Arrays; Introduction to Exception Handling

Chapter 8 of Visual C# How to Program, 6/e

OBJECTIVES

In this chapter you'll:

- Use arrays to store data in and retrieve data from lists and tables of values.
- Declare arrays, initialize arrays and refer to individual elements of arrays.
- Iterate through arrays with the `foreach` statement.
- Use `var` to declare implicitly typed local variables and let the compiler infer their types from their initializer values.
- Use exception handling to process runtime problems.
- Declare C# 6 getter-only auto-implemented properties.
- Initialize auto-implemented properties with C# 6 auto-property initializers.
- Pass arrays to methods.
- Declare and manipulate multidimensional arrays—both rectangular and jagged.
- Write methods that use variable-length argument lists.
- Read command-line arguments into an app.

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're 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*.

8.2 Arrays (cont.)

- 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, which is called the element’s **index**.
- Figure 8.1 shows a logical representation of an integer array called `c` containing sample values.

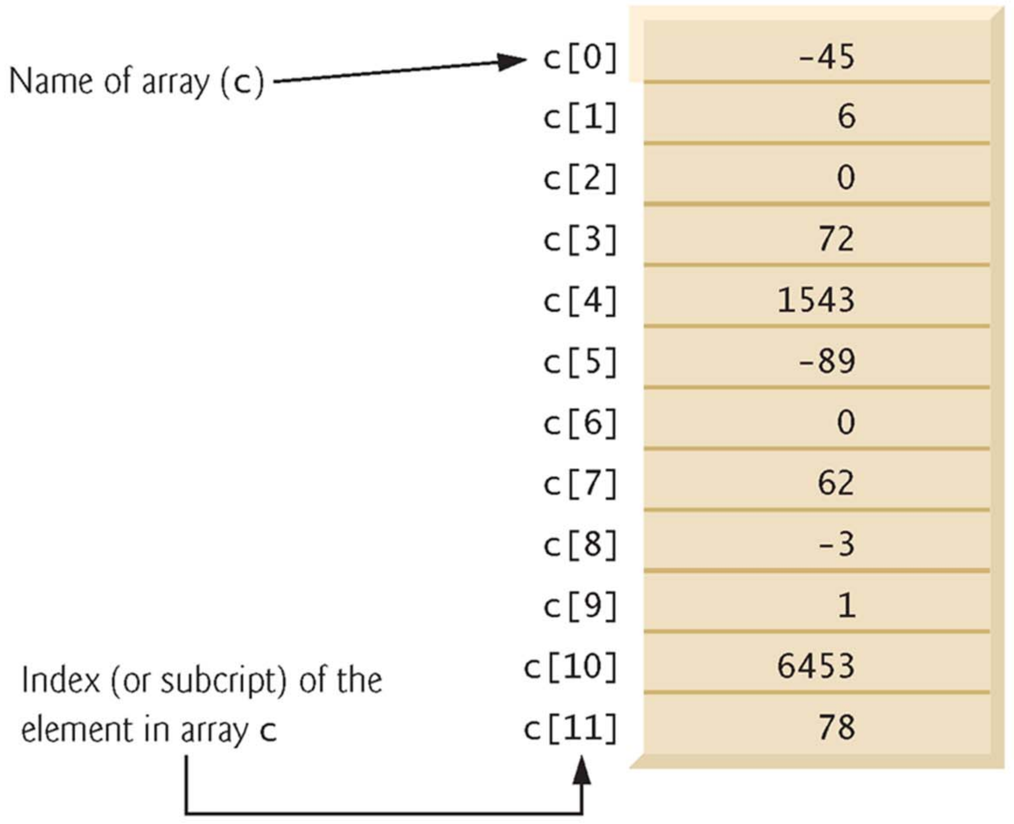


Fig. 8.1 | A 12-element array.

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

- Since arrays are objects, they're 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.)

- Creating the array 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 `int` indicate that `c` 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's 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 an array variable declaration, 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.)

Resizing an Array

- Though arrays are fixed-length entities, you can use the `static` `Array` method `Resize` which takes two arguments—the array to be resized and the new length—to create 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 without warning.

8.4 Examples Using Arrays

8.4.1 Creating and Initializing an Array

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

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

class InitArray
{
   static void Main()
   {
      // create the space for array and initialize to default zeros
      int[] array = new int[5]; // array contains 5 int elements

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

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

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

```
Index   Value
    0       0
    1       0
    2       0
    3       0
    4       0
```

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

8.4 Examples Using Arrays (cont.)

8.4.2 Using an Array Initializer

- An app 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.
- The app in Fig. 8.3 initializes an integer array with 5 values (line 10) and displays the array in tabular format.

```csharp
// Fig. 8.3: InitArray.cs
// Initializing the elements of an array with an array initializer.
using System;

class InitArray
{
   static void Main()
   {
      // initializer list specifies the value of each element
      int[] array = {32, 27, 64, 18, 95};

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

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

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

```
Index   Value
    0      32
    1      27
    2      64
    3      18
    4      95
```

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

8.4 Examples Using Arrays (cont.)

8.4.3 Calculating a Value to Store in Each Array Element

- The app in Fig. 8.4 creates a 5-element array and assigns to each element one of the even integers from 2 to 10 (2, 4, 6, 8, 10).
- Constants must be initialized when they're declared and cannot be modified thereafter.
- Constants use the same Pascal Case naming conventions as classes, methods and properties.

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

class InitArray
{
   static void Main()
   {
      const int ArrayLength = 5; // create a named constant
      int[] array = new int[ArrayLength]; // create array

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

```

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

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

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

```
Index   Value
    0       2
    1       4
    2       6
    3       8
    4      10
```

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

Good Programming Practice 8.1

Constants are also called ***named constants****. Apps using constants often are more readable than those that use literal values (e.g.,* `5`*)—a named constant such as* `ArrayLength` *clearly indicates its purpose, whereas the literal value* `5` *could have different meanings based on the context in which it's used. Another advantage to using named constants is that if the constant's value must be changed, the change is necessary only in the declaration, thus reducing code-maintenance costs.*

Good Programming Practice 8.2

Defining the size of an array as a named constant instead of a literal makes code clearer. This technique eliminates so-called ***magic numbers****. For example, repeatedly mentioning the size 5 in array-processing code for a five-element array gives the number 5 an artificial significance and can be confusing when the program includes other 5s that have nothing to do with the array size.*

