (lines 15–16) is identical to that in the previous example.

Calculating a Value to Store in Each Array Element

- The application in Fig. 8.4 creates a 10-element array and assigns to each element one of the even integers from 2 to 20 (`2`, `4`, `6`, ..., `20`).

InitArray.cs

(1 of 2)

```
// Fig. 8.4: InitArray.cs
// Calculating values to be placed into the elements of an array.
using System;

public class InitArray
{
  public static void Main( string[] args )
  {
    const int ARRAY_LENGTH = 10; // create a named constant
    int[] array = new int[ ARRAY_LENGTH ]; // create array

    // calculate value for each array element
    for ( int counter = 0; counter < array.Length; counter++ )
      array[ counter ] = 2 + 2 * counter;

```

Constants must be initialized when they are declared and cannot be modified thereafter.

Fig. 8.4 | Calculating values to be placed into the elements of an array. (Part 1 of 2.)

InitArray.cs

(2 of 2)

```
      Console.WriteLine( "{0}{1,8}", "Index", "Value" ); // headings

      // output each array element's value
      for ( int counter = 0; counter < array.Length; counter++ )
         Console.WriteLine( "{0,5}{1,8}", counter, array[ counter ] );
   } // end Main
} // end class InitArray
```

```
Index   Value
    0       2
    1       4
    2       6
    3       8
    4      10
    5      12
    6      14
    7      16
    8      18
    9      20
```

Fig. 8.4 | Calculating values to be placed into the elements of an array. (Part 2 of 2.)

Summing the Elements of an Array

- The application in Fig. 8.5 sums the values contained in a 10-element integer array.

SumArray.cs

```
// Fig. 8.5: SumArray.cs
// Computing the sum of the elements of an array.
using System;

public class SumArray
{
  public static void Main( string[] args )
  {
    int[] array = { 87, 68, 94, 100, 83, 78, 85, 91, 76, 87 };
    int total = 0;

    // add each element's value to total
    for ( int counter = 0; counter < array.Length; counter++ )
      total += array[ counter ];

    Console.WriteLine( "Total of array elements: {0}", total );
  } // end Main
} // end class SumArray
```

Loop through the array elements and sum their values.

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

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

Using Bar Charts to Display Array Data Graphically

Outline

BarChart.cs

(1 of 2)

- The application in Fig. 8.6 stores grade distribution data in an array of 11 elements, each corresponding to a category of grades.

```
// Fig. 8.6: BarChart.cs
// Bar chart displaying application.
using System;

public class BarChart
{
   public static void Main( string[] args )
   {
      int[] array = { 0, 0, 0, 0, 0, 0, 1, 2, 4, 2, 1 };

      Console.WriteLine( "Grade distribution:" );

      // for each array element, output a bar of the chart
      for ( int counter = 0; counter < array.Length; counter++ )
      {
         // output bar labels ( "00-09: ", ..., "90-99: ", "100: " )
         if ( counter == 10 )
            Console.Write( " 100: " );
         else
            Console.Write( "{0:D2}-{1:D2}: ",
```

Fig. 8.6 | Bar chart displaying application. (Part 1 of 2.)

```

         // display bar of asterisks
         for ( int stars = 0; stars < array[ counter ]; stars++ )
            Console.Write( "*" );

         Console.WriteLine(); // start a new line of output
      } // end outer for
   } // end Main
} // end class BarChart
```

```
Grade distribution:
00-09:
10-19:
20-29:
30-39:
40-49:
50-59:
60-69: *
70-79: **
80-89: ****
90-99: **
  100: *
```

BarChart.cs

(2 of 2)

Output the number of stars corresponding to the value of the array in each row.

Fig. 8.6 | Bar chart displaying application. (Part 2 of 2.)

- `array[ 0 ]` indicates the number of grades in the range 0–9.
- `array[ 7 ]` indicates the number of grades in the range 70–79.
- `array[ 10 ]` indicates the number of 100 grades.

Using the Elements of an Array as Counters

- An array version of our die-rolling application from Fig. 7.8 is shown in Fig. 8.7.

RollDie.cs

(1 of 2)

```
// Fig. 8.7: RollDie.cs
// Roll a six-sided die 6000 times.
using System;

public class RollDie
{
   public static void Main( string[] args )
   {
      Random randomNumbers = new Random(); // random-number generator
      int[] frequency = new int[ 7 ]; // array of frequency counters

      // roll die 6000 times; use die value as frequency index
      for ( int roll = 1; roll <= 6000; roll++ )
         ++frequency[ randomNumbers.Next( 1, 7 ) ];

      Console.WriteLine( "{0}{1,10}", "Face", "Frequency" );
```

Use a seven-element array, ignoring `frequency[ 0 ]` because it is more logical to simply use the face value as an index for array `frequency`.

Use `frequency` to count the occurrences of each side of the die.

Fig. 8.7 | Roll a six-sided die 6000 times. (Part 1 of 2.)

RollDie.cs

(2 of 2)

```

      // output each array element's value
      for ( int face = 1; face < frequency.Length; face++ )
         Console.WriteLine( "{0,4}{1,10}", face, frequency[ face ] );
   } // end Main
} // end class RollDie
```

```
Face Frequency
   1       956
   2       981
   3      1001
   4      1030
   5      1035
   6       997
```

Fig. 8.7 | Roll a six-sided die 6000 times. (Part 2 of 2.)

Using Arrays to Analyze Survey Results

- Our next example (Fig. 8.8) uses arrays to summarize the results of data collected in a survey:

Forty students were asked to rate the quality of the food in the student cafeteria on a scale of 1 to 10 (where 1 means awful and 10 means excellent). Place the 40 responses in an integer array and summarize the results of the poll.

