4a

Arrays

OBJECTIVES

In this lecture you will learn:

- What arrays are.
- To use arrays to store data in and retrieve data from lists and tables of values.
- To declare arrays, initialize arrays and refer to individual elements of arrays.
- To use the `foreach` statement to iterate through arrays.

OBJECTIVES

- To use implicitly typed local variables.
- To pass arrays to methods.
- To declare and manipulate multidimensional arrays.
- To write methods that use variable-length argument lists.
- To read command-line arguments into an application.

Outline

Outline

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.)

Figure 8.1 shows a logical representation of an integer array called `c`.

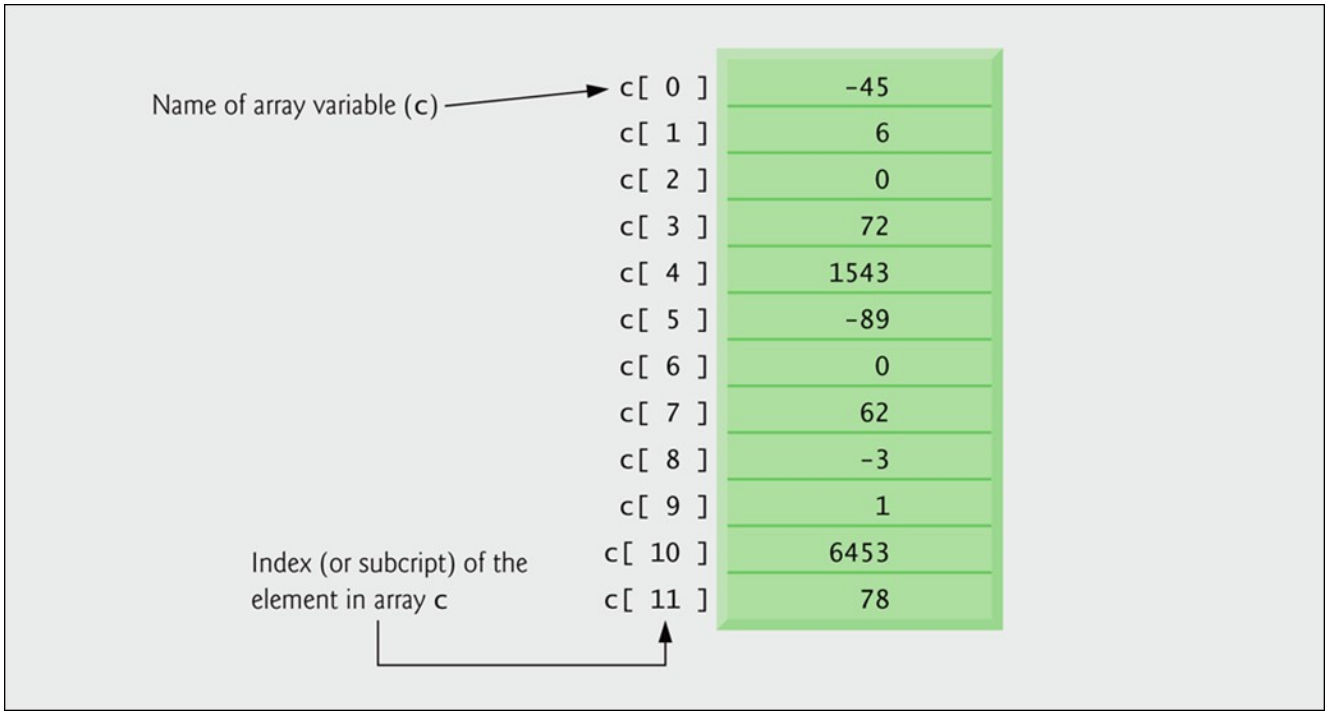

Fig. 8.1 | A 12-element array.

- This array contains 12 elements.

8.2 Arrays (Cont.)

Elements are accessed with an **array-access expression** that includes the name of the array, followed by the index of the particular element in **square brackets** (`[]`).

The first element in every array has **index zero** and is sometimes called the **zeroth element.**

An index must be a nonnegative integer and can be an expression.

Every array's length is stored in its `Length` property.

8.3 Declaring and Creating Arrays

Because arrays are objects, they are typically created with keyword new.

To create an array object, specify the type and the number of array elements as part of an **array-creation expression** that uses keyword `new`.

The following declaration and array-creation expression create an array object containing 12 `int` elements and store the array's reference in variable `c`:

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

8.3 Declaring and Creating Arrays (Cont.)

This task also can be performed as follows:

```
int[] c; // declare the array variable
c = new int[ 12 ]; // create the array; assign to
array variable
```

- The square brackets following the type `int` indicate that `c` is a variable that will refer to an array of `int`s.
- The array variable `c` receives the reference to a new array object of 12 `int` elements.

8.3 Declaring and Creating Arrays (Cont.)

Note: In C#, we cannot declare a constant array with the following syntax:

```
public const string[] values =
        { "value1", "value2", "value3", "value4"
```

This will give a compiler error because the `const` keyword is **used for values that are known at the compile-time**. But an array does not get initialized during compile-time, so the array's value is not known during the compile-time.

This error can be avoided by using the `readonly` keyword in C#. The `readonly` keyword is used to specify that a variable's value cannot be modified after initialization. The following code example shows us how we can **declare a constant array** with the `readonly` keyword in C#.

8.3 Declaring and Creating Arrays (Cont.)

```
class Program
    {
        public static readonly string[] Values =
                    { "Value1", "Value2", "Value3" };
        static void Main(string[] args)
        {
            foreach(var Value in Values)
            {
                Console.WriteLine(Value);
            }
        }
    }
```

8.3 Declaring and Creating Arrays (Cont.)

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.

Common Programming Error 8.1

In the declaration of a variable that will refer to an array, specifying the number of elements in the square brackets (e.g., `int[ 12 ] c;`) is a syntax error.

8.3 Declaring and Creating Arrays (Cont.)

An application can create several arrays in a single declaration.

For readability, it is better to write each array declaration in its own statement.

Good Programming Practice 8.1

For readability, declare only one variable per declaration. Keep each declaration on a separate line and include a comment describing the variable being declared.