Common Programming Error 8.2

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

Common Programming Error 8.3

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

8.4 Examples Using Arrays (Cont.)

8.4.4 Summing the Elements of an Array

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

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

class SumArray
{
   static void Main()
   {
      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]; // add element value to total
      }

      Console.WriteLine($"Total of array elements: {total}");
   }
}
```

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

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

8.4 Examples Using Arrays (Cont.)

8.4.5 Iterating Through Arrays with foreach

- 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.

8.4 Examples Using Arrays (Cont.)

8.4.5 Iterating Through Arrays with `foreach`

- The type of the iteration variable must be consistent with the type of the elements in the array.
- The iteration variable represents successive values in the array on successive iterations of the `foreach` statement.

8.4 Examples Using Arrays (Cont.)

- Figure 8.6 uses the `foreach` statement to calculate the sum of the integers in an array of student grades.
- 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.

Common Programming Error 8.4

Any attempt to change the iteration variable's value in the body of a `foreach` *statement results in a compilation error.*

```
// Fig. 8.6: ForEachTest.cs
// Using the foreach statement to total integers in an array.
using System;

class ForEachTest
{
   static void Main()
   {
      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: {total}");
   }
}
```

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

Fig. 8.6 | Using the `foreach` statement to total integers in an array.

Common Programming Error 8.5

Attempting to modify an array element's value using a `foreach` *statement's iteration variable is a logic error—the iteration variable can be used only to* access *each array element's value, not modify it.*

8.4 Examples Using Arrays (Cont.)

8.4.6 Using Bar Charts to Display Array Data Graphically; Introducing Type Inference with var

- The app in Fig. 8.7 stores grade distribution data in an array of 11 elements, each corresponding to a category of grades.
- `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.

```
// Fig. 8.7: BarChart.cs
// Bar chart displaying app.
using System;

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

      Console.WriteLine("Grade distribution:");

```

Fig. 8.7 | Bar chart displaying app. (Part 1 of 3.)

```
        // for each array element, output a bar of the chart
        for (var counter = 0; counter < array.Length; ++counter)
        {
           // output bar labels ("00-09: ", ..., "90-99: ", "100: ")
           if (counter == 10)
           {
              Console.Write("  100: ");
           }
           else
           {
              Console.Write($"{counter * 10:D2}-{counter * 10 + 9:D2}: ");
           }

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

           Console.WriteLine(); // start a new line of output
        }
     }
  }
```

Fig. 8.7 | Bar chart displaying app. (Part 2 of 3.)

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

Fig. 8.7 | Bar chart displaying app. (Part 3 of 3.)

8.4 Examples Using Arrays (Cont.)

8.4.7 Using the Elements of an Array as Counters

- An array version of our die-rolling app from Fig. 7.7 is shown in Fig. 8.8.

Common Programming Error 8.6

Initializer lists can be used with both arrays and collections. If an implicitly typed local variable is initialized via an initializer list without `new[]`, *a compilation error occurs, because the compiler cannot infer whether the variable should be an array or a collection. We use a* `List` *collection in Chapter 9 and cover collections in detail in Chapter 21.*

```
// Fig. 8.8: RollDie.cs
// Roll a six-sided die 60,000,000 times.
using System;

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

      // roll die 60,000,000 times; use die value as frequency index
      for (var roll = 1; roll <= 60000000; ++roll)
      {
         ++frequency[randomNumbers.Next(1, 7)];
      }
```

Fig. 8.8 | Roll a six-sided die 60,000,000 times. (Part 1 of 2.)

```

         Console.WriteLine($"{"Face"}{"Frequency",10}");

         // output each array element's value
         for (var face = 1; face < frequency.Length; ++face)
         {
            Console.WriteLine($"{face,4}{frequency[face],10}");
         }
      }
   }
```

```
Face Frequency
   1  10004131
   2   9998200
   3  10003734
   4   9999332
   5   9999792
   6   9994811
```

Fig. 8.8 | Roll a six-sided die 60,000,000 times. (Part 2 of 2.)

8.5 Using Arrays to Analyze Survey Results; Intro to Exception Handling

- Figure 8.9 uses arrays to summarize data collected in a survey:
  - Twenty students were asked to rate on a scale of 1 to 5 the quality of the food in the student cafeteria, with 1 being “awful” and 5 being “excellent.” Place the 20 responses in an integer array and determine the frequency of each rating.

8.5 Using Arrays to Analyze Survey Results; Intro to Exception Handling

- When a C# program executes, the runtime checks array element indices for validity—all indices must be greater than or equal to 0 and less than the length of the array.
- Any attempt to access an element *outside* that range of indices results in a runtime error that's known as an `IndexOutOfRangeException`.
- At the end of this section, we'll discuss the invalid response value, demonstrate array **bounds checking** and introduce C#'s exception-handling mechanism, which can be used to detect and handle an `IndexOutOfRangeException`.

```
// Fig. 8.9: StudentPoll.cs
// Poll analysis app.
using System;

class StudentPoll
{
   static void Main()
   {
      // student response array (more typically, input at runtime)
      int[] responses = {1, 2, 5, 4, 3, 5, 2, 1, 3, 3, 1, 4, 3, 3, 3,
         2, 3, 3, 2, 14};
      var frequency = new int[6]; // array of frequency counters

```

Fig. 8.9 | Poll analysis app. (Part 1 of 4.)

```
        // for each answer, select responses element and use that value
        // as frequency index to determine element to increment
        for (var answer = 0; answer < responses.Length; ++answer)
        {
           try
           {
              ++frequency[responses[answer]];
           }
           catch (IndexOutOfRangeException ex)
           {
              Console.WriteLine(ex.Message);
              Console.WriteLine(
                 $"   responses[{answer}] = {responses[answer]}\n");
           }
        }
```

Fig. 8.9 | Poll analysis app. (Part 2 of 4.)

```

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

        // output each array element's value
        for (var rating = 1; rating < frequency.Length; ++rating)
        {
           Console.WriteLine($"{rating,6}{frequency[rating],10}");
        }
     }
  }
```

Fig. 8.9 | Poll analysis app. (Part 3 of 4.)