StudentPoll.cs

(1 of 2)

```
// Fig. 8.8: StudentPoll.cs
// Poll analysis application.
using System;

public class StudentPoll
{
   public static void Main( string[] args )
   {
      // array of survey responses
      int[] responses = { 1, 2, 6, 4, 8, 5, 9, 7, 8, 10, 1, 6, 3, 8, 6,
         10, 3, 8, 2, 7, 6, 5, 7, 6, 8, 6, 7, 5, 6, 6, 5, 6, 7, 5, 6,
         4, 8, 6, 8, 10 };
      int[] frequency = new int[ 11 ]; // array of frequency counters

```

Use 11-element array `frequency` to count the number of occurrences of each response. As in the previous example, we ignore `frequency[ 0 ]`.

Fig. 8.8 | Poll analysis application. (Part 1 of 2.)

```
      // for each answer, select responses element and use that value
      // as frequency index to determine element to increment
      for ( int answer = 0; answer < responses.Length; answer++ )
         ++frequency[ responses[ answer ] ];

      Console.WriteLine( "{0}{1,10}", "Rating", "Frequency" );

      // output each array element's value
      for ( int rating = 1; rating < frequency.Length; rating++ )
         Console.WriteLine( "{0,6}{1,10}", rating, frequency[ rating ] );
   } // end Main
} // end class StudentPoll
```

```
Rating Frequency
     1         2
     2         2
     3         2
     4         2
     5         5
     6        11
     7         5
     8         7
     9         1
    10         3
```

StudentPoll.cs

(2 of 2)

Increment the appropriate `frequency` counter, depending on the value of `responses[ answer ]`.

Fig. 8.8 | Poll analysis application. (Part 2 of 2.)

8.4 Examples Using Arrays (Cont.)

- In many programming languages, like C and C++, writing outside the bounds of an array is allowed, but often causes disastrous results.
- In C#, accessing any array element forces a check on the array index to ensure that it is valid. This is called **bounds checking**.
- If an application uses an invalid index, the Common Language Runtime generates an **`IndexOutOfRangeException`** to indicate that an error occurred in the application at execution time.

8.6 foreach Statement

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

  `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 match the type of the elements in the array.
- The iteration variable represents successive values in the array on successive iterations of the `foreach` statement.

- Figure 8.12 uses the `foreach` statement to calculate the sum of the integers in an array of student grades.

ForEachTest.cs

(1 of 2)

```
// Fig. 8.12: ForEachTest.cs
// Using the foreach statement to total integers in an array.
using System;

public class ForEachTest
{
   public static void Main( string[] args )
   {
      int[] array = { 87, 68, 94, 100, 83, 78, 85, 91, 76, 87 };
      int total = 0;

      // add each element's value to total
      foreach ( int number in array )
         total += number;

      Console.WriteLine( "Total of array elements: {0}", total );
   } // end Main
} // end class ForEachTest
```

For each iteration, `number` represents the next `int` value in the array.

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

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

Common Programming Error 8.4

The `foreach` statement can be used only to access array elements—it cannot be used to modify elements. Any attempt to change the value of the iteration variable in the body of a `foreach` statement will cause a compilation error.

`ForEachTest.cs`

(2 of 2)

- The `foreach` statement can be used in place of the `for` statement whenever code looping through an array does not need to know the index of the current array element.

8.6 foreach Statement (Cont.)

Implicitly Typed Local Variables

- C# provides a new feature—called **implicitly typed local variables** —that enables the compiler to infer a local variable's type based on the type of the variable's initializer.
- To distinguish such an initialization from a simple assignment statement, the `var` keyword is used in place of the variable's type.
- The compiler assumes that floating-point number values are of type `double`.
- You can use local type inference with control variables in the header of a `for` or `foreach` statement.
- For example, the following `for` statement headers are equivalent:

```
for ( int counter = 1; counter < 10; counter++ )
for ( var counter = 1; counter < 10; counter++ )
```

8.6 foreach Statement (Cont.)

- Similarly, if `myArray` is an array of `int`s, the following `foreach` statement headers are equivalent:

```
foreach (int number in myArray)
foreach (var number in myArray)
```

- The implicitly typed local-variable feature is one of several new Visual C# 2008 features that support Language Integrated Query (LINQ).
- Implicitly typed local variables can be also used to initialize arrays without explicitly giving their type.
  - There are no square brackets on the left side of the assignment operator.
  - `new[]` is used on the right to specify that the variable is an array.

8.7 Passing Arrays and Array Elements to Methods

- To pass an array argument to a method, specify the name of the array without any brackets. For a method to receive an array reference through a method call, the method's parameter list must specify an array parameter.
- When an argument to a method is an entire array or an individual array element of a reference type, the called method receives a copy of the reference.
- When an argument to a method is an individual array element of a value type, the called method receives a copy of the element's value.
- To pass an individual array element to a method, use the indexed name of the array as an argument in the method call.

- Figure 8.13 demonstrates the difference between passing an entire array and passing a value-type array element to a method.

PassArray.cs

(1 of 3)

```
// Fig. 8.13: PassArray.cs
// Passing arrays and individual array elements to methods.
using System;

public class PassArray
{
   // Main creates array and calls ModifyArray and ModifyElement
   public static void Main( string[] args )
   {
      int[] array = { 1, 2, 3, 4, 5 };

      Console.WriteLine(
         "Effects of passing reference to entire array:\n" +
         "The values of the original array are:" );

      // output original array elements
      foreach ( int value in array )
         Console.Write( "   {0}", value );

```

Fig. 8.13 | Passing arrays and individual array elements to methods. (Part 1 of 3.)

Outline

```
      ModifyArray( array ); // pass array reference
         Console.WriteLine( "\n\nThe values of the modified array are:" );

      // output modified array elements
      foreach ( int value in array )
         Console.Write( "  {0}", value );

      Console.WriteLine(
         "\n\nEffects of passing array element value:\n" +
         "array[3] before ModifyElement: {0}", array[ 3 ] );

      ModifyElement( array[ 3 ] ); // attempt to modify array[ 3 ]
      Console.WriteLine(
         "array[3] after ModifyElement: {0}", array[ 3 ] );
   } // end Main

   // multiply each element of an array by 2
   public static void ModifyArray( int[] array2 )
   {
      for ( int counter = 0; counter < array2.Length; counter++ )
         array2[ counter ] *= 2;
   } // end method ModifyArray

```

PassArray.cs

(2 of 3)

Method receives a copy of `array`’s reference.

