3b

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**.

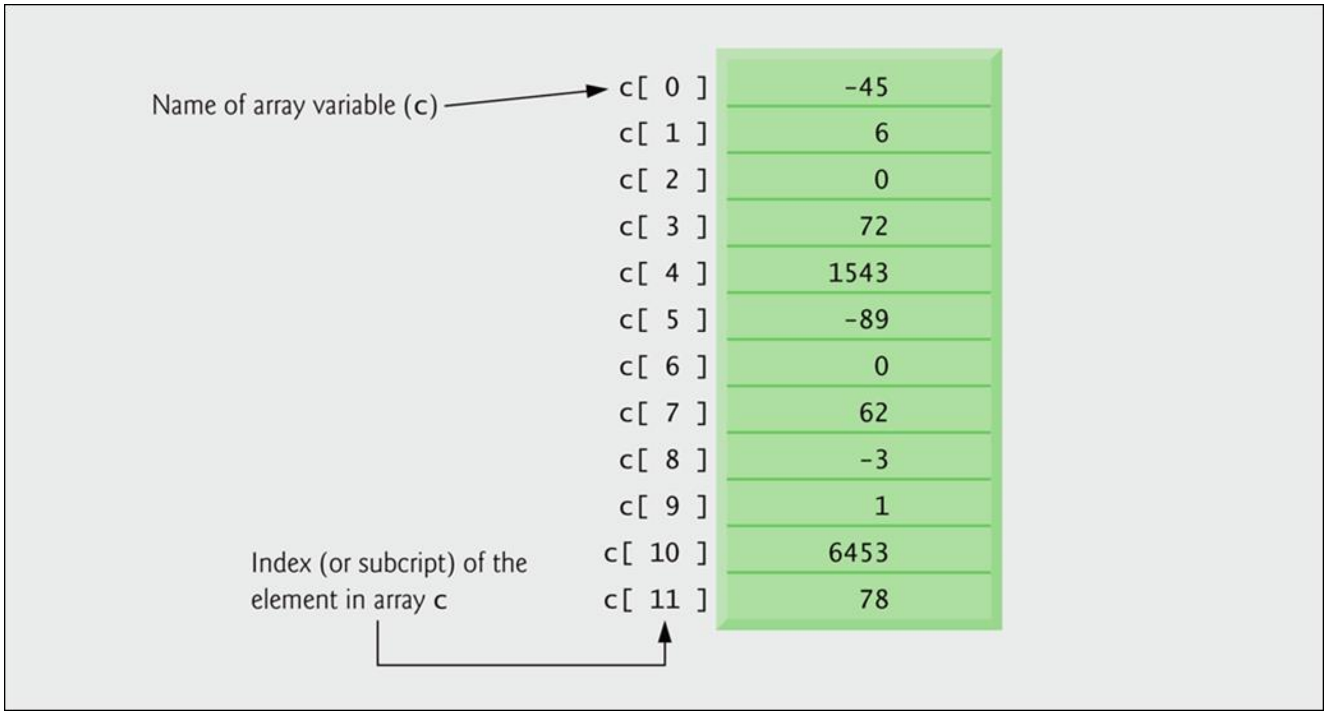

Fig. 8.1 | A 12-element array.

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.)

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
```

Creating and Initializing an Array

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


```
// Fig. 8.2: InitArray.cs
// Creating an array.
using System;

public class InitArray
{
   public static void Main( string[] args )
   {
      int[] array; // declare array named array

      // create the space for array and initialize to default zeros
      array = new int[ 10 ]; // 10 int elements

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

```

Declare the array variable.

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

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

InitArray.cs

(2 of 2)

```
      // 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
```

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

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

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

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.)

```
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.

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


```
// 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.)

Outline

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.)

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.

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

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


```
// 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.)

Outline

```
         // 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
```

BarChart.cs

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

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

Fig. 8.6 | 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.

```
// 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


```
      // 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.

```
// 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
```

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

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

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.