```
Index was outside the bounds of the array.
   responses[19] = 14

Rating Frequency
     1         3
     2         4
     3         8
     4         2
     5         2
```

Fig. 8.9 | Poll analysis app. (Part 4 of 4.)

8.5 Using Arrays to Analyze Survey Results; Intro to Exception Handling

Exception Handling: Processing the Incorrect Response

- An **exception** indicates a problem that occurs while a program executes.
- **Exception handling** enables you to create **fault-tolerant programs** that can resolve (or handle) exceptions.

8.5 Using Arrays to Analyze Survey Results; Intro to Exception Handling

- In many cases, this allows a program to continue executing as if no problems were encountered.
- More severe problems might prevent a program from continuing normal execution, instead requiring the program to notify the user of the problem, then terminate.
- When the runtime or a method detects a problem, such as an invalid array index or an invalid method argument, it **throws** an exception—that is, an exception occurs.

8.5 Using Arrays to Analyze Survey Results; Intro to Exception Handling

The `try` Statement

- To handle an exception, place any code that might throw an exception in a **`try` statement**.
- The **`try` block** contains the code that might throw an exception, and the **`catch` block** contains the code that *handles* the exception if one occurs.

8.5 Using Arrays to Analyze Survey Results; Intro to Exception Handling

Executing the `catch` Block

- Because the runtime performs array bounds checking, it generates an exception—specifically line 20 throws an `IndexOutOfRangeException` to notify the program of this problem.
- At this point the `try` block terminates and the `catch` block begins executing—if you declared any variables in the `try` block, they no longer exist, so they're *not accessible* in the `catch` block.

8.5 Using Arrays to Analyze Survey Results; Intro to Exception Handling

- The `catch` block declares a type (`IndexOutOfRangeException`) and an exception parameter (ex).
- The `catch` block can handle exceptions of the specified type.
- Inside the `catch` block, you can use the parameter's identifier to interact with a caught exception object.

Error-Prevention Tip 8.1

When writing code to access an array element, ensure that the array index remains greater than or equal to 0 and less than the length of the array. This will help prevent `IndexOutOfRangeException`*s in your program.*

8.5 Using Arrays to Analyze Survey Results; Intro to Exception Handling

Message Property of the Exception Parameter

- When lines 22–27 *catch* the exception, the program displays a message indicating the problem that occurred.
- Line 24 uses the exception object’s **`Message`** **property** to get the error message that's stored in the exception object and display it.

8.6 Case Study: Card Shuffling and Dealing Simulation

Class Card and Getter-Only Auto-Implemented Properties

- Class `Card` (Fig. 8.10) represents a playing card that has a face and a suit.
- Prior to C# 6, auto-implemented properties required both a `get` and a `set` accessor.
- C# 6 getter-only auto-implemented properties are read only.

```csharp
// Fig. 8.10: Card.cs
// Card class represents a playing card.
class Card
{
   private string Face { get; } // Card’s face ("Ace", "Deuce", ...)
   private string Suit { get; } // Card’s suit ("Hearts", "Diamonds", ...)

   // two-parameter constructor initializes card's Face and Suit
   public Card(string face, string suit)
   {
      Face = face; // initialize face of card
      Suit = suit; // initialize suit of card
   }

   // return string representation of Card
   public override string ToString() => $"{Face} of {Suit}";
}
```

Fig. 8.10 | Card class represents a playing card.

8.6 Case Study: Card Shuffling and Dealing Simulation

Class Card and Getter-Only Auto-Implemented Properties

- Getter-only auto-implemented properties can be initialized only either in their declarations or in all of the type’s constructors.
- Initializing an auto-implemented property in its declaration is another C# 6 feature known as auto-property initializers.
  - `Type PropertyName { get; set; } = initializer;`

8.6 Case Study: Card Shuffling and Dealing Simulation

Class DeckOfCards

- Class `DeckOfCards` (Fig. 8.11) represents a deck of 52 `Card` objects.

```
// Fig. 8.11: DeckOfCards.cs
// DeckOfCards class represents a deck of playing cards.
using System;

class DeckOfCards
{
   // create one Random object to share among DeckOfCards objects
   private static Random randomNumbers = new Random();

   private const int NumberOfCards = 52; // number of cards in a deck
   private Card[] deck = new Card[NumberOfCards];
   private int currentCard = 0; // index of next Card to be dealt (0-51)

```

Fig. 8.11 | DeckOfCards class represents a deck of playing cards. (Part 1 of 4.)

```
    // 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"};

       // populate deck with Card objects
       for (var count = 0; count < deck.Length; ++count)
       {
          deck[count] = new Card(faces[count % 13], suits[count / 13]);
       }
    }

```

Fig. 8.11 | DeckOfCards class represents a deck of playing cards. (Part 2 of 4.)

```
   // 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 (var first = 0; first < deck.Length; ++first)
      {
         // select a random number between 0 and 51
         var second = randomNumbers.Next(NumberOfCards);

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

```

Fig. 8.11 | DeckOfCards class represents a deck of playing cards. (Part 3 of 4.)

```
   // 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
      }
   }
}
```

Fig. 8.11 | DeckOfCards class represents a deck of playing cards. (Part 4 of 4.)

8.6 Case Study: Card Shuffling and Dealing Simulation

Shuffling and Dealing Cards

- The app of Fig. 8.12 demonstrates the card dealing and shuffling capabilities of class `DeckOfCards`.

```
// Fig. 8.12: DeckOfCardsTest.cs
// Card shuffling and dealing app.
using System;

class DeckOfCardsTest
{
   // execute app
   static void Main()
   {
      var myDeckOfCards = new DeckOfCards();
      myDeckOfCards.Shuffle(); // place Cards in random order

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

         if ((i + 1) % 4 == 0)
         {
            Console.WriteLine();
         }
      }
   }
}
```

Fig. 8.12 | Card-shuffling-and-dealing app. (Part 1 of 2.)

```
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
```

Fig. 8.12 | Card-shuffling-and-dealing app. (Part 2 of 2.)

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.

Specifying an Array Parameter

- For a method to receive an array reference through a method call, the method's parameter list must specify an array parameter.

8.7 Passing Arrays and Array Elements to Methods

Pass-By-Value vs. Pass-By-Reference

- 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.

8.7 Passing Arrays and Array Elements to Methods (cont.)

Passing an Entire Array vs. Passing a Single Array Element

- 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.

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

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

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

      // output original array elements
      foreach (var value in array)
      {
         Console.Write($"   {value}");
      }

```

Fig. 8.13 | Passing arrays and individual array elements to methods. (Part 1 of 4.)