Fig. 8.13 | Passing arrays and individual array elements to methods. (Part 2 of 3.)

Outline

```
    // multiply argument by 2
    public static void ModifyElement( int element )
    {
       element *= 2;
       Console.WriteLine(
          "Value of element in ModifyElement:, element );
    } // end method ModifyElement
} // end class PassArray
```

```
Effects of passing reference to entire array:
The values of the original array are:
   1   2   3   4   5

The values of the modified array are:
   2   4   6   8   10

Effects of passing array element value:
array[3] before ModifyElement: 8
Value of element in ModifyElement: 16
array[3] after ModifyElement: 8
```

`PassArray.cs`

(3 of 3)

Does not modify the array because `ModifyElement` receives a copy of the `int` value of `array[ 3 ]`.

Fig. 8.13 | Passing arrays and individual array elements to methods. (Part 3 of 3.)

8.8 Passing Arrays by Value and by Reference (Cont.)

- You can use keyword `ref` to pass a reference-type variable *by reference*, which allows the called method to modify the original variable in the caller and make that variable refer to a different object.
- This is a subtle capability, which, if misused, can lead to problems.

Outline

ArrayReferenceTest.cs

(1 of 5)

- The application in Fig. 8.14 demonstrates the subtle difference between passing a reference by value and passing a reference by reference with keyword `ref`.

```
// Fig. 8.14: ArrayReferenceTest.cs
// Testing the effects of passing array references
// by value and by reference.
using System;

public class ArrayReferenceTest
{
   public static void Main( string[] args )
   {
      // create and initialize firstArray
      int[] firstArray = { 1, 2, 3 };

      // copy the reference in variable firstArray
      int[] firstArrayCopy = firstArray;

      Console.WriteLine(
         "Test passing firstArray reference by value" );

```

Fig. 8.14 | Passing an array reference by value and by reference. (Part 1 of 5.)

Outline

```
      Console.Write( "\nContents of firstArray " +
         "before calling FirstDouble:\n\t" );

      // display contents of firstArray
      for ( int i = 0; i < firstArray.Length; i++ )
         Console.Write( "{0} ", firstArray[ i ] );

      // pass variable firstArray by value to FirstDouble
      FirstDouble( firstArray );

      Console.Write( "\n\nContents of firstArray after " +
         "calling FirstDouble\n\t" );

      // display contents of firstArray
      for ( int i = 0; i < firstArray.Length; i++ )
         Console.Write( "{0} ", firstArray[ i ] );

      // test whether reference was changed by FirstDouble
      if ( firstArray == firstArrayCopy )
         Console.WriteLine(
            "\n\nThe references refer to the same array" );
      else
         Console.WriteLine(
            "\n\nThe references refer to different arrays" );

```

ArrayReference Test.cs

(2 of 5)

Fig. 8.14 | Passing an array reference by value and by reference. (Part 2 of 5.)

```
      // create and initialize secondArray
      int[] secondArray = { 1, 2, 3 };

      // copy the reference in variable secondArray
      int[] secondArrayCopy = secondArray;

      Console.WriteLine( "\nTest passing secondArray " +
         "reference by reference" );

      Console.Write( "\nContents of secondArray " +
         "before calling SecondDouble:\n\t" );

      // display contents of secondArray before method call
      for ( int i = 0; i < secondArray.Length; i++ )
         Console.Write( "{0} ", secondArray[ i ] );

      // pass variable secondArray by reference to SecondDouble
      SecondDouble( ref secondArray );

      Console.Write( "\n\nContents of secondArray " +
         "after calling SecondDouble:\n\t" );

      // display contents of secondArray after method call
      for ( int i = 0; i < secondArray.Length; i++ )
         Console.Write( "{0} ", secondArray[ i ] );
```

ArrayReference Test.cs

(3 of 5)

Fig. 8.14 | Passing an array reference by value and by reference. (Part 3 of 5.)

```
         // test whether reference was changed by SecondDouble
      if ( secondArray == secondArrayCopy )
         Console.WriteLine(
            "\n\nThe references refer to the same array" );
      else
         Console.WriteLine(
            "\n\nThe references refer to different arrays" );
   } // end Main

   // modify elements of array and attempt to modify reference
   public static void FirstDouble( int[] array )
   {
         // double each element's value
      for ( int i = 0; i < array.Length; i++ )
         array[ i ] *= 2;

      // create new object and assign its reference to array
      array = new int[] { 11, 12, 13 };
   } // end method FirstDouble

   // modify elements of array and change reference array
   // to refer to a new array
```

This does not overwrite the caller's reference `firstDouble`.

ArrayReferenceTest.cs

(4 of 5)

Fig. 8.14 | Passing an array reference by value and by reference. (Part 4 of 5.)

```
   public static void SecondDouble( ref int[] array )
   {
      // double each element's value
      for ( int i = 0; i < array.Length; i++ )
         array[ i ] *= 2;

      // create new object and assign its reference to array
      array = new int[] { 11, 12, 13 };
   } // end method SecondDouble
} // end class ArrayReferenceTest
```

```
Test passing firstArray reference by value

Contents of firstArray before calling FirstDouble:
       1 2 3

Contents of firstArray after calling FirstDouble
       2 4 6

The references refer to the same array

Test passing secondArray reference by reference

Contents of secondArray before calling SecondDouble:
       1 2 3

Contents of secondArray after calling SecondDouble:
       11 12 13

The references refer to different arrays
```

ArrayReferenceTest.cs

(5 of 5)

This assignment modifies the caller’s `secondDouble` reference to reference a new array.

Fig. 8.14 | Passing an array reference by value and by reference. (Part 5 of 5.)

Storing Student Grades in an Array in Class **GradeBook**

Outline

- The version of class `GradeBook` (Fig. 8.15) presented here uses an array of integers to store the grades of several students on a single exam.