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***

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

DeckOfCards.cs


```
// Fig. 8.10: DeckOfCards.cs
// DeckOfCards class represents a deck of playing cards.
using System;

public class DeckOfCards
{
   private Card[] deck; // array of Card objects
   private int currentCard; // index of next Card to be dealt
   private const int NUMBER_OF_CARDS = 52; // constant number of Cards
   private Random randomNumbers; // random-number generator

   // constructor fills deck of Cards
   public DeckOfCards()
   {
      string[] faces = { "Ace", "Deuce", "Three", "Four", "Five", "Six",
         "Seven", "Eight", "Nine", "Ten", "Jack", "Queen", "King" };
      string[] suits = { "Hearts", "Diamonds", "Clubs", "Spades" };

      deck = new Card[ NUMBER_OF_CARDS ]; // create array of Card objects
      currentCard = 0; // set currentCard so deck[ 0 ] is dealt first
      randomNumbers = new Random(); // create random-number generator

```

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.)

```
      // populate deck with Card objects
      for ( int count = 0; count < deck.Length; count++ )
         deck[ count ] =
            new Card( faces[ count % 13 ], suits[ count / 13 ] );
   } // end DeckOfCards constructor

   // shuffle deck of Cards with one-pass algorithm
   public void Shuffle()
   {
      // after shuffling, dealing should start at deck[ 0 ] again
      currentCard = 0; // reinitialize currentCard

      // for each Card, pick another random Card and swap them
      for ( int first = 0; first < deck.Length; first++ )
      {
         // select a random number between 0 and 51
         int second = randomNumbers.Next( NUMBER_OF_CARDS );

         // swap current Card with randomly selected Card
         Card temp = deck[ first ];
         deck[ first ] = deck[ second ];
         deck[ second ] = temp;
      } // end for
   } // 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)

```
   // deal one Card
   public Card DealCard()
   {
      // determine whether Cards remain to be dealt
      if ( currentCard < deck.Length )
         return deck[ currentCard++ ]; // return current Card in array
      else
         return null; // indicate that all Cards were dealt
   } // end method DealCard
} // end class DeckOfCards
```

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

Shuffling and Dealing Cards

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

DeckOfCardsTest .cs

(1 of 2)

```
// Fig. 8.11: DeckOfCardsTest.cs
// Card shuffling and dealing application.
using System;

public class DeckOfCardsTest
{
   // execute application
   public static void Main( string[] args )
   {
      DeckOfCards myDeckOfCards = new DeckOfCards();
      myDeckOfCards.Shuffle(); // place Cards in random order

      // display all 52 Cards in the order in which they are dealt
      for ( int i = 0; i < 52; i++ )
      {
         Console.Write( "{0,-19}", myDeckOfCards.DealCard() );

```

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

```
            if ( ( i + 1 ) % 4 == 0 )
               Console.WriteLine();
         } // end for
      } // end Main
   } // 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

(1 of 2)

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.

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

ForEachTest.cs


```
// 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.

- 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.

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

PassArray.cs


```
// 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.)

```
      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.)

PassArray.cs

(3 of 3)

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

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.

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`


```
// 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.)

```
      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" );

```

ArrayReferenceTest.cs

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 ] );
```

ArrayReferenceTest.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`.

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

Outline

ArrayReferenceTest.cs

(5 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
```

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

```
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
```

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`***

- 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`


```
// 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.)

Outline

```
   // 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

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.)

```
         // 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
```

GradeBook.cs

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`.


```
// 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


```
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`.