- An application can declare arrays of value-type elements or reference-type elements.
- For example, every element of an `int` array is an `int` value, and every element of a `string` array is a reference to a `string` object.

8.3 Declaring and Creating Arrays (Cont.)

Resizing an Array

Though arrays are fixed-length entities, you can resize an array using the `static` `Array` method `Resize`.

`Resize` takes two arguments—the array to be resized and the new length. It performs the following operations:

- Creates a new array with the specified length
- Copies the contents of the old array into the new array
- Sets the array variable to reference the new array.

Any content that cannot fit into the new array is truncated.

```
int[] newArray = new int[5];
Array.Resize(ref newArray, 10); // newArray.Length = 10
```

Outline

Creating and Initializing an Array

- The application in Fig. 8.2 uses keyword `new` to create an array of 10 `int` elements.

`InitArray.cs`

(1 of 2)

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

Declare the array variable.

Create the array object in memory with 10 `int` elements.

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

Outline

InitArray.cs

(2 of 2)

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

```
Index   Value
    0       0
    1       0
    2       0
    3       0
    4       0
    5       0
    6       0
    7       0
    8       0
    9       0
```

Output the index number (`counter`) and the value (`array[ counter ]`) of each array element.

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

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

Outline

- 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.)

Outline

```
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
```

InitArray.cs

(2 of 2)

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.

Outline

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)

```
1  // Fig. 8.4: InitArray.cs
2  // Calculating values to be placed into the elements of an array.
3  using System;
4
5  public class InitArray
6  {
7     public static void Main( string[] args )
8     {
9        const int ARRAY_LENGTH = 10; // create a named constant
10       int[] array = new int[ ARRAY_LENGTH ]; // create array
11
12       // calculate value for each array element
13       for ( int counter = 0; counter < array.Length; counter++ )
14          array[ counter ] = 2 + 2 * counter;
15
```

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.)

Outline

InitArray.cs

(2 of 2)

```
16       Console.WriteLine( "{0}{1,8}", "Index", "Value" ); // headings
17
18       // output each array element's value
19       for ( int counter = 0; counter < array.Length; counter++ )
20          Console.WriteLine( "{0,5}{1,8}", counter, array[ counter ] );
21    } // end Main
22 } // 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.)

8.4 Examples Using Arrays (Cont.)

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

We declare constants with all capital letters by convention to make them stand out in the code.

Good Programming Practice 8.2

Constants also are called named constants. Applications using constants often are more readable than those that use literal values (e.g., 10)—a named constant such as ARRAY_LENGTH clearly indicates its purpose, whereas a literal value could have different meanings based on the context in which it is used. Another advantage to using named constants is that if the value of the constant must be changed, the change is necessary only in the declaration, thus reducing the cost of maintaining the code.

8.4 Examples Using Arrays (Cont.)

Good Programming Practice 8.2

Assigning a value to a named constant after it has been initialized is a compilation error.

Good Programming Practice 8.2

Attempting to declare a named constant without initializing it is a compilation error.

Outline

Summing the Elements of an Array

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

SumArray.cs

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

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

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

BarChart.cs

(1 of 2)

```
1  // Fig. 8.6: BarChart.cs
2  // Bar chart displaying application.
3  using System;
4
5  public class BarChart
6  {
7     public static void Main( string[] args )
8     {
9        int[] array = { 0, 0, 0, 0, 0, 0, 1, 2, 4, 2, 1 };
10
11       Console.WriteLine( "Grade distribution:" );
12
13       // for each array element, output a bar of the chart
14       for ( int counter = 0; counter < array.Length; counter++ )
15       {
16          // output bar labels ( "00-09: ", ..., "90-99: ", "100: " )
17          if ( counter == 10 )
18             Console.Write( "  100: " );
19          else
20             Console.Write( "{0:D2}-{1:D2}: ",
```

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

Outline

```
22
23          // display bar of asterisks
24          for ( int stars = 0; stars < array[ counter ]; stars++ )
25             Console.Write( "*" );
26
27          Console.WriteLine(); // start a new line of output
28       } // end outer for
29    } // end Main
30 } // 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.

Outline

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)

```
1  // Fig. 8.7: RollDie.cs
2  // Roll a six-sided die 6000 times.
3  using System;
4
5  public class RollDie
6  {
7     public static void Main( string[] args )
8     {
9        Random randomNumbers = new Random(); // random-number generator
10       int[] frequency = new int[ 7 ]; // array of frequency counters
11
12       // roll die 6000 times; use die value as frequency index
13       for ( int roll = 1; roll <= 6000; roll++ )
14          ++frequency[ randomNumbers.Next( 1, 7 ) ];
15
16       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.)

Outline

RollDie.cs

(2 of 2)

```
17
18       // output each array element's value
19       for ( int face = 1; face < frequency.Length; face++ )
20          Console.WriteLine( "{0,4}{1,10}", face, frequency[ face ] );
21    } // end Main
22 } // end class RollDie
```

```
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.)

Outline

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)

```
1  // Fig. 8.8: StudentPoll.cs
2  // Poll analysis application.
3  using System;
4
5  public class StudentPoll
6  {
7     public static void Main( string[] args )
8     {
9        // array of survey responses
10       int[] responses = { 1, 2, 6, 4, 8, 5, 9, 7, 8, 10, 1, 6, 3, 8, 6,
11          10, 3, 8, 2, 7, 6, 5, 7, 6, 8, 6, 7, 5, 6, 6, 5, 6, 7, 5, 6,
12          4, 8, 6, 8, 10 };
13       int[] frequency = new int[ 11 ]; // array of frequency counters
14
```

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.)