GradeBook.cs

(1 of 6)

```
// Fig. 8.15: GradeBook.cs
// Grade book using an array to store test grades.
using System;

public class GradeBook
{
   private int[] grades; // array of student grades

   // auto-implemented property CourseName
   public string CourseName { get; set; }

   // two-parameter constructor initializes
   // auto-implemented property CourseName and grades array
   public GradeBook( string name, int[] gradesArray )
   {
      CourseName = name; // set CourseName to name
      grades = gradesArray; // initialize grades array
   } // end two-parameter GradeBook constructor

```

The application that creates a Gradebook object is responsible for creating an array of the grades. The size of array `grades` is determined by the class that passes the array to the constructor.

Fig. 8.15 | Grade book using an array to store test grades. (Part 1 of 6.)

```
   // display a welcome message to the GradeBook user
   public void DisplayMessage()
   {
      // auto-implemented property CourseName gets the name of course
      Console.WriteLine( "Welcome to the grade book for\n{0}!\n",
         CourseName );
   } // end method DisplayMessage

   // perform various operations on the data
   public void ProcessGrades()
   {
      // output grades array
      OutputGrades();

      // call method GetAverage to calculate the average grade
      Console.WriteLine( "\nClass average is {0:F}", GetAverage() );

      // call methods GetMinimum and GetMaximum
      Console.WriteLine( "Lowest grade is {0}\nHighest grade is {1}\n",
         GetMinimum(), GetMaximum() );

      // call OutputBarChart to display grade distribution chart
      OutputBarChart();
   } // end method ProcessGrades

```

GradeBook.cs

(2 of 6)

Fig. 8.15 | Grade book using an array to store test grades. (Part 2 of 6.)

Outline

GradeBook.cs

(3 of 6)

```
   // find minimum grade
   public int GetMinimum()
   {
      int lowGrade = grades[ 0 ]; // assume grades[ 0 ] is smallest

      // loop through grades array
      foreach ( int grade in grades )
      {
         // if grade lower than lowGrade, assign it to lowGrade
         if ( grade < lowGrade )
            lowGrade = grade; // new lowest grade
      } // end for

      return lowGrade; // return lowest grade
   } // end method GetMinimum

   // find maximum grade
   public int GetMaximum()
   {
      int highGrade = grades[ 0 ]; // assume grades[ 0 ] is largest

```

Use a `foreach` statement to find the minimum grade.

Fig. 8.15 | Grade book using an array to store test grades. (Part 3 of 6.)

```
      // loop through grades array
      foreach ( int grade in grades )
      {
         // if grade greater than highGrade, assign it to highGrade
         if ( grade > highGrade )
            highGrade = grade; // new highest grade
      } // end for

      return highGrade; // return highest grade
   } // end method GetMaximum

   // determine average grade for test
   public double GetAverage()
   {
      int total = 0; // initialize total

      // sum grades for one student
      foreach ( int grade in grades )
         total += grade;

      // return average of grades
      return ( double ) total / grades.Length;
   } // end method GetAverage

```

GradeBook.cs

(4 of 6)

Total the grades using a `foreach` statement.

Fig. 8.15 | Grade book using an array to store test grades. (Part 4 of 6.)

```
   // output bar chart displaying grade distribution
   public void OutputBarChart()
   {
      Console.WriteLine( "Grade distribution:" );

      // stores frequency of grades in each range of 10 grades
      int[] frequency = new int[ 11 ];

      // for each grade, increment the appropriate frequency
      foreach ( int grade in grades )
         ++frequency[ grade / 10 ];

      // for each grade frequency, display bar in chart
      for ( int count = 0; count < frequency.Length; count++ )
      {
         // output bar label ( "00-09: ", ..., "90-99: ", "100: " )
         if ( count == 10 )
            Console.Write( "  100: " );
         else
            Console.Write( "{0:D2}-{1:D2}: ",
               count * 10, count * 10 + 9 );

```

GradeBook.cs

(5 of 6)

Use integer division to count the frequency of grades in 10-point ranges.

Fig. 8.15 | Grade book using an array to store test grades. (Part 5 of 6.)

Outline

GradeBook.cs

(6 of 6)

```
               // display bar of asterisks
         for ( int stars = 0; stars < frequency[ count ]; stars++ )
            Console.Write( "*" );

         Console.WriteLine(); // start a new line of output
      } // end outer for
   } // end method OutputBarChart

   // output the contents of the grades array
   public void OutputGrades()
   {
      Console.WriteLine( "The grades are:\n" );

      // output each student's grade
      for ( int student = 0; student < grades.Length; student++ )
         Console.WriteLine( "Student {0,2}: {1,3}",
            student + 1, grades[ student ] );
   } // end method OutputGrades
} // end class GradeBook
```

A for statement, rather than a foreach, must be used in this case, because counter variable student's value is needed.

Fig. 8.15 | Grade book using an array to store test grades. (Part 6 of 6.)

Class **`GradeBookTest`** *That Demonstrates Class* **`GradeBook`**

- The application in Fig. 8.16 demonstrates class `GradeBook`.

`GradeBookTest.cs`

(1 of 4)

```
// Fig. 8.16: GradeBookTest.cs
// Create GradeBook object using an array of grades.
public class GradeBookTest
{
   // Main method begins application execution
   public static void Main( string[] args )
   {
      // one-dimensional array of student grades
      int[] gradesArray = { 87,68,94,100,83,78,85,91,76,87 };

      GradeBook myGradeBook = new GradeBook(
         "CS101 Introduction to C# Programming",gradesArray );
      myGradeBook.DisplayMessage();
      myGradeBook.ProcessGrades();
   } // end Main
} // end class GradeBookTest
```

Fig. 8.16 | Create a `GradeBook` object using an array of grades. (Part 1 of 3.)