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

        // output modified array elements
        foreach (var value in array)
        {
           Console.Write($"   {value}");
        }

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

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

```

Fig. 8.13 | Passing arrays and individual array elements to methods. (Part 2 of 4.)

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

      // multiply argument by 2
      static void ModifyElement(int element)
      {
         element *= 2;
         Console.WriteLine($"Value of element in ModifyElement: {element}");
      }
   }
```

Fig. 8.13 | Passing arrays and individual array elements to methods. (Part 3 of 4.)

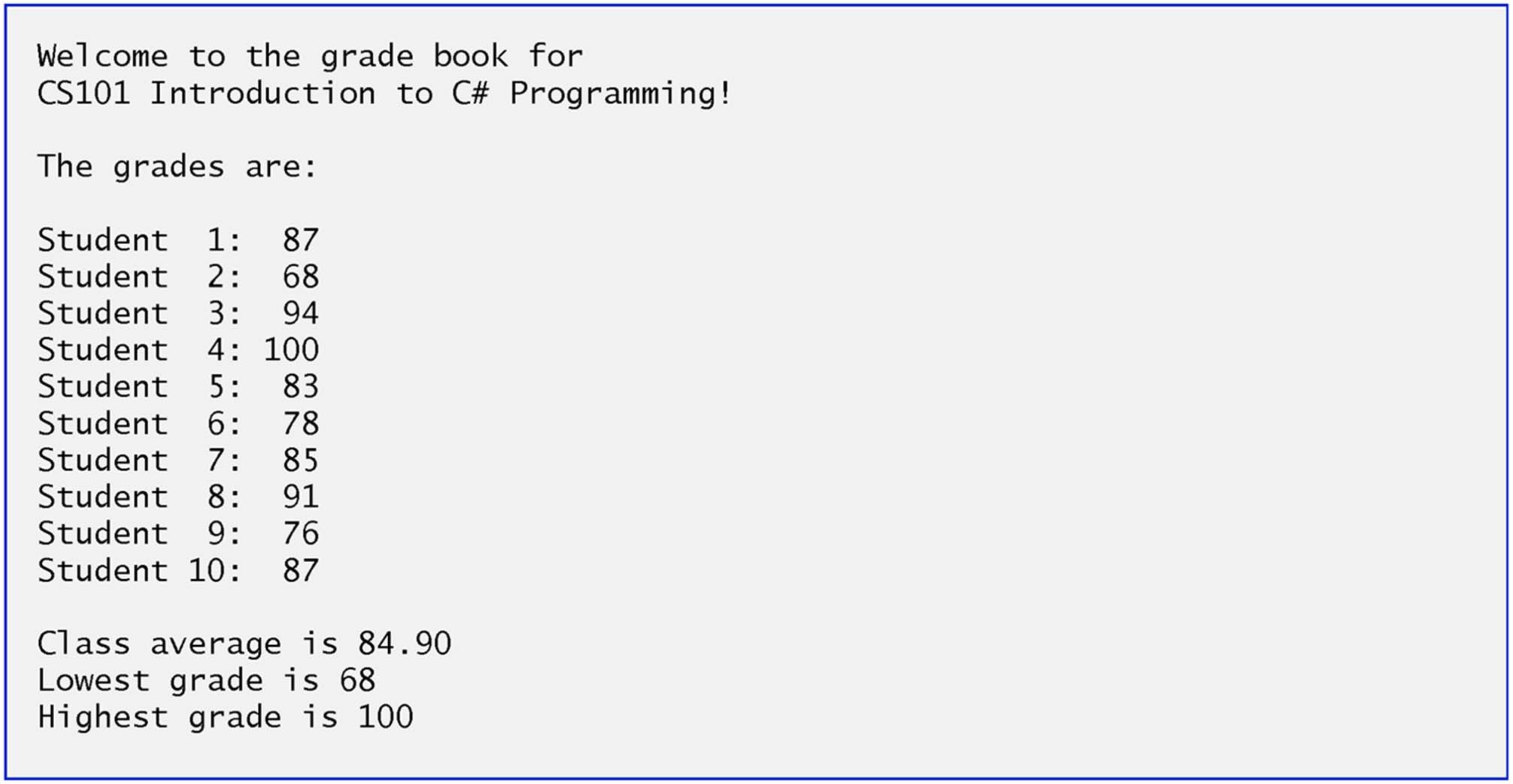
```
Effects of passing reference to entire array:
The values of the original array are:
   1   2   3   4   5

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 4 of 4.)

8.8 Case Study: GradeBook Using an Array to Store Grades

Storing Student Grades in an Array in Class GradeBook

- Fig. 8.14 shows the output of a program that tests class `GradeBook`.
- Class `GradeBook` (Fig. 8.15) uses an array of integers to store the grades of several students on a single exam.



```
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

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

Fig. 8.14 | Output of the `GradeBook` example that stores one exam's grades in an array. (Part 1 of 2.)

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

Fig. 8.14 | Output of the GradeBook example that stores one exam's grades in an array. (Part 2 of 2.)

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

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

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

```

Fig. 8.15 | Grade book using an array to store test grades. (Part 1 of 9.)

```
    // 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
    }

    // 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{CourseName}!\n");
    }

```

Fig. 8.15 | Grade book using an array to store test grades. (Part 2 of 9.)

```
      // 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 {GetAverage():F}");

         // call methods GetMinimum and GetMaximum
         Console.WriteLine($"Lowest grade is {GetMinimum()}");
         Console.WriteLine($"Highest grade is {GetMaximum()}\n");

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

```

Fig. 8.15 | Grade book using an array to store test grades. (Part 3 of 9.)

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

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

      return lowGrade; // return lowest grade
   }

```

Fig. 8.15 | Grade book using an array to store test grades. (Part 4 of 9.)

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

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

      return highGrade; // return highest grade
   }

```

Fig. 8.15 | Grade book using an array to store test grades. (Part 5 of 9.)