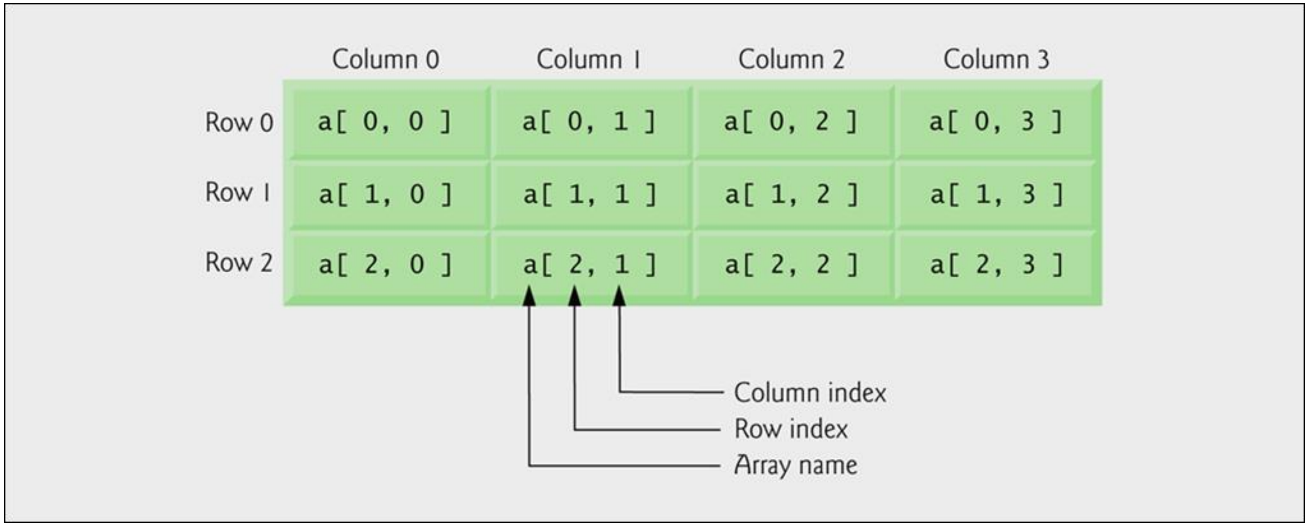

Fig. 8.17 | Rectangular array with three rows and four columns.

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.

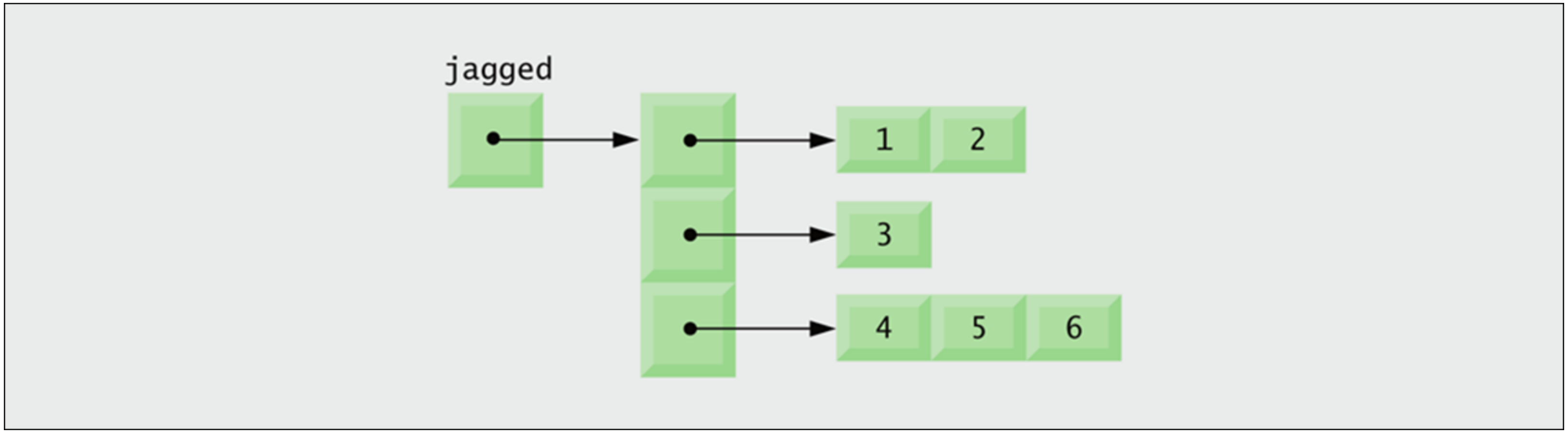

Fig. 8.18 | Jagged array with three rows of different lengths.

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
```

Outline

InitArray.cs

(1 of 3)

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.

```
// 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.)

Outline

```
      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

```

InitArray.cs

(2 of 3)

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

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`***

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.

```
// 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

```


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

GradeBook.cs

(3 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 ];

```

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.)

GradeBook.cs


```
      // 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

```

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.

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

Outline

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`.


```
// 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.)

Outline

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.)

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 `double`s.

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

Outline

ParamArrayTest.cs

(3 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
```

```
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.

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.)

```
         // 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
```

```
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.)

```
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.)