GradeBookTest.cs

(2 of 4)

```
Welcome to the grade book for
CS101 Introduction to C# Programming!

The grades are:

Student  1:  87
Student  2:  68
Student  3:  94
Student  4: 100
Student  5:  83
Student  6:  78
Student  7:  85
Student  8:  91
Student  9:  76
Student 10:  87
```

Fig. 8.16 | Create a `GradeBook` object using an array of grades. (Part 2 of 3.)

Outline

GradeBookTest.cs

(3 of 4)

```
Class average is 84.90
Lowest grade is 68
Highest grade is 100

Grade distribution:
00-09:
10-19:
20-29:
30-39:
40-49:
50-59:
60-69: *
70-79: **
80-89: ****
90-99: **
  100: *
```

Fig. 8.16 | Create a `GradeBook` object using an array of grades. (Part 3 of 3.)

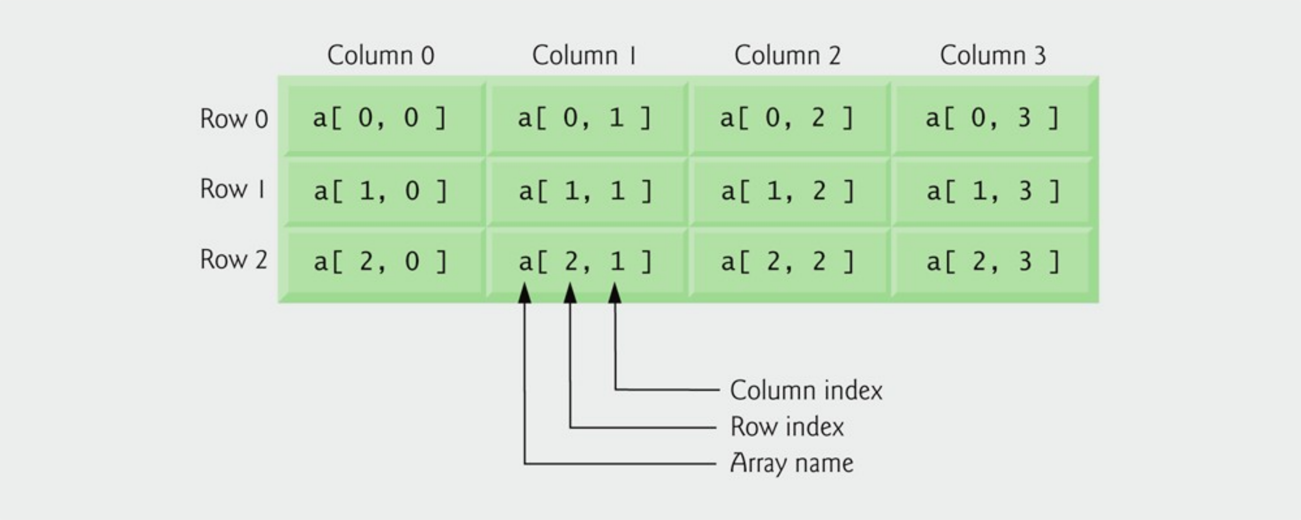
8.10 Multidimensional Arrays

- **Multidimensional arrays** with two dimensions are often used to represent **tables of values** consisting of information arranged in **rows** and **columns**.
- To identify a particular table element, we must specify two indices. By convention, the first identifies the element’s row and the second its column.
- Arrays that require two indices to identify a particular element are called **two-dimensional arrays**.

8.10 Multidimensional Arrays (Cont.)

Rectangular Arrays

- In rectangular arrays, each row has the same number of columns.
- Figure 8.17 illustrates a three-by-four rectangular array named `a`.



	Column 0	Column 1	Column 2	Column 3
Row 0	`a[ 0, 0 ]`	`a[ 0, 1 ]`	`a[ 0, 2 ]`	`a[ 0, 3 ]`
Row 1	`a[ 1, 0 ]`	`a[ 1, 1 ]`	`a[ 1, 2 ]`	`a[ 1, 3 ]`
Row 2	`a[ 2, 0 ]`	`a[ 2, 1 ]`	`a[ 2, 2 ]`	`a[ 2, 3 ]`

Fig. 8.17 | Rectangular array with three rows and four columns.

8.10 Multidimensional Arrays (Cont.)

- An array with *m* rows and *n* columns is called an ***m-by-n* array**.
- Every element in array a is identified by an array-access expression of the form a[*row, column*];
- A two-by-two rectangular array b can be declared and initialized with **nested array initializers** as follows:

```
int[ , ] b = { { 1, 2 }, { 3, 4 } };
```

  - The initializer values are grouped by row in braces.
- The compiler will generate an error if the number of initializers in each row is not the same, because every row of a rectangular array must have the same length.

8.10 Multidimensional Arrays (Cont.)

Jagged Arrays

- A **jagged array** is a one-dimensional array whose elements are one-dimensional arrays.
- The lengths of the rows in the array need not be the same.
- Elements in a jagged array are accessed using an array-access expression of the form *arrayName*[*row*][*column*].
- A jagged array with three rows of different lengths could be declared and initialized as follows:

```
int[][] jagged = { new int[] { 1, 2 },
                   new int[] { 3 },
                   new int[] { 4, 5, 6 } };
```

8.10 Multidimensional Arrays (Cont.)

- Figure 8.18 illustrates the array reference `jagged` after it has been declared and initialized.

