Visual Programming
Array Data Structure

Indah Permatasari, M.Kom.

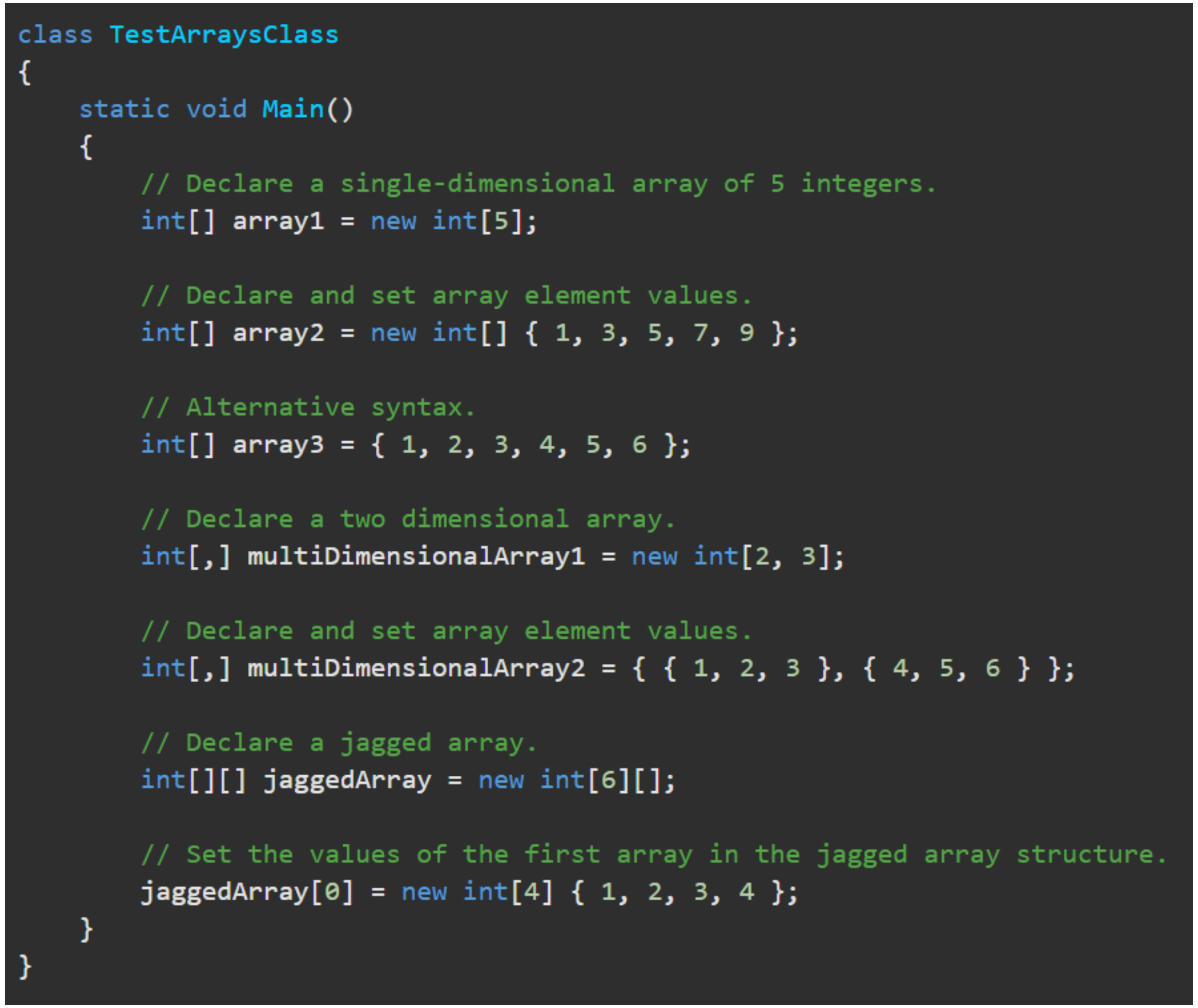
(6th meeting)

Outline

- Array
- Single-Dimensional Array
- Multidimensional Array
- Jagged Array
- Using foreach with array
- Passing Array as Arguments
- Implicitly Typed Array

Introduction

- can **store multiple variables of the same type** in an array data structure.
- declare an array by specifying the type of its elements.
  - `type[] arrayName;`



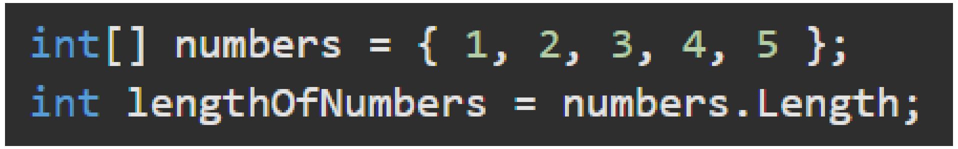
```
class TestArraysClass
{
    static void Main()
    {
        // Declare a single-dimensional array of 5 integers.
        int[] array1 = new int[5];

        // Declare and set array element values.
        int[] array2 = new int[] { 1, 3, 5, 7, 9 };

        // Alternative syntax.
        int[] array3 = { 1, 2, 3, 4, 5, 6 };

        // Declare a two dimensional array.
        int[,] multiDimensionalArray1 = new int[2, 3];

        // Declare and set array element values.
        int[,] multiDimensionalArray2 = { { 1, 2, 3 }, { 4, 5, 6 } };

        // Declare a jagged array.
        int[][] jaggedArray = new int[6][];

        // Set the values of the first array in the jagged array structure.
        jaggedArray[0] = new int[4] { 1, 2, 3, 4 };
    }
}
```

Properties

- An array can be single-dimensional, multidimensional or jagged.
- The number of dimensions and the length of each dimension are established when the array instance is created. These values can't be changed during the lifetime of the instance.
- The default values of numeric array elements are set to zero, and reference elements are set to null.
- A jagged array is an array of arrays, and therefore its elements are reference types and are initialized to null.
- Arrays are zero indexed: an array with n elements is indexed from 0 to n-1.
- Array elements can be of any type, including an array type.
- Array types are reference types derived from the abstract base type Array. Since this type implements IEnumerable and IEnumerable<T>, you can use foreach iteration on all arrays in C#.