Outline

```
15       // for each answer, select responses element and use that value
16       // as frequency index to determine element to increment
17       for ( int answer = 0; answer < responses.Length; answer++ )
18          ++frequency[ responses[ answer ] ];
19
20       Console.WriteLine( "{0}{1,10}", "Rating", "Frequency" );
21
22       // output each array element's value
23       for ( int rating = 1; rating < frequency.Length; rating++ )
24          Console.WriteLine( "{0,6}{1,10}", rating, frequency[ rating ] );
25    } // end Main
26 } // 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.

8.4 Examples Using Arrays (Cont.)

Error-Prevention Tip 8.1

An exception indicates that an error has occurred in an application. You often can write code to recover from an exception and continue application execution, rather than abnormally terminating the application. Exception handling is discussed in Chapter 13.

Error-Prevention Tip 8.2

When writing code to loop through an array, ensure that the array index remains greater than or equal to 0 and less than the length of the array. The loop-continuation condition should prevent the accessing of elements outside this range.

Outline

Class Card

- Class `Card` (Fig. 8.9) represents a playing card that has a face and a suit.

`Card.cs`

```
1  // Fig. 8.9: Card.cs
2  // Card class represents a playing card.
3  public class Card
4  {
5     private string face; // face of card ("Ace", "Deuce", ...)
6     private string suit; // suit of card ("Hearts", "Diamonds", ...)
7
8     // two-parameter constructor initializes card's face and suit
9     public Card( string cardFace, string cardSuit )
10    {
11       face = cardFace; // initialize face of card
12       suit = cardSuit; // initialize suit of card
13    } // end two-parameter Card constructor
14
15    // return string representation of Card
16    public override string ToString()
17    {
18       return face + " of " + suit;
19    } // end method ToString
20 } // end class Card
```

Method `ToString` (lines 16–19) creates a `string` representing the card, such as `"Ace of Hearts"`.

Fig. 8.9 | `Card` class represents a playing card.

Class DeckOfCards

Outline

- Class DeckOfCards (Fig. 8.10) represents a deck of 52 Card objects.

DeckOfCards.cs

(1 of 3)

```
1  // Fig. 8.10: DeckOfCards.cs
2  // DeckOfCards class represents a deck of playing cards.
3  using System;
4
5  public class DeckOfCards
6  {
7     private Card[] deck; // array of Card objects
8     private int currentCard; // index of next Card to be dealt
9     private const int NUMBER_OF_CARDS = 52; // constant number of Cards
10    private Random randomNumbers; // random-number generator
11
12    // constructor fills deck of Cards
13    public DeckOfCards()
14    {
15       string[] faces = { "Ace", "Deuce", "Three", "Four", "Five", "Six",
16          "Seven", "Eight", "Nine", "Ten", "Jack", "Queen", "King" };
17       string[] suits = { "Hearts", "Diamonds", "Clubs", "Spades" };
18
19       deck = new Card[ NUMBER_OF_CARDS ]; // create array of Card objects
20       currentCard = 0; // set currentCard so deck[ 0 ] is dealt first
21       randomNumbers = new Random(); // create random-number generator
22
```

Store the Cards in an array, specifying the type Card in the declaration.

Instantiate the deck array to be of size NUMBER_OF_CARDS.

Fig. 8.10 | DeckOfCards class represents a deck of playing cards. (Part 1 of 3.)

Outline

```
23       // populate deck with Card objects
24       for ( int count = 0; count < deck.Length; count++ )
25          deck[ count ] =
26             new Card( faces[ count % 13 ], suits[ count / 13 ] );
27    } // end DeckOfCards constructor
28
29    // shuffle deck of Cards with one-pass algorithm
30    public void Shuffle()
31    {
32       // after shuffling, dealing should start at deck[ 0 ] again
33       currentCard = 0; // reinitialize currentCard
34
35       // for each Card, pick another random Card and swap them
36       for ( int first = 0; first < deck.Length; first++ )
37       {
38          // select a random number between 0 and 51
39          int second = randomNumbers.Next( NUMBER_OF_CARDS );
40
41          // swap current Card with randomly selected Card
42          Card temp = deck[ first ];
43          deck[ first ] = deck[ second ];
44          deck[ second ] = temp;
45       } // end for
46    } // end method Shuffle
```

DeckOfCards.cs

(2 of 3)

`count % 13` always results in a value from 0 to 12 and the calculation `count / 13` always results in a value from 0 to 3.

When swapping two values in an array, a temporary variable is needed to avoid losing one of the values.

Fig. 8.10 | `DeckOfCards` class represents a deck of playing cards. (Part 2 of 3.)

Outline

DeckOfCards.cs

(3 of 3)

```
47
48    // deal one Card
49    public Card DealCard()
50    {
51       // determine whether Cards remain to be dealt
52       if ( currentCard < deck.Length )
53          return deck[ currentCard++ ]; // return current Card in array
54       else
55          return null; // indicate that all Cards were dealt
56    } // end method DealCard
57 } // end class DeckOfCards
```

Fig. 8.10 | DeckOfCards class represents a deck of playing cards. (Part 3 of 3.)

Shuffling and Dealing Cards

Outline

- The application of Fig. 8.11 demonstrates the card dealing and shuffling capabilities of class `DeckOfCards`.

`DeckOfCardsTest.cs`

(1 of 2)

```
1  // Fig. 8.11: DeckOfCardsTest.cs
2  // Card shuffling and dealing application.
3  using System;
4
5  public class DeckOfCardsTest
6  {
7     // execute application
8     public static void Main( string[] args )
9     {
10       DeckOfCards myDeckOfCards = new DeckOfCards();
11       myDeckOfCards.Shuffle(); // place Cards in random order
12
13       // display all 52 Cards in the order in which they are dealt
14       for ( int i = 0; i < 52; i++ )
15       {
16          Console.Write( "{0,-19}", myDeckOfCards.DealCard() );
17
```

Fig. 8.11 | Card shuffling and dealing application. (Part 1 of 2.)