```
      // determine average grade for test
      public double GetAverage()
      {
         var total = 0.0; // initialize total as a double

         // sum students' grades
         foreach (var grade in grades)
         {
            total += grade;
         }

         // return average of grades
         return total / grades.Length;
      }

```

Fig. 8.15 | Grade book using an array to store test grades. (Part 6 of 9.)

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

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

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

```

Fig. 8.15 | Grade book using an array to store test grades. (Part 7 of 9.)

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

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

           Console.WriteLine(); // start a new line of output
        }
     }
```

Fig. 8.15 | Grade book using an array to store test grades. (Part 8 of 9.)

```

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

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

Fig. 8.15 | Grade book using an array to store test grades. (Part 9 of 9.)

8.9 Case Study: Class GradeBook Using an Array to Store Grades (cont.)

Class GradeBookTest That Demonstrates Class GradeBook

- The app in Fig. 8.16 demonstrates class GradeBook.

Software Engineering Observation 8.1

A test harness (or test app) creates an object of the class to test and provides it with data, which could be placed directly into an array with an array initializer, come from the user at the keyboard or come from a file (as you'll see in Chapter 17). After initializing an object, the test harness uses the object's members to manipulate the data. Gathering data in the test harness like this allows the class to manipulate data from several sources.

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

      var myGradeBook = new GradeBook(
         "CS101 Introduction to C# Programming", gradesArray);
      myGradeBook.DisplayMessage();
      myGradeBook.ProcessGrades();
   }
}
```

Fig. 8.16 | Create a GradeBook object using an array of grades.

8.9 Multidimensional Arrays

- **Two-dimensional arrays** 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.

8.9 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`.
- An array with *m* rows and *n* columns is called an ***m*-by-*n* array**.

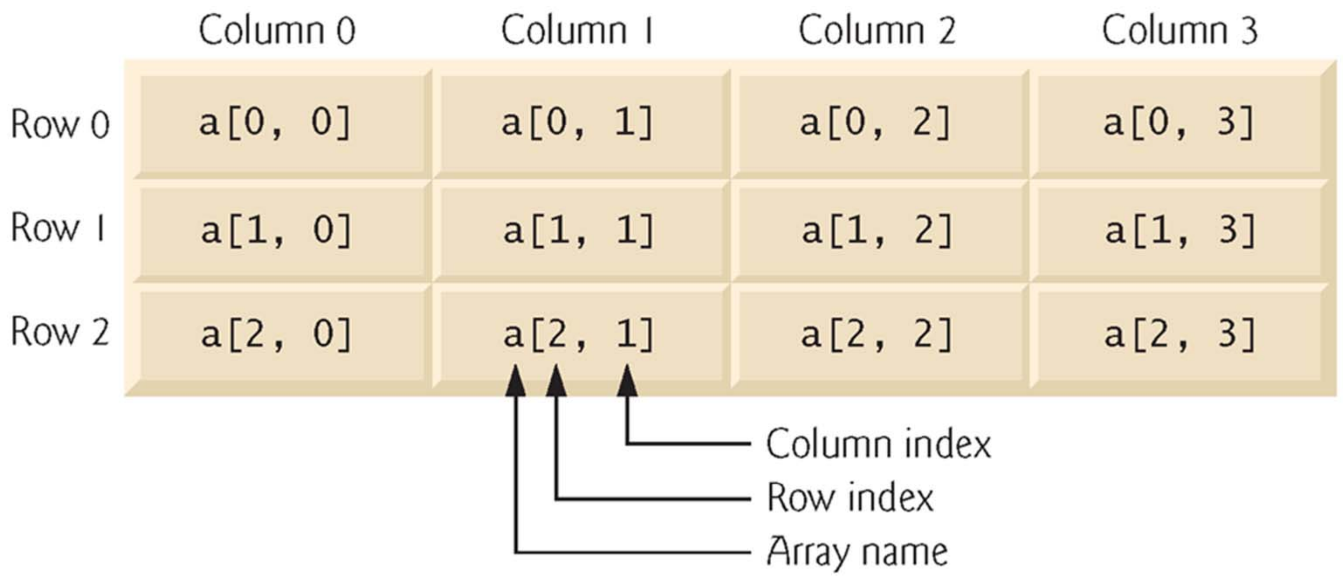


Fig. 8.17 | Rectangular array with three rows and four columns.

8.9 Multidimensional Arrays (Cont.)

Array-Access Expression for a Two-Dimensional Rectangular Array

- Every element in array `a` is identified by an array-access expression of the form a[*row, column*];

Array Initializer for a Two-Dimensional Rectangular Array

- 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 number of columns

8.9 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.

8.9 Multidimensional Arrays (Cont.)

Array Initializer for a Two-Dimensional Jagged Array

- 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}};
```

8.9 Multidimensional Arrays (Cont.)

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

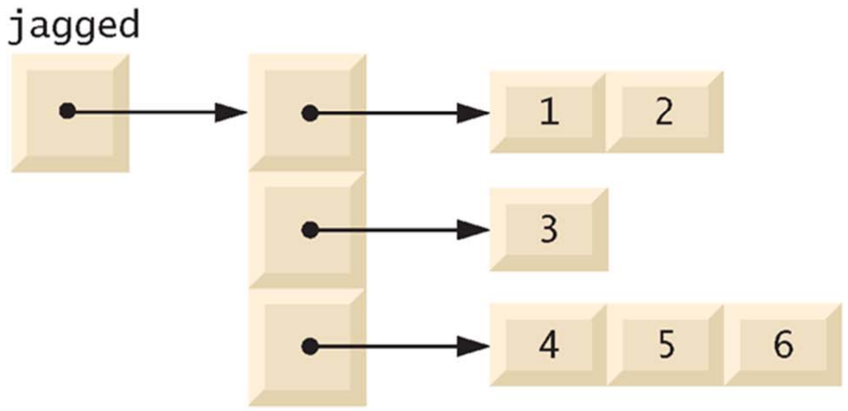


Fig. 8.18 | Jagged array with three rows of different lengths.

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
```

8.10 Multidimensional Arrays (Cont.)

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;

class InitArray
{
   // create and output rectangular and jagged arrays
   static void Main()
   {
      // with rectangular arrays,
      // every row 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 row does not need to be the same length.
      int[][] jagged = {new int[] {1, 2},
                        new int[] {3},
                        new int[] {4, 5, 6}};

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

Fig. 8.19 | Initializing jagged and rectangular arrays. (Part 1 of 4.)