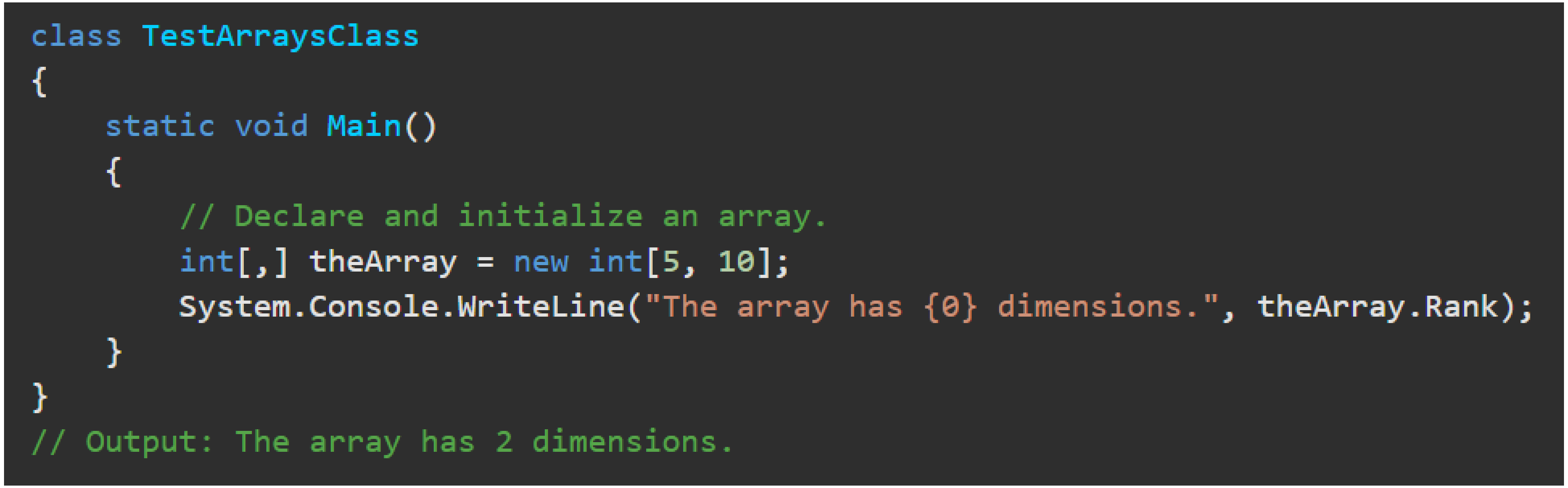
Arrays as Objects

- In C#, arrays are actually objects, and not just addressable regions of contiguous memory as in C and C++. Array is the abstract base type of all array types. You can use the properties and other class members that Array has. An example of this is using the **`Length` property** to **get the length of an array**.
- The following code assigns the length of the `numbers` array, which is `5`, to a variable called `lengthOfNumbers`:



```
int[] numbers = { 1, 2, 3, 4, 5 };
int lengthOfNumbers = numbers.Length;
```

- The Array class provides many other **useful methods and properties for sorting, searching, and copying arrays**. The following example uses the Rank property to display the number of dimensions of an array.



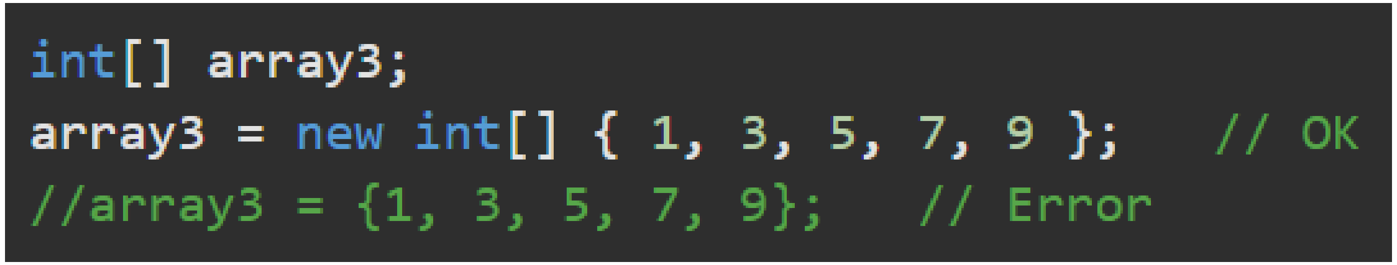
```
class TestArraysClass
{
    static void Main()
    {
        // Declare and initialize an array.
        int[,] theArray = new int[5, 10];
        System.Console.WriteLine("The array has {0} dimensions.", theArray.Rank);
    }
}
// Output: The array has 2 dimensions.
```

Single-Dimensional Array

- create a single-dimensional array using the new operator specifying the array element type and the number of elements.
- example declares an array of five integers: `int[] array = new int[5];`
- Arrays can store any element type you specify, such as the following example that declares an array of strings: `string[] stringArray = new string[6];`