```
18          if ( ( i + 1 ) % 4 == 0 )
19             Console.WriteLine();
20       } // end for
21    } // end Main
22 } // end class DeckOfCardsTest
```

```
Eight of Clubs     Ten of Clubs       Ten of Spades      Four of Spades
Ace of Spades      Jack of Spades     Three of Spades    Seven of Spades
Three of Diamonds  Five of Clubs      Eight of Spades    Five of Hearts
Ace of Hearts      Ten of Hearts      Deuce of Hearts    Deuce of Clubs
Jack of Hearts     Nine of Spades     Four of Hearts     Seven of Clubs
Queen of Spades    Seven of Diamonds  Five of Diamonds   Ace of Clubs
Four of Clubs      Ten of Diamonds    Jack of Clubs      Six of Diamonds
Eight of Diamonds  King of Hearts     Three of Clubs     King of Spades
King of Diamonds   Six of Spades      Deuce of Spades    Five of Spades
Queen of Clubs     King of Clubs      Queen of Hearts    Seven of Hearts
Ace of Diamonds    Deuce of Diamonds  Four of Diamonds   Nine of Clubs
Queen of Diamonds  Jack of Diamonds   Six of Hearts      Nine of Diamonds
Nine of Hearts     Three of Hearts    Six of Clubs       Eight of Hearts
```

DeckOfCardsTest.cs

Fig. 8.11 | Card shuffling and dealing application. (Part 2 of 2.)

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.

Outline

- 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)

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

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

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

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.

Declare a variable by specifying a **data type and an identifier**, like this:

```
double myDbl;
```

It was also mentioned that you should **assign a value to a variable before you attempt to use it.** You can **declare and initialize** a variable in the same statement, like this:

```
double myDbl = 99.09;
```

The **compiler assumes** that floating-point number values are of type `double`.

8.6 foreach Statement (Cont.)

Implicitly Typed Local Variables

C# compiler to infer the type of a variable from an expression and use this type when declaring the variable by using the `var` keyword in place of the type, like this:

```
var myVariable = 99;
var myOtherVariable = "Hello";
```

To **distinguish such an initialization** from a simple assignment statement, the `var` keyword is used in place of the variable's type.

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.

```
var names = new[]{"John", "Diana", "James", "Francesca"};
```

Implicitly typed arrays are most useful when you are working with **anonymous types**, described in the **following lectures.**

8.6 foreach Statement (Cont.)

Implicitly Typed Local Variables- Errors

The following declaration is illegal and will cause a **compilation error**:

```
var yetAnotherVariable; // Error - compiler cannot infer type
```

You **must ensure that all the initializers have the same type**. This next example will cause the **compile-time error** "No best type found for implicitly typed array":

```
var bad = new[]{"John", "Diana", 99, 100};
```

However, the **`numbers`** array is an array of `double` because the constants 3.5 and 99.999 are both `double`, and the C# compiler can convert the `integer` values 1 and 2 to `double` values:

```
var numbers = new[]{1, 2, 3.5, 99.999};
```

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.

Outline

- 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)

```
1  // Fig. 8.13: PassArray.cs
2  // Passing arrays and individual array elements to methods.
3  using System;
4
5  public class PassArray
6  {
7     // Main creates array and calls ModifyArray and ModifyElement
8     public static void Main( string[] args )
9     {
10       int[] array = { 1, 2, 3, 4, 5 };
11
12       Console.WriteLine(
13          "Effects of passing reference to entire array:\n" +
14          "The values of the original array are:" );
15
16       // output original array elements
17       foreach ( int value in array )
18          Console.Write( "   {0}", value );
19
```

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

Outline

```
20       ModifyArray( array ); // pass array reference
21       Console.WriteLine( "\n\nThe values of the modified array are:" );
22
23       // output modified array elements
24       foreach ( int value in array )
25          Console.Write( "   {0}", value );
26
27       Console.WriteLine(
28          "\n\nEffects of passing array element value:\n" +
29          "array[3] before ModifyElement: {0}", array[ 3 ] );
30
31       ModifyElement( array[ 3 ] ); // attempt to modify array[ 3 ]
32       Console.WriteLine(
33          "array[3] after ModifyElement: {0}", array[ 3 ] );
34    } // end Main
35
36    // multiply each element of an array by 2
37    public static void ModifyArray( int[] array2 )
38    {
39       for ( int counter = 0; counter < array2.Length; counter++ )
40          array2[ counter ] *= 2;
41    } // end method ModifyArray
42
```

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

```
43    // multiply argument by 2
44    public static void ModifyElement( int element )
45    {
46       element *= 2;
47       Console.WriteLine(
48          "Value of element in ModifyElement: {0}", element );
49    } // end method ModifyElement
50 } // end class PassArray
```

PassArray.cs

(3 of 3)

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

```
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
```

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

8.8 Passing Arrays by Value and by Reference

Changes to the local copy of a value-type argument in a called method do not affect the original variable in the caller.

If the argument is of a reference type, the method makes a copy of the reference, not a copy of the actual object that is referenced.

Performance Tip 8.1

Passing arrays and other objects by reference makes sense for performance reasons. If arrays were passed by value, a copy of each element would be passed. For large, frequently passed arrays, this would waste time and would consume considerable storage for the copies of the arrays—both of these problems cause poor performance.

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

- 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`.

ArrayReferenceTest.cs


```
1  // Fig. 8.14: ArrayReferenceTest.cs
2  // Testing the effects of passing array references
3  // by value and by reference.
4  using System;
5
6  public class ArrayReferenceTest
7  {
8     public static void Main( string[] args )
9     {
10       // create and initialize firstArray
11       int[] firstArray = { 1, 2, 3 };
12
13       // copy the reference in variable firstArray
14       int[] firstArrayCopy = firstArray;
15
16       Console.WriteLine(
17          "Test passing firstArray reference by value" );
18
```

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

Outline

```
19       Console.Write( "\nContents of firstArray " +
20          "before calling FirstDouble:\n\t" );
21
22       // display contents of firstArray
23       for ( int i = 0; i < firstArray.Length; i++ )
24          Console.Write( "{0} ", firstArray[ i ] );
25
26       // pass variable firstArray by value to FirstDouble
27       FirstDouble( firstArray );
28
29       Console.Write( "\n\nContents of firstArray after " +
30          "calling FirstDouble\n\t" );
31
32       // display contents of firstArray
33       for ( int i = 0; i < firstArray.Length; i++ )
34          Console.Write( "{0} ", firstArray[ i ] );
35
36       // test whether reference was changed by FirstDouble
37       if ( firstArray == firstArrayCopy )
38          Console.WriteLine(
39             "\n\nThe references refer to the same array" );
40       else
41          Console.WriteLine(
42             "\n\nThe references refer to different arrays" );
43
```

ArrayReference Test.cs

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

Outline

ArrayReference Test.cs

(3 of 5)