```

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

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

         Console.WriteLine(); // start new line of output
      }
   }

```

Fig. 8.19 | Initializing jagged and rectangular arrays. (Part 2 of 4.)

```
      // output rows and columns of a jagged array
      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($"{element}  ");
            }

            Console.WriteLine(); // start new line of output
         }
      }
   }
```

Fig. 8.19 | Initializing jagged and rectangular arrays. (Part 3 of 4.)

```
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
```

Fig. 8.19 | Initializing jagged and rectangular arrays. (Part 4 of 4.)

8.10 Case Study: GradeBook Using a Rectangular Array

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

- Figure 8.20 shows the output of a program that manipulates a `GradeBook` (Fig. 8.21) that uses a rectangular array grades to store the grades of a number of students on multiple exams.

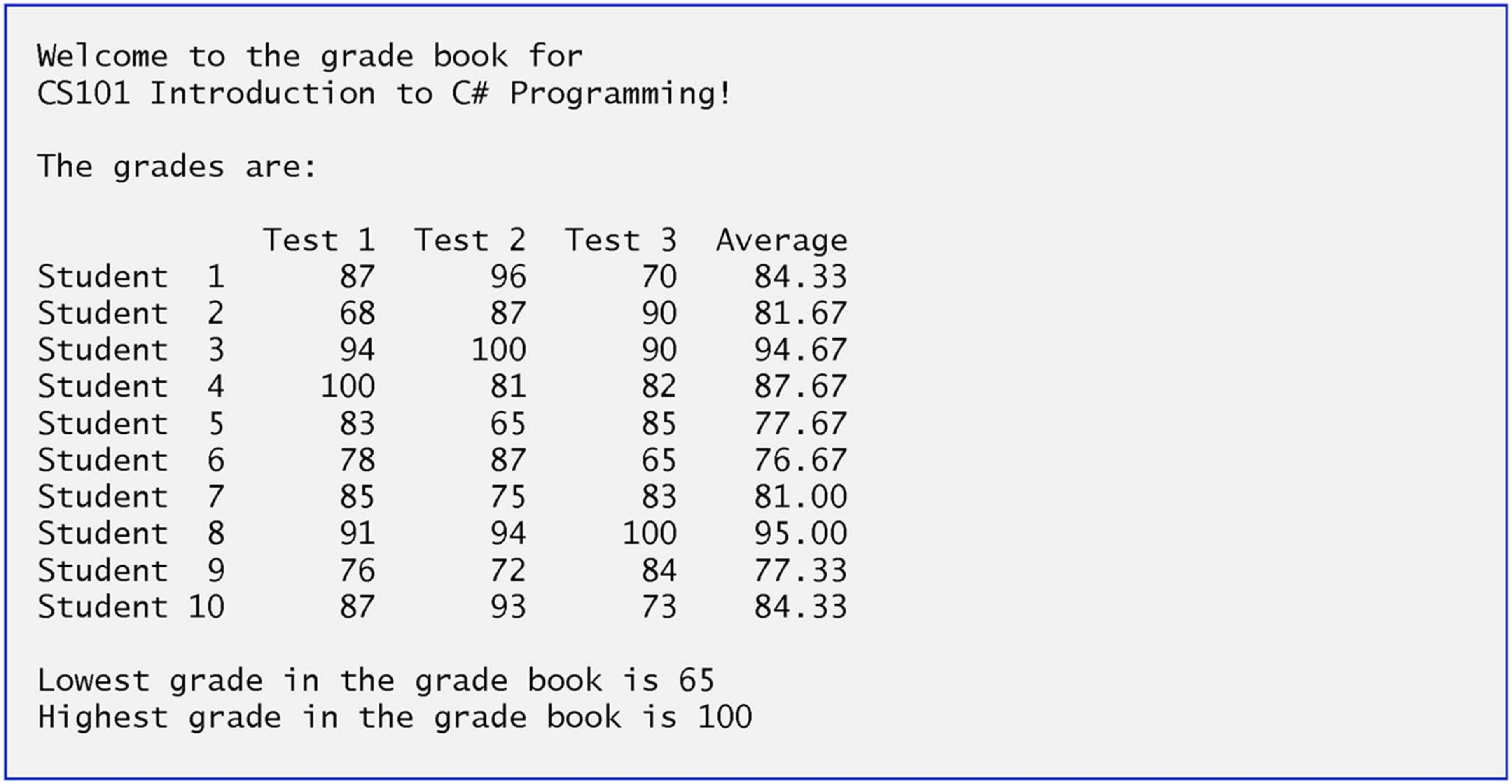

```
Welcome to the grade book for
CS101 Introduction to C# Programming!

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

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

Fig. 8.20 | Output of GradeBook that uses two-dimensional arrays. (Part 1 of 2.)

```
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. 8.20 | Output of `GradeBook` that uses two-dimensional arrays. (Part 2 of 2.)

```
// Fig. 8.21: GradeBook.cs
// Grade book using a rectangular array to store grades.
using System;

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

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

   // 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
   }
```

Fig. 8.21 | Grade book using a rectangular array to store grades. (Part 1 of 9.)

```

   // 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{CourseName}!\n");
   }

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

      // call methods GetMinimum and GetMaximum
      Console.WriteLine(
         $"\nLowest grade in the grade book is {GetMinimum()}" +
         $"\nHighest grade in the grade book is {GetMaximum()}\n");

      // output grade distribution chart of all grades on all tests
      OutputBarChart();
   }
```

Fig. 8.21 | Grade book using a rectangular array to store grades. (Part 2 of 9.)

```

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

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

         return lowGrade; // return lowest grade
      }
```

Fig. 8.21 | Grade book using a rectangular array to store grades. (Part 3 of 9.)

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

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

         return highGrade; // return highest grade
      }
```

Fig. 8.21 | Grade book using a rectangular array to store grades. (Part 4 of 9.)

```
      // determine average grade for particular student
      public double GetAverage(int student)
      {
         // get the number of grades per student
         var gradeCount = grades.GetLength(1);
         var total = 0.0; // initialize total

         // sum grades for one student
         for (var exam = 0; exam < gradeCount; ++exam)
         {
            total += grades[student, exam];
         }

         // return average of grades
         return total / gradeCount;
      }

```

Fig. 8.21 | Grade book using a rectangular array to store grades. (Part 5 of 9.)

```
    // 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
       var frequency = new int[11];

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

```

Fig. 8.21 | Grade book using a rectangular array to store grades. (Part 6 of 9.)