Single-Dimensional Array Initialization

- int[] array1 = new int[] { 1, 3, 5, 7, 9 };
- string[] weekDays = new string[] { "Sun", "Mon", "Tue", "Wed", "Thu", "Fri", "Sat" };



```
int[] array3;
array3 = new int[] { 1, 3, 5, 7, 9 };   // OK
//array3 = {1, 3, 5, 7, 9};   // Error
```

Value Type and Reference Type Arrays

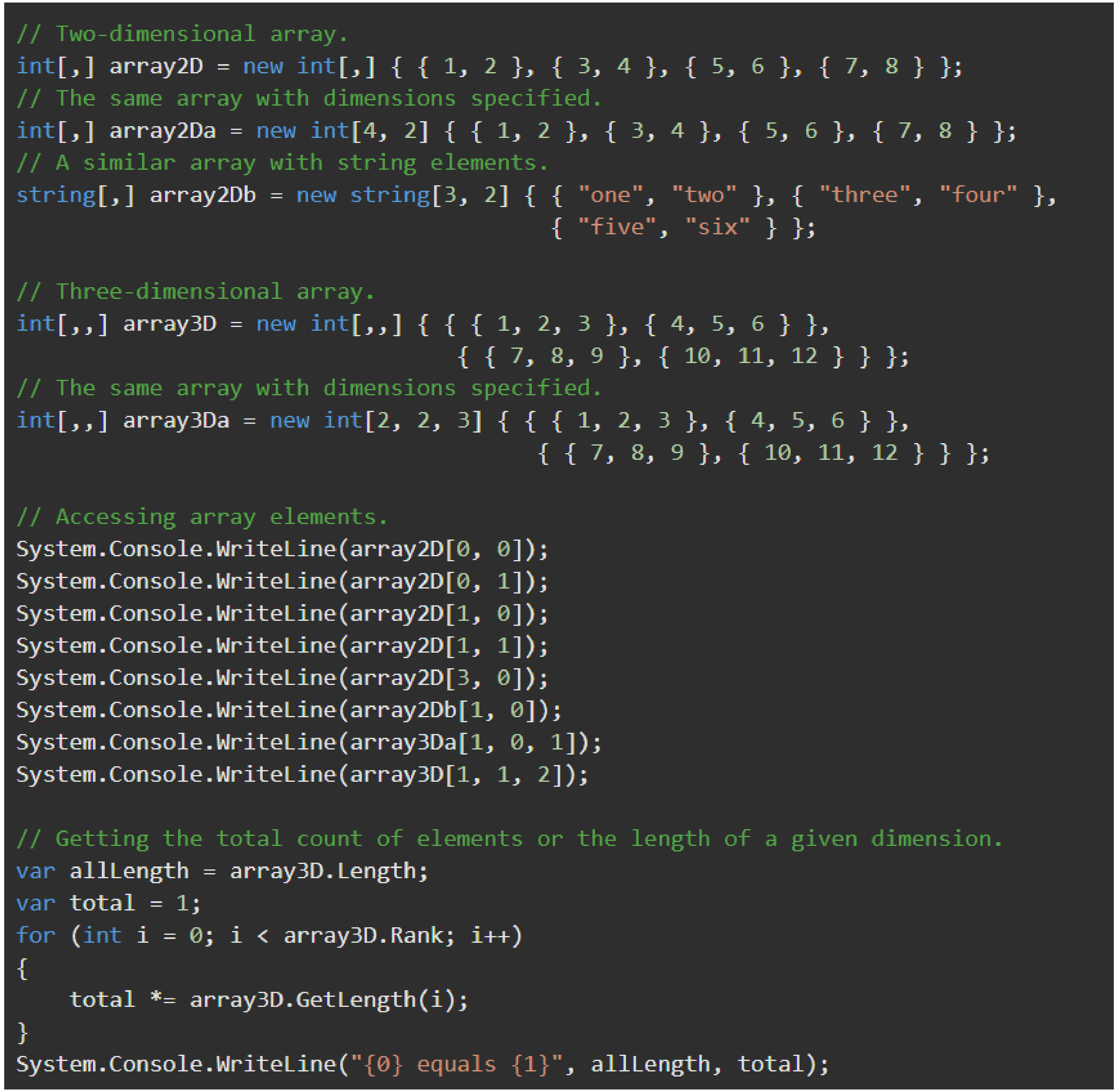
- Consider the following array declaration:
  - SomeType[] array4 = new SomeType[10];
- The result of this statement **depends on** whether **SomeType** is a value type or a reference type.
- If it's a **value type**, the statement creates an array of 10 elements, each of which has the type SomeType.
- If SomeType is a **reference type**, the statement creates an array of 10 elements, each of which is initialized to a **null reference**.

Multidimensional Arrays

- Arrays can have more than one dimension.
- For example, the following declaration creates a two-dimensional array of four rows and two columns.
  - `int[,] array = new int[4, 2];`
- The following declaration creates an array of three dimensions, 4, 2, and 3.
  - `int[,,] array1 = new int[4, 2, 3];`