```
44       // create and initialize secondArray
45       int[] secondArray = { 1, 2, 3 };
46
47       // copy the reference in variable secondArray
48       int[] secondArrayCopy = secondArray;
49
50       Console.WriteLine( "\nTest passing secondArray " +
51          "reference by reference" );
52
53       Console.Write( "\nContents of secondArray " +
54          "before calling SecondDouble:\n\t" );
55
56       // display contents of secondArray before method call
57       for ( int i = 0; i < secondArray.Length; i++ )
58          Console.Write( "{0} ", secondArray[ i ] );
59
60       // pass variable secondArray by reference to SecondDouble
61       SecondDouble( ref secondArray );
62
63       Console.Write( "\n\nContents of secondArray " +
64          "after calling SecondDouble:\n\t" );
65
66       // display contents of secondArray after method call
67       for ( int i = 0; i < secondArray.Length; i++ )
68          Console.Write( "{0} ", secondArray[ i ] );
```

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

Outline

```
69
70       // test whether reference was changed by SecondDouble
71       if ( secondArray == secondArrayCopy )
72          Console.WriteLine(
73             "\n\nThe references refer to the same array" );
74       else
75          Console.WriteLine(
76             "\n\nThe references refer to different arrays" );
77    } // end Main
78
79    // modify elements of array and attempt to modify reference
80    public static void FirstDouble( int[] array )
81    {
82       // double each element's value
83       for ( int i = 0; i < array.Length; i++ )
84          array[ i ] *= 2;
85
86       // create new object and assign its reference to array
87       array = new int[] { 11, 12, 13 };
88    } // end method FirstDouble
89
90    // modify elements of array and change reference array
91    // to refer to a new array
```

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

ArrayReference Test.cs

(4 of 5)

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

Outline

```
92    public static void SecondDouble( ref int[] array )
93    {
94       // double each element's value
95       for ( int i = 0; i < array.Length; i++ )
96          array[ i ] *= 2;
97
98       // create new object and assign its reference to array
99       array = new int[] { 11, 12, 13 };
100   } // end method SecondDouble
101} // 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.)

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

Software Engineering Observation 8.1

When a method receives a reference-type parameter by value, a copy of the object's reference is passed. This prevents a method from overwriting references passed to that method. In the vast majority of cases, protecting the caller's reference from modification is the desired behavior. If you encounter a situation where you truly want the called procedure to modify the caller's reference, pass the reference-type parameter using keyword `ref`—but, again, such situations are rare.

Software Engineering Observation 8.2

In C#, objects (including arrays) are effectively passed by reference, because references to objects are passed to called methods. A called method receiving a reference to an object in a caller can interact with, and possibly change, the caller's object.

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)

```
1  // Fig. 8.15: GradeBook.cs
2  // Grade book using an array to store test grades.
3  using System;
4
5  public class GradeBook
6  {
7     private int[] grades; // array of student grades
8
9     // auto-implemented property CourseName
10    public string CourseName { get; set; }
11
12    // two-parameter constructor initializes
13    // auto-implemented property CourseName and grades array
14    public GradeBook( string name, int[] gradesArray )
15    {
16       CourseName = name; // set CourseName to name
17       grades = gradesArray; // initialize grades array
18    } // end two-parameter GradeBook constructor
19
```

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.)

Outline

GradeBook.cs

(2 of 6)

```
20    // display a welcome message to the GradeBook user
21    public void DisplayMessage()
22    {
23       // auto-implemented property CourseName gets the name of course
24       Console.WriteLine( "Welcome to the grade book for\n{0}!\n",
25          CourseName );
26    } // end method DisplayMessage
27
28    // perform various operations on the data
29    public void ProcessGrades()
30    {
31       // output grades array
32       OutputGrades();
33
34       // call method GetAverage to calculate the average grade
35       Console.WriteLine( "\nClass average is {0:F}", GetAverage() );
36
37       // call methods GetMinimum and GetMaximum
38       Console.WriteLine( "Lowest grade is {0}\nHighest grade is {1}\n",
39          GetMinimum(), GetMaximum() );
40
41       // call OutputBarChart to display grade distribution chart
42       OutputBarChart();
43    } // end method ProcessGrades
44
```

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

Outline

GradeBook.cs

(3 of 6)

```
45    // find minimum grade
46    public int GetMinimum()
47    {
48       int lowGrade = grades[ 0 ]; // assume grades[ 0 ] is smallest
49
50       // loop through grades array
51       foreach ( int grade in grades )
52       {
53          // if grade lower than lowGrade, assign it to lowGrade
54          if ( grade < lowGrade )
55             lowGrade = grade; // new lowest grade
56       } // end for
57
58       return lowGrade; // return lowest grade
59    } // end method GetMinimum
60
61    // find maximum grade
62    public int GetMaximum()
63    {
64       int highGrade = grades[ 0 ]; // assume grades[ 0 ] is largest
65
```

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.)

Outline

GradeBook.cs

(4 of 6)

```
66       // loop through grades array
67       foreach ( int grade in grades )
68       {
69          // if grade greater than highGrade, assign it to highGrade
70          if ( grade > highGrade )
71             highGrade = grade; // new highest grade
72       } // end for
73
74       return highGrade; // return highest grade
75    } // end method GetMaximum
76
77    // determine average grade for test
78    public double GetAverage()
79    {
80       int total = 0; // initialize total
81
82       // sum grades for one student
83       foreach ( int grade in grades )
84          total += grade;
85
86       // return average of grades
87       return ( double ) total / grades.Length;
88    } // end method GetAverage
89
```

Total the grades using a `foreach` statement.