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

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

         Console.WriteLine(); // start a new line of output
      }
   }
```

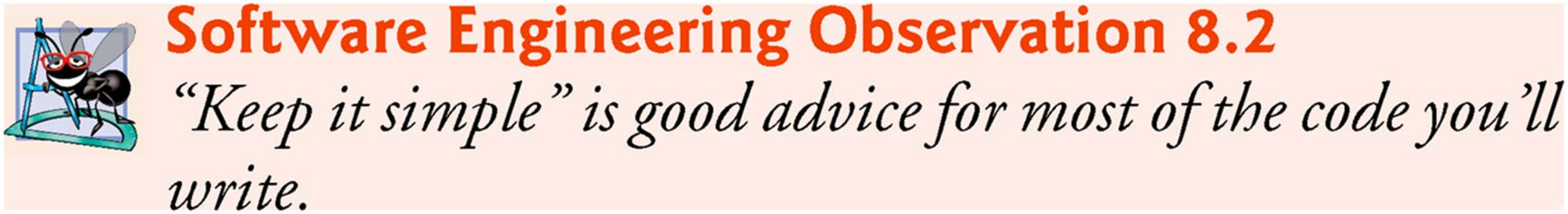
Fig. 8.21 | Grade book using a rectangular array to store grades. (Part 7 of 9.)

```
   // 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 (var test = 0; test < grades.GetLength(1); ++test)
      {
         Console.Write($"Test {test + 1}  ");
      }

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

```

Fig. 8.21 | Grade book using a rectangular array to store grades. (Part 8 of 9.)

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

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

         // call method GetAverage to calculate student's average grade;
         // pass row number as the argument to GetAverage
         Console.WriteLine($"{GetAverage(student),9:F}");
      }
   }
}
```

Fig. 8.21 | Grade book using a rectangular array to store grades. (Part 9 of 9.)



Software Engineering Observation 8.2

“Keep it simple” is good advice for most of the code you’ll write.

8.10 Case Study: GradeBook Using a Rectangular Array(Cont.)

Class `GradeBookTest` That Demonstrates Class `GradeBook`

- The app in Fig. 8.22 demonstrates class `GradeBook`.

```
// Fig. 8.22: GradeBookTest.cs
// Create a GradeBook object using a rectangular array of grades.
class GradeBookTest
{
   // Main method begins app execution
   static void Main()
   {
      // 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}};
```

Fig. 8.22 | Create a GradeBook object using a rectangular array of grades. (Part 1 of 2.)

```

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

Fig. 8.22 | Create a GradeBook object using a rectangular array of grades. (Part 2 of 2.)

8.11 Variable-Length Argument Lists

- 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 parameter list's last parameter.
- Figure 8.23 demonstrates method `Average`, which receives a variable-length sequence of doubles.

Common Programming Error 8.7

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

```
// Fig. 8.23: ParamArrayTest.cs
// Using variable-length argument lists.
using System;

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

      // calculate total using the foreach statement
      foreach (var d in numbers)
      {
         total += d;
      }

      return numbers.Length != 0 ? total / numbers.Length : 0.0;
   }

```

Fig. 8.23 | Using variable-length argument lists. (Part 1 of 3.)

```
      static void Main()
      {
         var d1 = 10.0;
         var d2 = 20.0;
         var d3 = 30.0;
         var d4 = 40.0;

         Console.WriteLine(
            $"d1 = {d1:F1}\nd2 = {d2:F1}\nd3 = {d3:F1}\nd4 = {d4:F1}\n");
         Console.WriteLine($"Average of d1 and d2 is {Average(d1, d2):F1}");
         Console.WriteLine(
            $"Average of d1, d2 and d3 is {Average(d1, d2, d3):F1}");
         Console.WriteLine(
            $"Average of d1, d2, d3 and d4 is {Average(d1, d2, d3, d4):F1}");
      }
   }
```

Fig. 8.23 | Using variable-length argument lists. (Part 2 of 3.)

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

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

Fig. 8.23 | Using variable-length argument lists. (Part 3 of 3.)

8.12 Using Command-Line Arguments

- You can pass command-line arguments to an app 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 app'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 whitespace, not commas.
- Figure 8.24 uses three command-line arguments to initialize an array.

```csharp
// Fig. 8.24: InitArray.cs
// Using command-line arguments to initialize an array.
using System;

class InitArray
{
   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
         var arrayLength = int.Parse(args[0]);
         var array = new int[arrayLength]; // create array

```

Fig. 8.24 | Using command-line arguments to initialize an array. (Part 1 of 4.)

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

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

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

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

Fig. 8.24 | Using command-line arguments to initialize an array. (Part 2 of 4.)

```
C:\Users\PaulDeitel\Documents\examples\ch08\fig08_24>InitArray.exe
Error: Please re-enter the entire command, including
an array size, initial value and increment.
```

```
C:\Users\PaulDeitel\Documents\examples\ch08\fig08_24>InitArray.exe 5 0 4
Index   Value
    0       0
    1       4
    2       8
    3      12
    4      16
```

Fig. 8.24 | Using command-line arguments to initialize an array. (Part 3 of 4.)