Multidimensional Arrays Initialization

```
// Output:
// 1
// 2
// 3
// 4
// 7
// three
// 8
// 12
// 12 equals 12
```



```csharp
// Two-dimensional array.
int[,] array2D = new int[,] { { 1, 2 }, { 3, 4 }, { 5, 6 }, { 7, 8 } };
// The same array with dimensions specified.
int[,] array2Da = new int[4, 2] { { 1, 2 }, { 3, 4 }, { 5, 6 }, { 7, 8 } };
// A similar array with string elements.
string[,] array2Db = new string[3, 2] { { "one", "two" }, { "three", "four" },
                                        { "five", "six" } };

// Three-dimensional array.
int[,,] array3D = new int[,,] { { { 1, 2, 3 }, { 4, 5, 6 } },
                                { { 7, 8, 9 }, { 10, 11, 12 } } };
// The same array with dimensions specified.
int[,,] array3Da = new int[2, 2, 3] { { { 1, 2, 3 }, { 4, 5, 6 } },
                                      { { 7, 8, 9 }, { 10, 11, 12 } } };

// Accessing array elements.
System.Console.WriteLine(array2D[0, 0]);
System.Console.WriteLine(array2D[0, 1]);
System.Console.WriteLine(array2D[1, 0]);
System.Console.WriteLine(array2D[1, 1]);
System.Console.WriteLine(array2D[3, 0]);
System.Console.WriteLine(array2Db[1, 0]);
System.Console.WriteLine(array3Da[1, 0, 1]);
System.Console.WriteLine(array3D[1, 1, 2]);

// Getting the total count of elements or the length of a given dimension.
var allLength = array3D.Length;
var total = 1;
for (int i = 0; i < array3D.Rank; i++)
{
    total *= array3D.GetLength(i);
}
System.Console.WriteLine("{0} equals {1}", allLength, total);
```

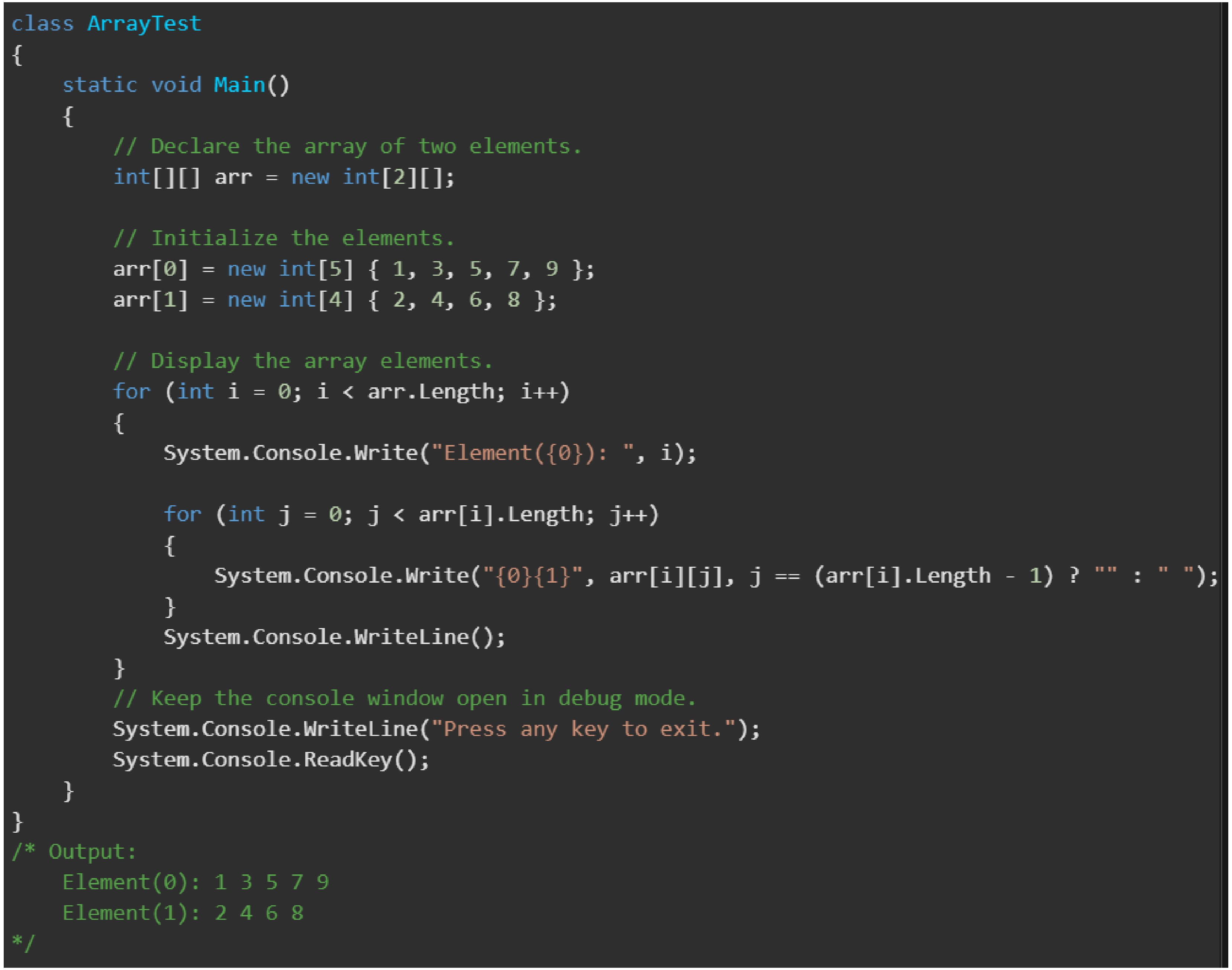
Jagged Arrays

- A jagged array is **an array whose elements are arrays**, possibly of different sizes.
- sometimes called an "array of arrays.
- The following is a declaration of a single-dimensional array that has three elements, each of which is a single-dimensional array of integers:
  - `int[][] jaggedArray = new int[3][];`

  Before:
  - `jaggedArray[0] = new int[5];`
  - `jaggedArray[1] = new int[4];`
  - `jaggedArray[2] = new int[2];`