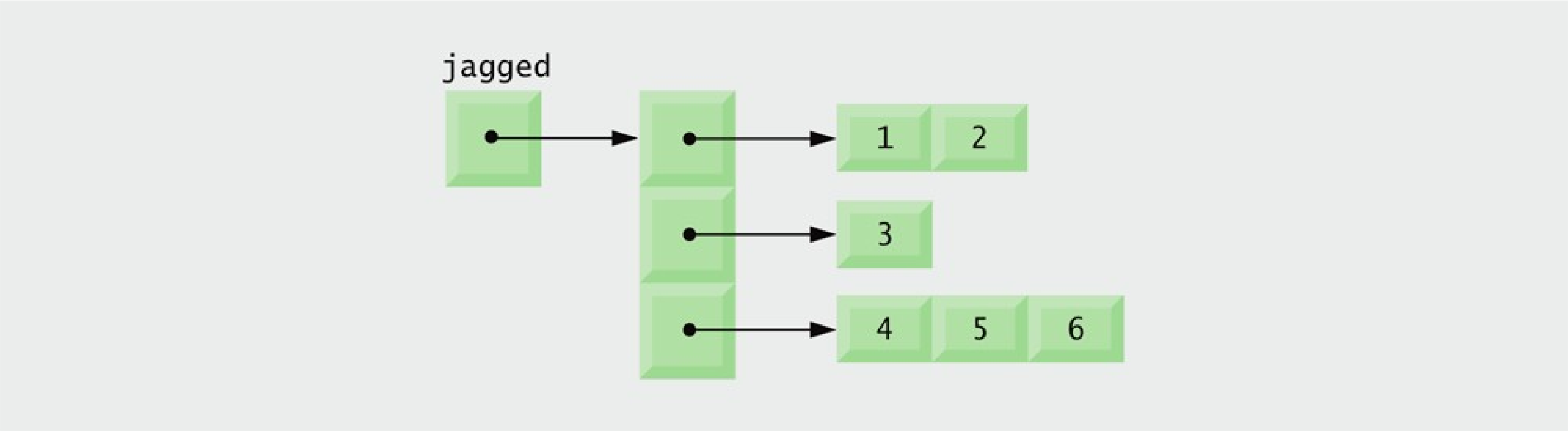


Fig. 8.18 | Jagged array with three rows of different lengths.

8.10 Multidimensional Arrays (Cont.)

Creating Two-Dimensional Arrays with Array-Creation Expressions

- A rectangular array can be created with an array-creation expression:

```
int[ , ] b;
b = new int[ 3, 4 ];
```

- A jagged array cannot be completely created with a single array-creation expression. Each one-dimensional array must be initialized separately.
- A jagged array can be created as follows:

```
int[][] c;
c = new int[ 2 ][ ]; // create 2 rows
c[ 0 ] = new int[ 5 ]; // create 5 columns for row 0
c[ 1 ] = new int[ 3 ]; // create 3 columns for row 1
```

Two-Dimensional Array Example: Displaying Element Values

- Figure 8.19 demonstrates initializing rectangular and jagged arrays with array initializers and using nested `for` loops to **traverse** the arrays.

InitArray.cs

(1 of 3)

```
// Fig. 8.19: InitArray.cs
// Initializing rectangular and jagged arrays.
using System;

public class InitArray
{
   // create and output rectangular and jagged arrays
   public static void Main( string[] args )
   {
      // with rectangular arrays,
      // every column must be the same length.
      int[ , ] rectangular = { { 1, 2, 3 }, { 4, 5, 6 } };

      // with jagged arrays,
      // we need to use "new int[]" for every row,
      // but every column does not need to be the same length.
      int[][] jagged = { new int[] { 1, 2 },
                         new int[] { 3 },
                         new int[] { 4, 5, 6 } };
```

Initialize a rectangular array using nested array initializers.

Each row of a jagged array is created with its own array initializer.

Fig. 8.19 | Initializing jagged and rectangular arrays. (Part 1 of 3.)

InitArray.cs

(2 of 3)

```
      OutputArray( rectangular ); // displays array rectangular by row
      Console.WriteLine(); // output a blank line
      OutputArray( jagged ); // displays array jagged by row
   } // end Main

   // output rows and columns of a rectangular array
   public static void OutputArray( int[ , ] array )
   {
      Console.WriteLine( "Values in the rectangular array by row are" );

      // loop through array's rows
      for ( int row = 0; row < array.GetLength( 0 ); row++ )
      {
         // loop through columns of current row
         for ( int column = 0; column < array.GetLength( 1 ); column++ )
            Console.Write( "{0} ", array[ row, column ] );

         Console.WriteLine(); // start new line of output
      } // end outer for
   } // end method OutputArray
```

Use the rectangular array's `GetLength` method to obtain the length of each dimension for the loop-continuation condition.

Fig. 8.19 | Initializing jagged and rectangular arrays. (Part 2 of 3.)

```
   // output rows and columns of a jagged array
   public static void OutputArray( int[][] array )
   {
      Console.WriteLine( "Values in the jagged array by row are" );

      // loop through each row
      foreach ( var row in array )
      {
         // loop through each element in current row
         foreach ( var element in row )
            Console.Write( "{0}  ", element );

         Console.WriteLine(); // start new line of output
      } // end outer foreach
   } // end method OutputArray
} // end class InitArray
```

```
Values in the rectangular array by row are
1  2  3
4  5  6

Values in the jagged array by row are
1  2
3
4  5  6
```

InitArray.cs

(3 of 3)

Using `foreach` statements allows the loop to determine the exact number of columns in each row.

Fig. 8.19 | Initializing jagged and rectangular arrays. (Part 3 of 3.)

Storing Student Grades in a Rectangular Array in Class **`GradeBook`**

Outline

- Figure 8.20 contains a version of class `GradeBook` that uses a rectangular array `grades` to store the grades of a number of students on multiple exams.

`GradeBook.cs`

(1 of 7)

```
// Fig. 8.20: GradeBook.cs
// Grade book using rectangular array to store grades.
using System;

public class GradeBook
{
   private int[ , ] grades; // rectangular array of student grades

   // auto-implemented property CourseName
   public string CourseName { get; set; }

   // two-parameter constructor initializes
   // auto-implemented property CourseName and grades array
   public GradeBook( string name, int[ , ] gradesArray )
   {
      CourseName = name; // set CourseName to name
      grades = gradesArray; // initialize grades array
   } // end two-parameter GradeBook constructor