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

Outline

GradeBook.cs

(5 of 6)

```
90    // output bar chart displaying grade distribution
91    public void OutputBarChart()
92    {
93       Console.WriteLine( "Grade distribution:" );
94
95       // stores frequency of grades in each range of 10 grades
96       int[] frequency = new int[ 11 ];
97
98       // for each grade, increment the appropriate frequency
99       foreach ( int grade in grades )
100         ++frequency[ grade / 10 ];
101
102      // for each grade frequency, display bar in chart
103      for ( int count = 0; count < frequency.Length; count++ )
104      {
105         // output bar label ( "00-09: ", ..., "90-99: ", "100: " )
106         if ( count == 10 )
107            Console.Write( "  100: " );
108         else
109            Console.Write( "{0:D2}-{1:D2}: ",
110               count * 10, count * 10 + 9 );
111
```

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)

```
112         // display bar of asterisks
113         for ( int stars = 0; stars < frequency[ count ]; stars++ )
114            Console.Write( "*" );
115
116         Console.WriteLine(); // start a new line of output
117      } // end outer for
118   } // end method OutputBarChart
119
120   // output the contents of the grades array
121   public void OutputGrades()
122   {
123      Console.WriteLine( "The grades are:\n" );
124
125      // output each student's grade
126      for ( int student = 0; student < grades.Length; student++ )
127         Console.WriteLine( "Student {0,2}: {1,3}",
128            student + 1, grades[ student ] );
129   } // end method OutputGrades
130} // 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.)

Outline

Class GradeBookTest That Demonstrates Class GradeBook

- The application in Fig. 8.16 demonstrates class GradeBook.

GradeBookTest.cs

(1 of 4)

```
1  // Fig. 8.16: GradeBookTest.cs
2  // Create GradeBook object using an array of grades.
3  public class GradeBookTest
4  {
5     // Main method begins application execution
6     public static void Main( string[] args )
7     {
8        // one-dimensional array of student grades
9        int[] gradesArray = { 87, 68, 94, 100, 83, 78, 85, 91, 76, 87 };
10
11       GradeBook myGradeBook = new GradeBook(
12          "CS101 Introduction to C# Programming", gradesArray );
13       myGradeBook.DisplayMessage();
14       myGradeBook.ProcessGrades();
15    } // end Main
16 } // end class GradeBookTest
```

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

Outline

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.)

Software Engineering Observation 8.3

A test harness (or test application) is responsible for creating an object of the class being tested and providing it with data. This data could come from any of several sources. Test data can be placed directly into an array with an array initializer, it can come from the user at the keyboard, it can come from a file (as you'll see in Chapter 19). After passing this data to the class's constructor to instantiate the object, the test harness should call the object to test its methods and manipulate its data. Gathering data in the test harness like this allows the class to manipulate data from several sources.

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.

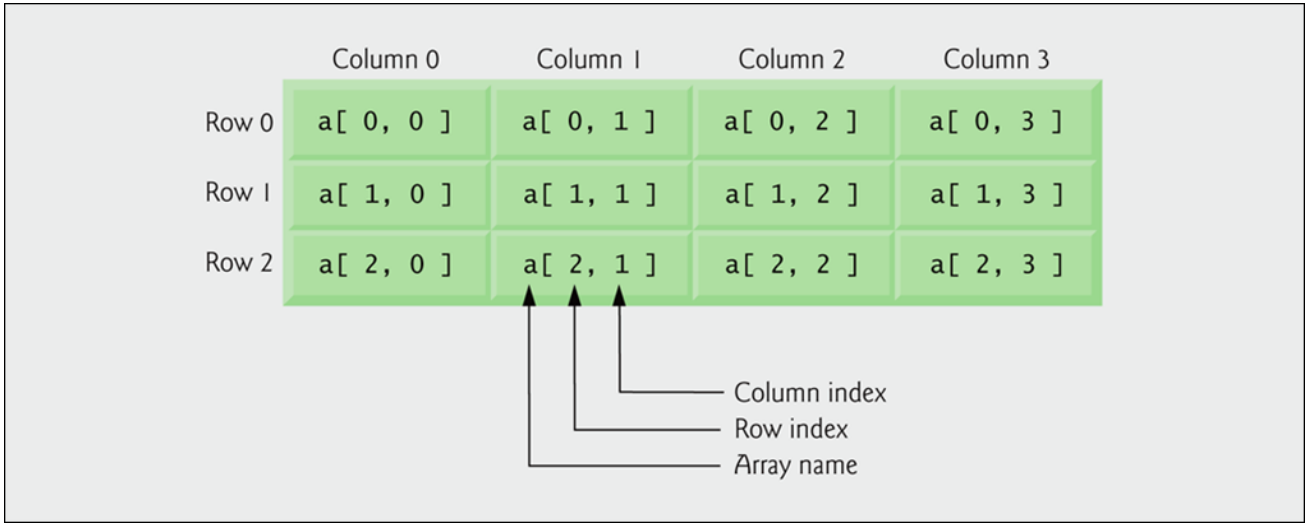

Fig. 8.17 | Rectangular array with three rows and four columns.

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.

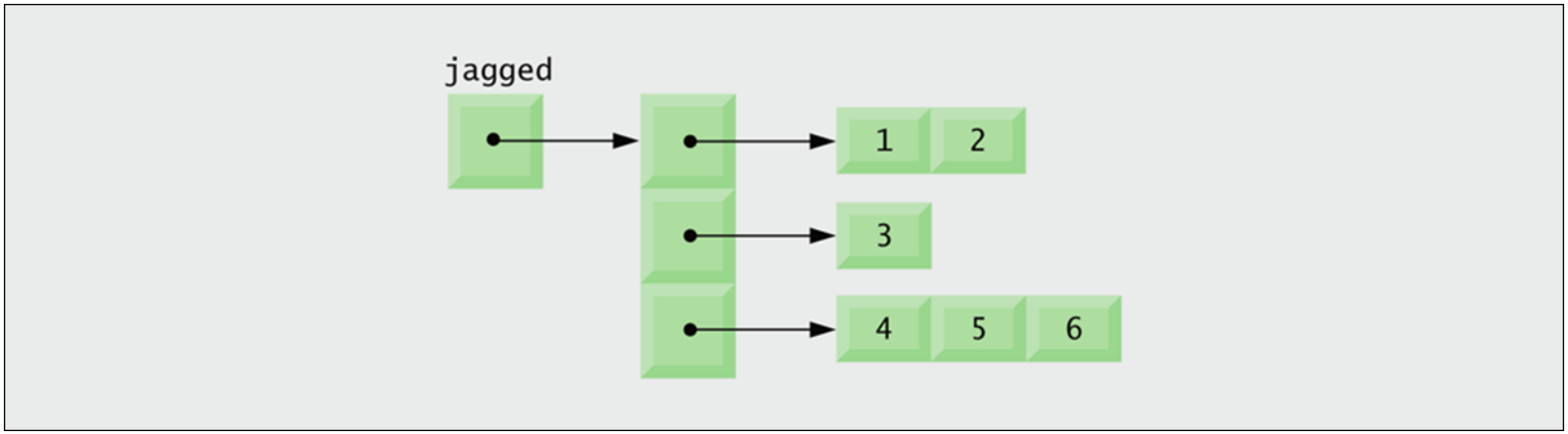

Fig. 8.18 | Jagged array with three rows of different lengths.

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

Outline

- 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)