Jagged Arrays Initialization

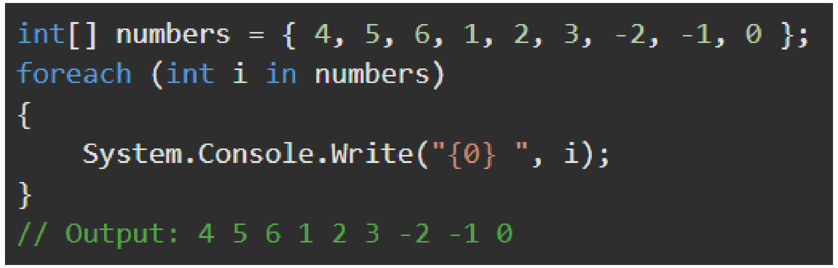
- From above, each of the elements is a single-dimensional array of integers.
- It is also possible to use initializers to fill the array elements with values, in which case you do not need the array size. For example:
  - `jaggedArray[0] = new int[] { 1, 3, 5, 7, 9 };`
  - `jaggedArray[1] = new int[] { 0, 2, 4, 6 };`
  - `jaggedArray[2] = new int[] { 11, 22 };`
- It's possible to mix jagged and multidimensional arrays.
  - `int[][,] jaggedArray4 = new int[3][,];`



```
class ArrayTest
{
    static void Main()
    {
        // Declare the array of two elements.
        int[][] arr = new int[2][];

        // Initialize the elements.
        arr[0] = new int[5] { 1, 3, 5, 7, 9 };
        arr[1] = new int[4] { 2, 4, 6, 8 };

        // Display the array elements.
        for (int i = 0; i < arr.Length; i++)
        {
            System.Console.Write("Element({0}): ", i);

            for (int j = 0; j < arr[i].Length; j++)
            {
                System.Console.Write("{0}{1}", arr[i][j], j == (arr[i].Length - 1) ? "" : " ");
            }
            System.Console.WriteLine();
        }
        // Keep the console window open in debug mode.
        System.Console.WriteLine("Press any key to exit.");
        System.Console.ReadKey();
    }
}
/* Output:
    Element(0): 1 3 5 7 9
    Element(1): 2 4 6 8
*/
```

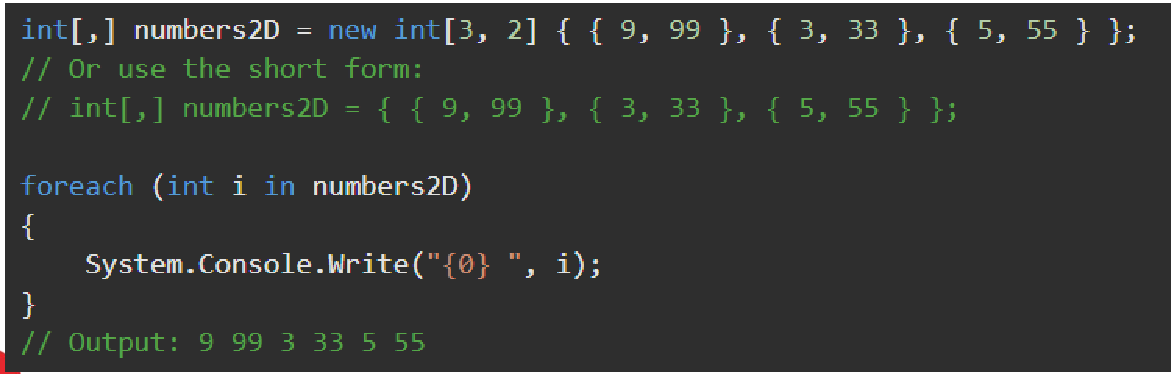
Using foreach with arrays

- The foreach statement provides a simple, clean way to iterate through the elements of an array.
- For single-dimensional arrays, the foreach statement processes elements in increasing index order, starting with index 0 and ending with index Length - 1:



```
int[] numbers = { 4, 5, 6, 1, 2, 3, -2, -1, 0 };
foreach (int i in numbers)
{
    System.Console.Write("{0} ", i);
}
// Output: 4 5 6 1 2 3 -2 -1 0
```

- For multi-dimensional arrays, elements are traversed such that the indices of the rightmost dimension are increased first, then the next left dimension, and so on to the left:



```
int[,] numbers2D = new int[3, 2] { { 9, 99 }, { 3, 33 }, { 5, 55 } };
// Or use the short form:
// int[,] numbers2D = { { 9, 99 }, { 3, 33 }, { 5, 55 } };

foreach (int i in numbers2D)
{
    System.Console.Write("{0} ", i);
}
// Output: 9 99 3 33 5 55
```

Passing arrays as arguments

- Arrays can be passed as arguments to method parameters. Because arrays are reference types, the method can change the value of the elements.

Passing single-dimensional arrays as arguments

- You can pass an initialized single-dimensional array to a method. For example, the following statement sends an array to a print method.

```
int[] theArray = { 1, 3, 5, 7, 9 };
PrintArray(theArray);
```