```

Fig. 8.20 | Grade book using rectangular array to store grades. (Part 1 of 7.)

```
   // display a welcome message to the GradeBook user
   public void DisplayMessage()
   {
      // auto-implemented property CourseName gets the name of course
      Console.WriteLine( "Welcome to the grade book for\n{0}!\n",
         CourseName );
   } // end method DisplayMessage

   // perform various operations on the data
   public void ProcessGrades()
   {
      // output grades array
      OutputGrades();

      // call methods GetMinimum and GetMaximum
      Console.WriteLine( "\n{0} {1}\n{2} {3}\n",
         "Lowest grade in the grade book is", GetMinimum(),
         "Highest grade in the grade book is", GetMaximum() );

      // output grade distribution chart of all grades on all tests
      OutputBarChart();
   } // end method ProcessGrades

```

GradeBook.cs

(2 of 7)

Fig. 8.20 | Grade book using rectangular array to store grades. (Part 2 of 7.)

```
   // find minimum grade
   public int GetMinimum()
   {
      // assume first element of grades array is smallest
      int lowGrade = grades[ 0, 0 ];

      // loop through elements of rectangular grades array
      foreach ( int grade in grades )
      {
         // if grade less than lowGrade, assign it to lowGrade
         if ( grade < lowGrade )
            lowGrade = grade;
      } // end foreach

      return lowGrade; // return lowest grade
   } // end method GetMinimum

   // find maximum grade
   public int GetMaximum()
   {
      // assume first element of grades array is largest
      int highGrade = grades[ 0, 0 ];

```

GradeBook.cs

(3 of 7)

The `foreach` statement looks at each element of the first row in order by index, then each element of the second row in order by index and so on.

Fig. 8.20 | Grade book using rectangular array to store grades. (Part 3 of 7.)

Outline

```
      // loop through elements of rectangular grades array
      foreach ( int grade in grades )
      {
         // if grade greater than highGrade, assign it to highGrade
         if ( grade > highGrade )
            highGrade = grade;
      } // end foreach

      return highGrade; // return highest grade
   } // end method GetMaximum

   // determine average grade for particular student
   public double GetAverage( int student )
   {
      // get the number of grades per student
      int amount = grades.GetLength( 1 );
      int total = 0; // initialize total

      // sum grades for one student
      for ( int exam = 0; exam < amount; exam++ )
         total += grades[ student, exam ];

      // return average of grades
      return ( double ) total / amount;
   } // end method GetAverage

```

GradeBook.cs

(4 of 7)

Calculate the average of the array elements in a paricular row to find a single student’s average.

Fig. 8.20 | Grade book using rectangular array to store grades. (Part 4 of 7.)

```
   // output bar chart displaying overall grade distribution
   public void OutputBarChart()
   {
      Console.WriteLine( "Overall grade distribution:" );

      // stores frequency of grades in each range of 10 grades
      int[] frequency = new int[ 11 ];

      // for each grade in GradeBook, increment the appropriate frequency
      foreach ( int grade in grades )
      {
         ++frequency[ grade / 10 ];
      } // end foreach

      // for each grade frequency, display bar in chart
      for ( int count = 0; count < frequency.Length; count++ )
      {
         // output bar label ( "00-09: ", ..., "90-99: ", "100: " )
         if ( count == 10 )
            Console.Write( "  100: " );
         else
            Console.Write( "{0:D2}-{1:D2}: ",
               count * 10, count * 10 + 9 );

```

Same as the frequency for the one-dimensional array.

GradeBook.cs

(5 of 7)

Fig. 8.20 | Grade book using rectangular array to store grades. (Part 5 of 7.)

Outline

GradeBook.cs

(6 of 7)

```
         // display bar of asterisks
         for ( int stars = 0; stars < frequency[ count ]; stars++ )
            Console.Write( "*" );

         Console.WriteLine(); // start a new line of output
      } // end outer for
   } // end method OutputBarChart

   // output the contents of the grades array
   public void OutputGrades()
   {
      Console.WriteLine( "The grades are:\n" );
      Console.Write( "         " ); // align column heads

      // create a column heading for each of the tests
      for ( int test = 0; test < grades.GetLength( 1 ); test++ )
         Console.Write( "Test {0}  ", test + 1 );

      Console.WriteLine( "Average" ); // student average column heading

```

Fig. 8.20 | Grade book using rectangular array to store grades. (Part 6 of 7.)

GradeBook.cs

(7 of 7)

```
      // create rows/columns of text representing array grades
      for ( int student = 0; student < grades.GetLength( 0 ); student++ )
      {
         Console.Write( "Student {0,2}", student + 1 );

         // output student's grades
         for ( int grade = 0; grade < grades.GetLength( 1 ); grade++ )
            Console.Write( "{0,8}", grades[ student, grade ] );

         // call method GetAverage to calculate student's average grade;
         // pass row number as the argument to GetAverage
         Console.WriteLine( "{0,9:F}", GetAverage( student ) );
      } // end outer for
   } // end method OutputGrades
} // end class GradeBook
```

Fig. 8.20 | Grade book using rectangular array to store grades. (Part 7 of 7.)

Class **GradeBookTest** *That Demonstrates Class* **GradeBook**

- The application in Fig. 8.21 demonstrates class `GradeBook`.

GradeBookTest.cs

(1 of 3)

```
// Fig. 8.21: GradeBookTest.cs
// Create GradeBook object using a rectangular array of grades.
public class GradeBookTest
{
   // Main method begins application execution
  public static void Main( string[] args )
  {
    // rectangular array of student grades
    int[ , ] gradesArray = { { 87, 96, 70 },
                             { 68, 87, 90 },
                             { 94, 100, 90 },
                             { 100, 81, 82 },
                             { 83, 65, 85 },
                             { 78, 87, 65 },
                             { 85, 75, 83 },
                             { 91, 94, 100 },
                             { 76, 72, 84 },
                             { 87, 93, 73 } };

```

Nested array intializer lists initialize the array of grade data.

Fig. 8.21 | Create `GradeBook` object using a rectangular array of grades. (Part 1 of 3.)