```
1  // Fig. 8.19: InitArray.cs
2  // Initializing rectangular and jagged arrays.
3  using System;
4
5  public class InitArray
6  {
7     // create and output rectangular and jagged arrays
8     public static void Main( string[] args )
9     {
10       // with rectangular arrays,
11       // every column must be the same length.
12       int[ , ] rectangular = { { 1, 2, 3 }, { 4, 5, 6 } };
13
14       // with jagged arrays,
15       // we need to use "new int[]" for every row,
16       // but every column does not need to be the same length.
17       int[][] jagged = { new int[] { 1, 2 },
18                          new int[] { 3 },
19                          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.)

Outline

InitArray.cs

(2 of 3)

```
20
21       OutputArray( rectangular ); // displays array rectangular by row
22       Console.WriteLine(); // output a blank line
23       OutputArray( jagged ); // displays array jagged by row
24    } // end Main
25
26    // output rows and columns of a rectangular array
27    public static void OutputArray( int[ , ] array )
28    {
29       Console.WriteLine( "Values in the rectangular array by row are" );
30
31       // loop through array's rows
32       for ( int row = 0; row < array.GetLength( 0 ); row++ )
33       {
34          // loop through columns of current row
35          for ( int column = 0; column < array.GetLength( 1 ); column++ )
36             Console.Write( "{0}  ", array[ row, column ] );
37
38          Console.WriteLine(); // start new line of output
39       } // end outer for
40    } // end method OutputArray
41
```

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.)

Outline

```
42    // output rows and columns of a jagged array
43    public static void OutputArray( int[][] array )
44    {
45       Console.WriteLine( "Values in the jagged array by row are" );
46
47       // loop through each row
48       foreach ( var row in array )
49       {
50          // loop through each element in current row
51          foreach ( var element in row )
52             Console.Write( "{0}  ", element );
53
54          Console.WriteLine(); // start new line of output
55       } // end outer foreach
56    } // end method OutputArray
57 } // 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  // Fig. 8.20: GradeBook.cs
2  // Grade book using rectangular array to store grades.
3  using System;
4
5  public class GradeBook
6  {
7     private int[ , ] grades; // rectangular array of student grades
8
9     // auto-implemented property CourseName
10    public string CourseName { get; set; }
11
12    // two-parameter constructor initializes
13    // auto-implemented property CourseName and grades array
14    public GradeBook( string name, int[ , ] gradesArray )
15    {
16       CourseName = name; // set CourseName to name
17       grades = gradesArray; // initialize grades array
18    } // end two-parameter GradeBook constructor
19
```

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

Outline

GradeBook.cs

(2 of 7)

```
20    // display a welcome message to the GradeBook user
21    public void DisplayMessage()
22    {
23       // auto-implemented property CourseName gets the name of course
24       Console.WriteLine( "Welcome to the grade book for\n{0}!\n",
25          CourseName );
26    } // end method DisplayMessage
27
28    // perform various operations on the data
29    public void ProcessGrades()
30    {
31       // output grades array
32       OutputGrades();
33
34       // call methods GetMinimum and GetMaximum
35       Console.WriteLine( "\n{0} {1}\n{2} {3}\n",
36          "Lowest grade in the grade book is", GetMinimum(),
37          "Highest grade in the grade book is", GetMaximum() );
38
39       // output grade distribution chart of all grades on all tests
40       OutputBarChart();
41    } // end method ProcessGrades
42
```

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

Outline

GradeBook.cs

(3 of 7)

```
43    // find minimum grade
44    public int GetMinimum()
45    {
46       // assume first element of grades array is smallest
47       int lowGrade = grades[ 0, 0 ];
48
49       // loop through elements of rectangular grades array
50       foreach ( int grade in grades )
51       {
52          // if grade less than lowGrade, assign it to lowGrade
53          if ( grade < lowGrade )
54             lowGrade = grade;
55       } // end foreach
56
57       return lowGrade; // return lowest grade
58    } // end method GetMinimum
59
60    // find maximum grade
61    public int GetMaximum()
62    {
63       // assume first element of grades array is largest
64       int highGrade = grades[ 0, 0 ];
65
```

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

GradeBook.cs

(4 of 7)

```
66       // loop through elements of rectangular grades array
67       foreach ( int grade in grades )
68       {
69          // if grade greater than highGrade, assign it to highGrade
70          if ( grade > highGrade )
71             highGrade = grade;
72       } // end foreach
73
74       return highGrade; // return highest grade
75    } // end method GetMaximum
76
77    // determine average grade for particular student
78    public double GetAverage( int student )
79    {
80       // get the number of grades per student
81       int amount = grades.GetLength( 1 );
82       int total = 0; // initialize total
83
84       // sum grades for one student
85       for ( int exam = 0; exam < amount; exam++ )
86          total += grades[ student, exam ];
87
88       // return average of grades
89       return ( double ) total / amount;
90    } // end method GetAverage
91
```

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.)