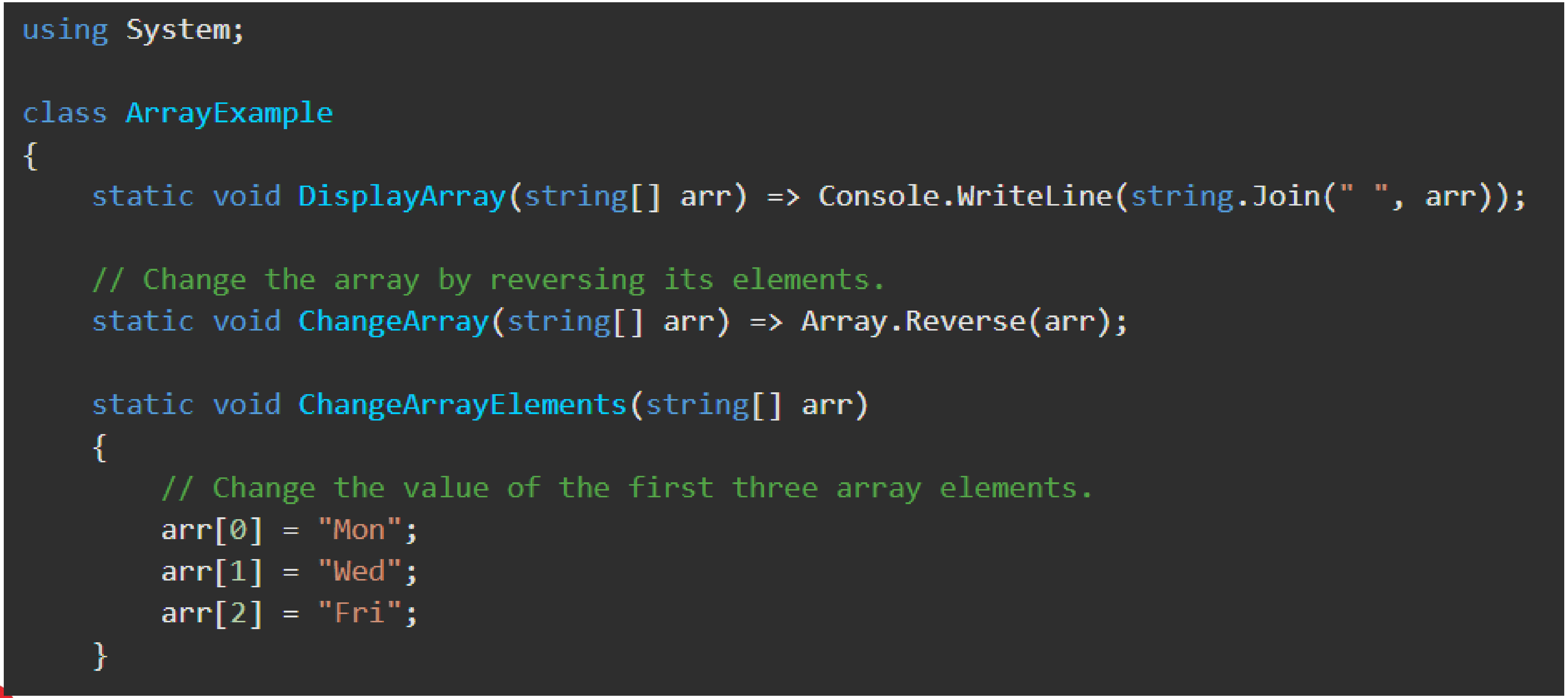
- The following code shows a partial implementation of the print method.

```
void PrintArray(int[] arr)
{
    // Method code.
}
```

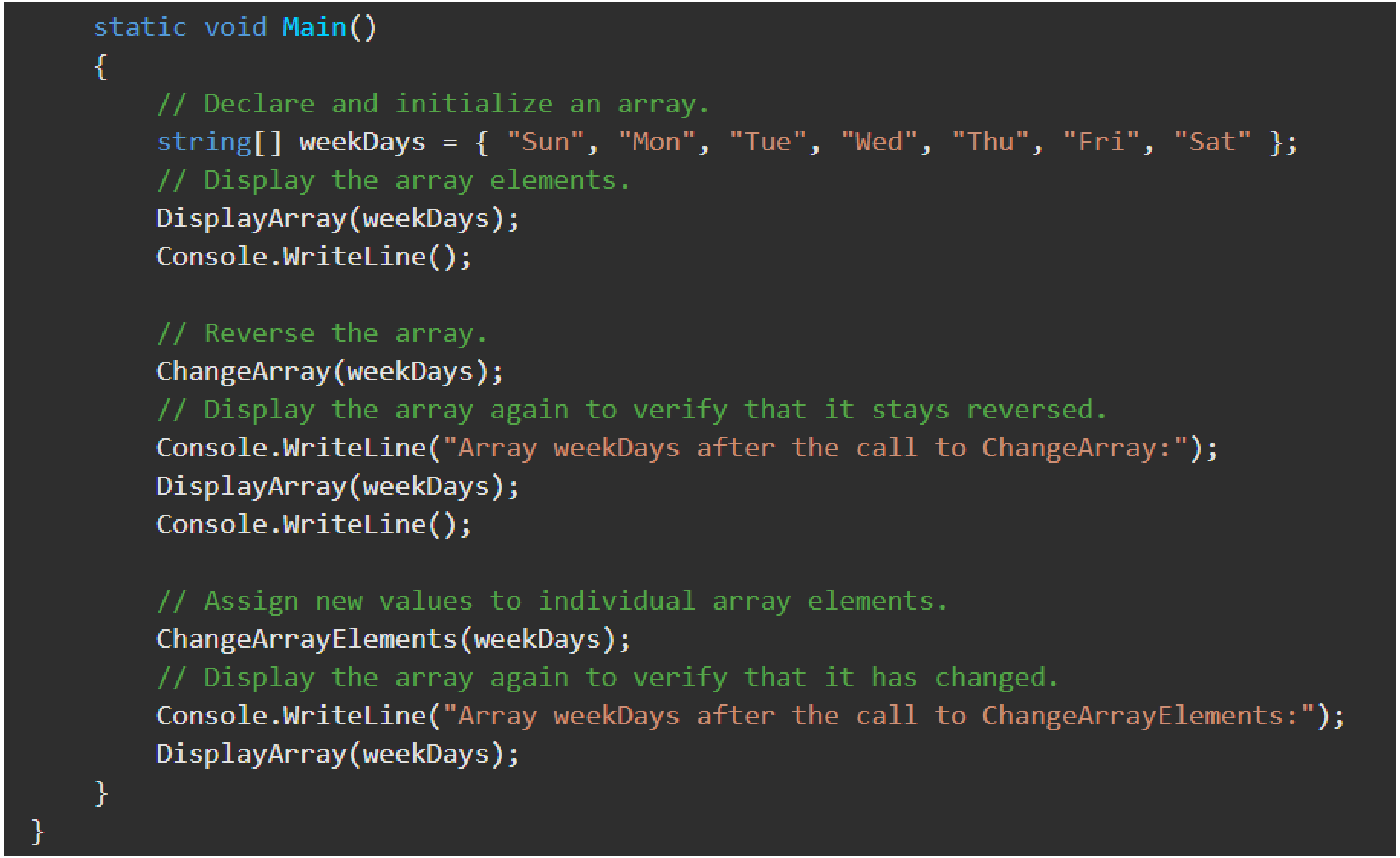
- You can initialize and pass a new array in one step, as is shown in the following example.

```
PrintArray(new int[] { 1, 3, 5, 7, 9 });
```