GradeBookTest.cs

(2 of 3)

```
         GradeBook myGradeBook = new GradeBook(
            "CS101 Introduction to C# Programming", gradesArray );
      myGradeBook.DisplayMessage();
      myGradeBook.ProcessGrades();
   } // end Main
} // end class GradeBookTest
```

```
Welcome to the grade book for
CS101 Introduction to C# Programming!

The grades are:

            Test 1  Test 2  Test 3  Average
Student  1      87      96      70    84.33
Student  2      68      87      90    81.67
Student  3      94     100      90    94.67
Student  4     100      81      82    87.67
Student  5      83      65      85    77.67
Student  6      78      87      65    76.67
Student  7      85      75      83    81.00
Student  8      91      94     100    95.00
Student  9      76      72      84    77.33
Student 10      87      93      73    84.33
```

Fig. 8.21 | Create `GradeBook` object using a rectangular array of grades. (Part 2 of 3.)

Outline

GradeBookTest.cs

(3 of 3)

```
Lowest grade in the grade book is 65
Highest grade in the grade book is 100

Overall grade distribution:
00-09:
10-19:
20-29:
30-39:
40-49:
50-59:
60-69: ***
70-79: ******
80-89: ***********
90-99: *******
  100: ***
```

Fig. 8.21 | Create `GradeBook` object using a rectangular array of grades. (Part 3 of 3.)

Outline

ParamArrayTest.cs

(1 of 3)

- **Variable-length argument lists** allow you to create methods that receive an arbitrary number of arguments.
- The necessary `params` modifier can occur only in the last entry of the parameter list.
- Figure 8.22 demonstrates method `Average`, which receives a variable-length sequence of `double`s.

```
// Fig. 8.22: ParamArrayTest.cs
// Using variable-length argument lists.
using System;

public class ParamArrayTest
{
   // calculate average
   public static double Average( params double[] numbers )
   {
      double total = 0.0; // initialize total

```

Fig. 8.22 | Using variable-length argument lists. (Part 1 of 3.)

ParamArrayTest.cs

(2 of 3)

```
      // calculate total using the foreach statement
      foreach ( double d in numbers )
         total += d;

      return total / numbers.Length;
   } // end method Average

   public static void Main( string[] args )
   {
      double d1 = 10.0;
      double d2 = 20.0;
      double d3 = 30.0;
      double d4 = 40.0;

      Console.WriteLine(
         "d1 = {0:F1}\nd2 = {1:F1}\nd3 = {2:F1}\nd4 = {3:F1}\n",
         d1, d2, d3, d4 );

```

The method body can manipulate the parameter `numbers` as an array of `doubles`.

Fig. 8.22 | Using variable-length argument lists. (Part 2 of 3.)

```
         Console.WriteLine( "Average of d1 and d2 is {0:F1}",
           Average( d1, d2 ) );
      Console.WriteLine( "Average of d1, d2 and d3 is {0:F1}",
           Average( d1, d2, d3 ) );
      Console.WriteLine( "Average of d1, d2, d3 and d4 is {0:F1}",
           Average( d1, d2, d3, d4 ) );
   } // end Main
} // end class ParamArrayTest
```

ParamArrayTest.cs

(3 of 3)

```
d1 = 10.0
d2 = 20.0
d3 = 30.0
d4 = 40.0

Average of d1 and d2 is 15.0
Average of d1, d2 and d3 is 20.0
Average of d1, d2, d3 and d4 is 25.0
```

Fig. 8.22 | Using variable-length argument lists. (Part 3 of 3.)

Common Programming Error 8.5

The `params` modifier may be used only with the last parameter of the parameter list.

8.13 Using Command-Line Arguments

- You can pass command-line arguments to an application by including a parameter of type `string[]` in the parameter list of `Main`.
- By convention, this parameter is named `args`.
- The execution environment passes the command-line arguments as an array to the application's `Main` method.
- The number of arguments passed from the command line is obtained by accessing the array's `Length` property.
- Command-line arguments are separated by white space, not commas.

Outline

- Figure 8.23 uses three command-line arguments to initialize an array.

InitArray.cs

(1 of 3)

```
// Fig. 8.23: InitArray.cs
// Using command-line arguments to initialize an array.
using System;

public class InitArray
{
   public static void Main( string[] args )
   {
      // check number of command-line arguments
      if ( args.Length != 3 )
         Console.WriteLine(
            "Error: Please re-enter the entire command, including\n" +
            "an array size, initial value and increment." );
      else
      {
         // get array size from first command-line argument
         int arrayLength = Convert.ToInt32( args[ 0 ] );
         int[] array = new int[ arrayLength ]; // create array

```

Convert the command-line arguments to `int` values and store them in local variables.

Fig. 8.23 | Using command-line arguments to initialize an array. (Part 1 of 3.)

Outline

InitArray.cs

(2 of 3)

```
         // get initial value and increment from command-line argument
         int initialValue = Convert.ToInt32( args[ 1 ] );
         int increment = Convert.ToInt32( args[ 2 ] );

         // calculate value for each array element
         for ( int counter = 0; counter < array.Length; counter++ )
            array[ counter ] = initialValue + increment * counter;

         Console.WriteLine( "{0}{1,8}", "Index", "Value" );

         // display array index and value
         for ( int counter = 0; counter < array.Length; counter++ )
            Console.WriteLine( "{0,5}{1,8}", counter, array[ counter ] );
      } // end else
   } // end Main
} // end class InitArray
```

Convert the command-line arguments to `int` values and store them in local variables.

```
C:\Examples\ch08\fig08_23>InitArray.exe
Error: Please re-enter the entire command, including
an array size, initial value and increment.
```

Fig. 8.23 | Using command-line arguments to initialize an array. (Part 2 of 3.)

InitArray.cs

(3 of 3)

```
C:\Examples\ch08\fig08_23>InitArray.exe 5 0 4
Index  Value
   0     0
   1     4
   2     8
   3    12
   4    16
```

```
C:\Examples\ch08\fig08_23>InitArray.exe 10 1 2
Index  Value
   0     1
   1     3
   2     5
   3     7
   4     9
   5    11
   6    13
   7    15
   8    17
   9    19
```

Fig. 8.23 | Using command-line arguments to initialize an array. (Part 3 of 3.)