Outline

GradeBook.cs

(5 of 7)

```
92    // output bar chart displaying overall grade distribution
93    public void OutputBarChart()
94    {
95       Console.WriteLine( "Overall grade distribution:" );
96
97       // stores frequency of grades in each range of 10 grades
98       int[] frequency = new int[ 11 ];
99
100      // for each grade in GradeBook, increment the appropriate frequency
101      foreach ( int grade in grades )
102      {
103         ++frequency[ grade / 10 ];
104      } // end foreach
105
106      // for each grade frequency, display bar in chart
107      for ( int count = 0; count < frequency.Length; count++ )
108      {
109         // output bar label ( "00-09: ", ..., "90-99: ", "100: " )
110         if ( count == 10 )
111            Console.Write( "  100: " );
112         else
113            Console.Write( "{0:D2}-{1:D2}: ",
114               count * 10, count * 10 + 9 );
115
```

Same as the frequency for the one-dimensional array.

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

Outline

GradeBook.cs

(6 of 7)

```
116         // display bar of asterisks
117         for ( int stars = 0; stars < frequency[ count ]; stars++ )
118            Console.Write( "*" );
119
120         Console.WriteLine(); // start a new line of output
121      } // end outer for
122   } // end method OutputBarChart
123
124   // output the contents of the grades array
125   public void OutputGrades()
126   {
127      Console.WriteLine( "The grades are:\n" );
128      Console.Write( "            " ); // align column heads
129
130      // create a column heading for each of the tests
131      for ( int test = 0; test < grades.GetLength( 1 ); test++ )
132         Console.Write( "Test {0}  ", test + 1 );
133
134      Console.WriteLine( "Average" ); // student average column heading
135
```

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

Outline

GradeBook.cs

(7 of 7)

```
136      // create rows/columns of text representing array grades
137      for ( int student = 0; student < grades.GetLength( 0 ); student++ )
138      {
139         Console.Write( "Student {0,2}", student + 1 );
140
141         // output student's grades
142         for ( int grade = 0; grade < grades.GetLength( 1 ); grade++ )
143            Console.Write( "{0,8}", grades[ student, grade ] );
144
145         // call method GetAverage to calculate student's average grade;
146         // pass row number as the argument to GetAverage
147         Console.WriteLine( "{0,9:F}", GetAverage( student ) );
148      } // end outer for
149   } // end method OutputGrades
150} // end class GradeBook
```

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

Outline

Class `GradeBookTest` That Demonstrates Class `GradeBook`

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

`GradeBookTest.cs`


```
1  // Fig. 8.21: GradeBookTest.cs
2  // Create GradeBook object using a rectangular array of grades.
3  public class GradeBookTest
4  {
5     // Main method begins application execution
6     public static void Main( string[] args )
7     {
8        // rectangular array of student grades
9        int[ , ] gradesArray = { { 87, 96, 70 },
10                                { 68, 87, 90 },
11                                { 94, 100, 90 },
12                                { 100, 81, 82 },
13                                { 83, 65, 85 },
14                                { 78, 87, 65 },
15                                { 85, 75, 83 },
16                                { 91, 94, 100 },
17                                { 76, 72, 84 },
18                                { 87, 93, 73 } };
19
```

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.)

Outline

```
20         GradeBook myGradeBook = new GradeBook(
21            "CS101 Introduction to C# Programming", gradesArray );
22         myGradeBook.DisplayMessage();
23         myGradeBook.ProcessGrades();
24      } // end Main
25   } // 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
```

GradeBookTest.cs

(2 of 3)

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

- **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.

`ParamArrayTest.cs`

(1 of 3)

```
1  // Fig. 8.22: ParamArrayTest.cs
2  // Using variable-length argument lists.
3  using System;
4
5  public class ParamArrayTest
6  {
7     // calculate average
8     public static double Average( params double[] numbers )
9     {
10       double total = 0.0; // initialize total
11
```

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

Outline

```
12       // calculate total using the foreach statement
13       foreach ( double d in numbers )
14          total += d;
15
16       return total / numbers.Length;
17    } // end method Average
18
19    public static void Main( string[] args )
20    {
21       double d1 = 10.0;
22       double d2 = 20.0;
23       double d3 = 30.0;
24       double d4 = 40.0;
25
26       Console.WriteLine(
27          "d1 = {0:F1}\nd2 = {1:F1}\nd3 = {2:F1}\nd4 = {3:F1}\n",
28          d1, d2, d3, d4 );
29
```

ParamArrayTest.cs

(2 of 3)

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

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

Outline

ParamArrayTest.cs

(3 of 3)

```
30       Console.WriteLine( "Average of d1 and d2 is {0:F1}",
31          Average( d1, d2 ) );
32       Console.WriteLine( "Average of d1, d2 and d3 is {0:F1}",
33          Average( d1, d2, d3 ) );
34       Console.WriteLine( "Average of d1, d2, d3 and d4 is {0:F1}",
35          Average( d1, d2, d3, d4 ) );
36    } // end Main
37 } // end class ParamArrayTest
```

```
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)

```
1  // Fig. 8.23: InitArray.cs
2  // Using command-line arguments to initialize an array.
3  using System;
4
5  public class InitArray
6  {
7     public static void Main( string[] args )
8     {
9        // check number of command-line arguments
10       if ( args.Length != 3 )
11          Console.WriteLine(
12             "Error: Please re-enter the entire command, including\n" +
13             "an array size, initial value and increment." );
14       else
15       {
16          // get array size from first command-line argument
17          int arrayLength = Convert.ToInt32( args[ 0 ] );
18          int[] array = new int[ arrayLength ]; // create array
19
```

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

```
20          // get initial value and increment from command-line argument
21          int initialValue = Convert.ToInt32( args[ 1 ] );
22          int increment = Convert.ToInt32( args[ 2 ] );
23
24          // calculate value for each array element
25          for ( int counter = 0; counter < array.Length; counter++ )
26             array[ counter ] = initialValue + increment * counter;
27
28          Console.WriteLine( "{0}{1,8}", "Index", "Value" );
29
30          // display array index and value
31          for ( int counter = 0; counter < array.Length; counter++ )
32             Console.WriteLine( "{0,5}{1,8}", counter, array[ counter ] );
33       } // end else
34    } // end Main
35 } // end class InitArray
```

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

InitArray.cs

(2 of 3)

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 2 of 3.)

Outline

```
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
```

InitArray.cs

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