Example: an array of strings is initialized and passed as an argument to a `DisplayArray` method for strings



```
using System;

class ArrayExample
{
    static void DisplayArray(string[] arr) => Console.WriteLine(string.Join(" ", arr));

    // Change the array by reversing its elements.
    static void ChangeArray(string[] arr) => Array.Reverse(arr);

    static void ChangeArrayElements(string[] arr)
    {
        // Change the value of the first three array elements.
        arr[0] = "Mon";
        arr[1] = "Wed";
        arr[2] = "Fri";
    }
```



```
    static void Main()
    {
        // Declare and initialize an array.
        string[] weekDays = { "Sun", "Mon", "Tue", "Wed", "Thu", "Fri", "Sat" };
        // Display the array elements.
        DisplayArray(weekDays);
        Console.WriteLine();

        // Reverse the array.
        ChangeArray(weekDays);
        // Display the array again to verify that it stays reversed.
        Console.WriteLine("Array weekDays after the call to ChangeArray:");
        DisplayArray(weekDays);
        Console.WriteLine();

        // Assign new values to individual array elements.
        ChangeArrayElements(weekDays);
        // Display the array again to verify that it has changed.
        Console.WriteLine("Array weekDays after the call to ChangeArrayElements:");
        DisplayArray(weekDays);
    }
}
```

Passing multidimensional arrays as arguments

- pass an initialized multidimensional array to a method in the same way that you pass a one-dimensional array.

```
int[,] theArray = { { 1, 2 }, { 2, 3 }, { 3, 4 } };
Print2DArray(theArray);
```

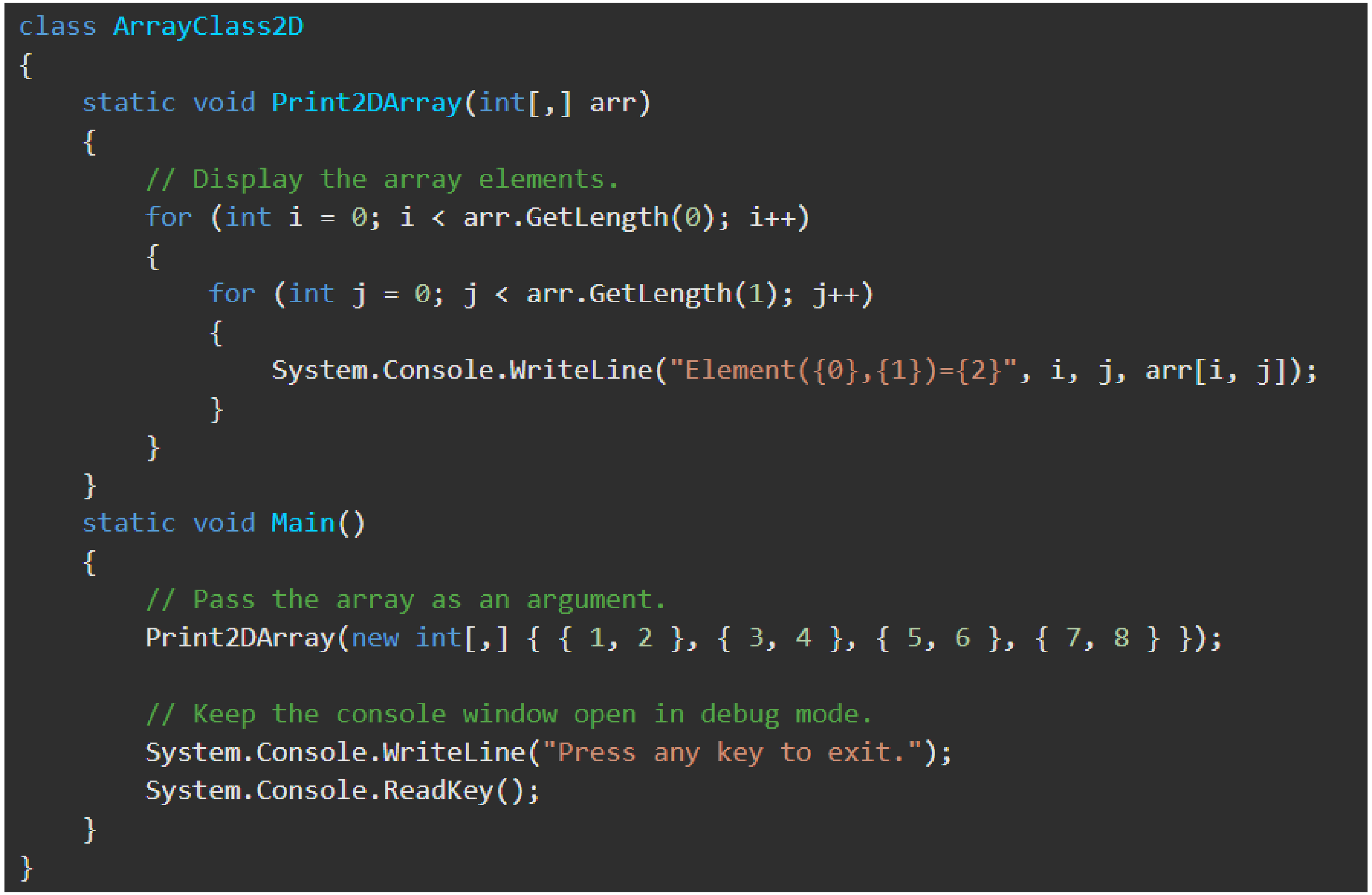
- a partial declaration of a print method that accepts a two-dimensional array as its argument.

```
void Print2DArray(int[,] arr)
{
    // Method code.
}
```