```
C:\Users\PaulDeitel\Documents\examples\ch08\fig08_24>InitArray.exe 10 1 2
Index   Value
    0       1
    1       3
    2       5
    3       7
    4       9
    5      11
    6      13
    7      15
    8      17
    9      19
```

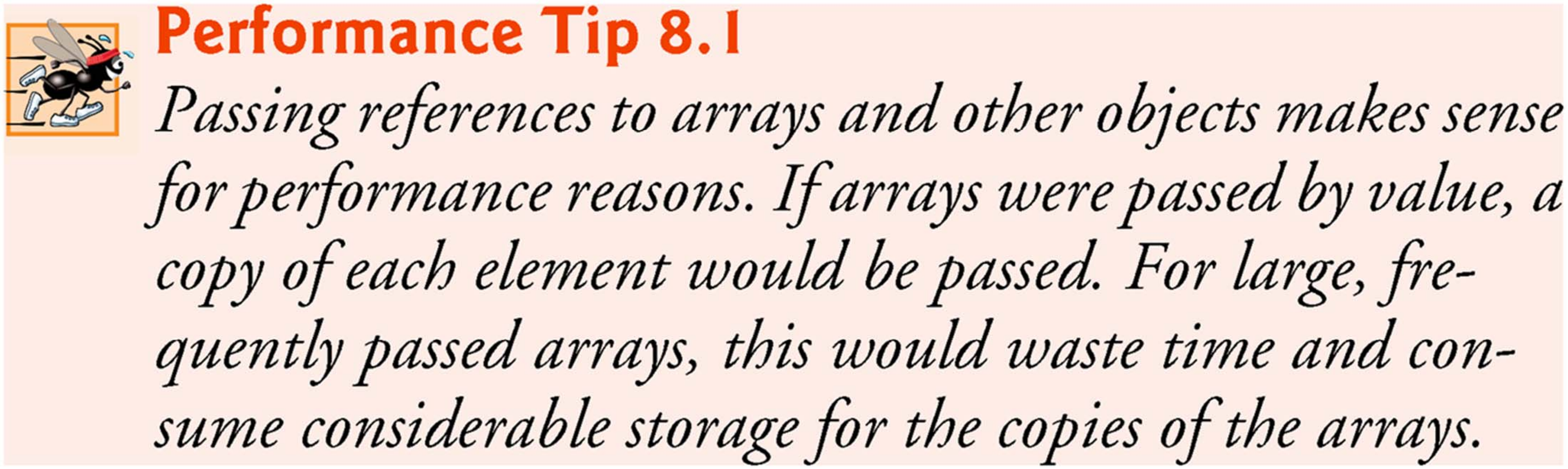
Fig. 8.24 | Using command-line arguments to initialize an array. (Part 4 of 4.)

8.13 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's referenced.

8.13 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 app in Fig. 8.25 demonstrates the subtle difference between passing a reference by value and passing a reference by reference with keyword `ref`.



Performance Tip 8.1

Passing references to arrays and other objects 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 consume considerable storage for the copies of the arrays.

```
// Fig. 8.25: ArrayReferenceTest.cs
// Testing the effects of passing array references
// by value and by reference.
using System;

class ArrayReferenceTest
{
   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");
      Console.Write(
         "Contents of firstArray before calling FirstDouble:\n\t");

```

Fig. 8.25 | Testing the effects of passing an array reference by value and by reference. (Part 1 of 9.)

```
         // display contents of firstArray
         foreach (var element in firstArray)
         {
            Console.Write($"{element} ");
         }

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

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

         // display contents of firstArray
         foreach (var element in firstArray)
         {
            Console.Write($"{element} ");
         }

```

Fig. 8.25 | Testing the effects of passing an array reference by value and by reference. (Part 2 of 9.)

```
         // 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");
         }

```

Fig. 8.25 | Testing the effects of passing an array reference by value and by reference. (Part 3 of 9.)

```
        // 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(
           "Contents of secondArray before calling SecondDouble:\n\t");

        // display contents of secondArray before method call
        foreach (var element in secondArray)
        {
           Console.Write($"{element} ");
        }
```

Fig. 8.25 | Testing the effects of passing an array reference by value and by reference. (Part 4 of 9.)

```csharp

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

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

         // display contents of secondArray after method call
         foreach (var element in secondArray)
         {
            Console.Write($"{element} ");
         }

```

Fig. 8.25 | Testing the effects of passing an array reference by value and by reference. (Part 5 of 9.)

```
         // 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");
         }
      }

```

Fig. 8.25 | Testing the effects of passing an array reference by value and by reference. (Part 6 of 9.)

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

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

```

Fig. 8.25 | Testing the effects of passing an array reference by value and by reference. (Part 7 of 9.)

```
    // modify elements of array and change reference array
    // to refer to a new array
    static void SecondDouble(ref int[] array)
    {
       // double each element's value
       for (var i = 0; i < array.Length; ++i)
       {
          array[i] *= 2;
       }

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

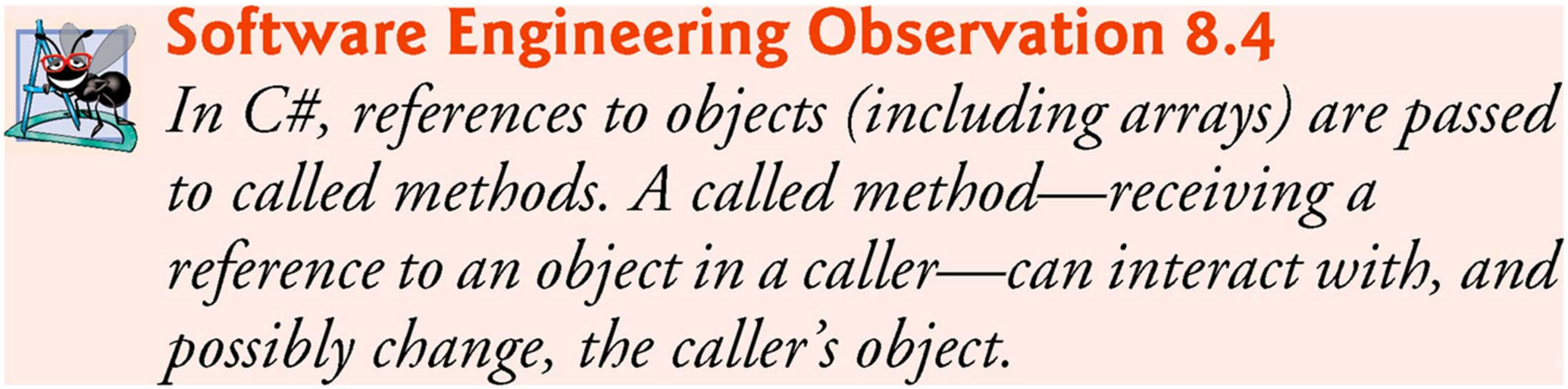
Fig. 8.25 | Testing the effects of passing an array reference by value and by reference. (Part 8 of 9.)

```
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. 8.25 | Testing the effects of passing an array reference by value and by reference. (Part 9 of 9.)

Software Engineering Observation 8.3

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 8.4

In C#, references to objects (including arrays) 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.

Exercises

- 8.6, 8.7, 8.14, and 8.16 (pp. 355-356 of the textbook)