- initialize and pass a new array in one step, as is shown in the following example:

```
Print2DArray(new int[,] { { 1, 2 }, { 3, 4 }, { 5, 6 }, {
7, 8 } });
```

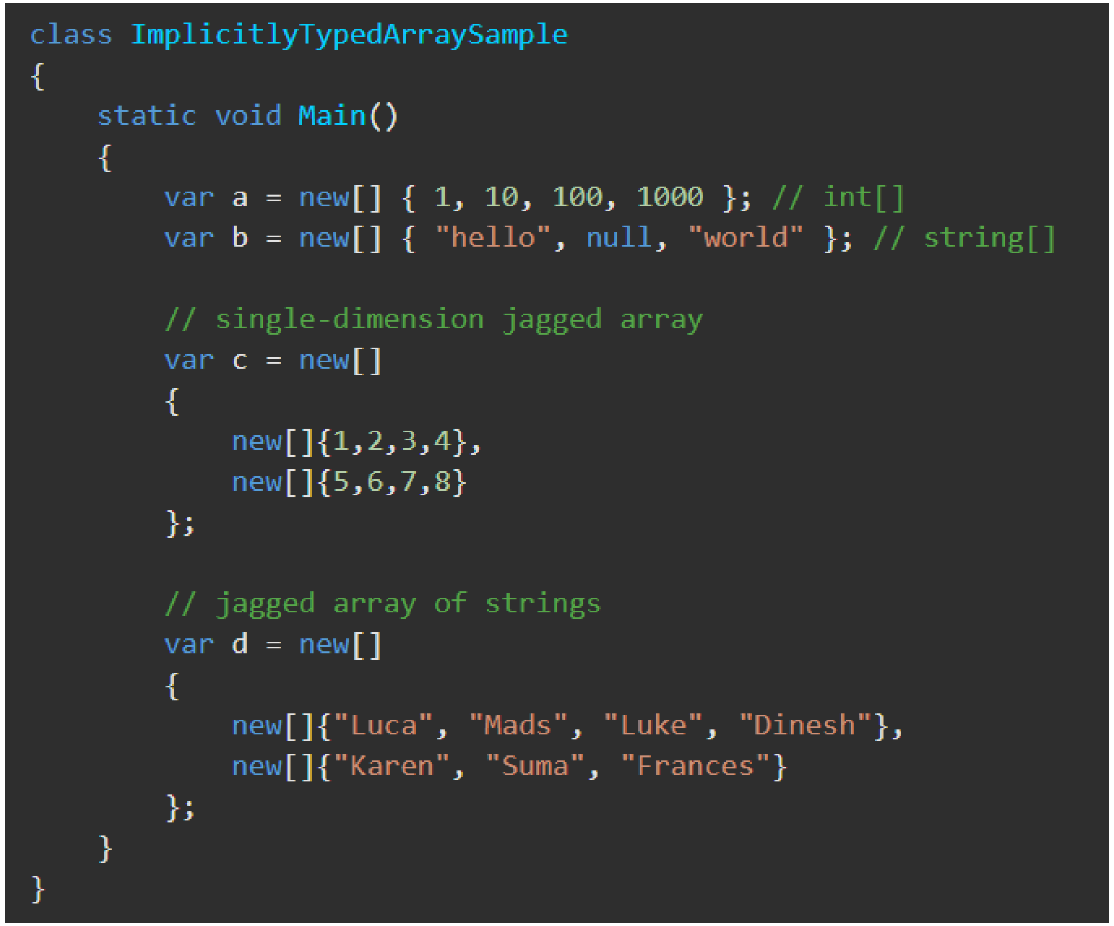
Example



```
class ArrayClass2D
{
    static void Print2DArray(int[,] arr)
    {
        // Display the array elements.
        for (int i = 0; i < arr.GetLength(0); i++)
        {
            for (int j = 0; j < arr.GetLength(1); j++)
            {
                System.Console.WriteLine("Element({0},{1})={2}", i, j, arr[i, j]);
            }
        }
    }
    static void Main()
    {
        // Pass the array as an argument.
        Print2DArray(new int[,] { { 1, 2 }, { 3, 4 }, { 5, 6 }, { 7, 8 } });

        // Keep the console window open in debug mode.
        System.Console.WriteLine("Press any key to exit.");
        System.Console.ReadKey();
    }
}
```

Implicitly Typed Arrays

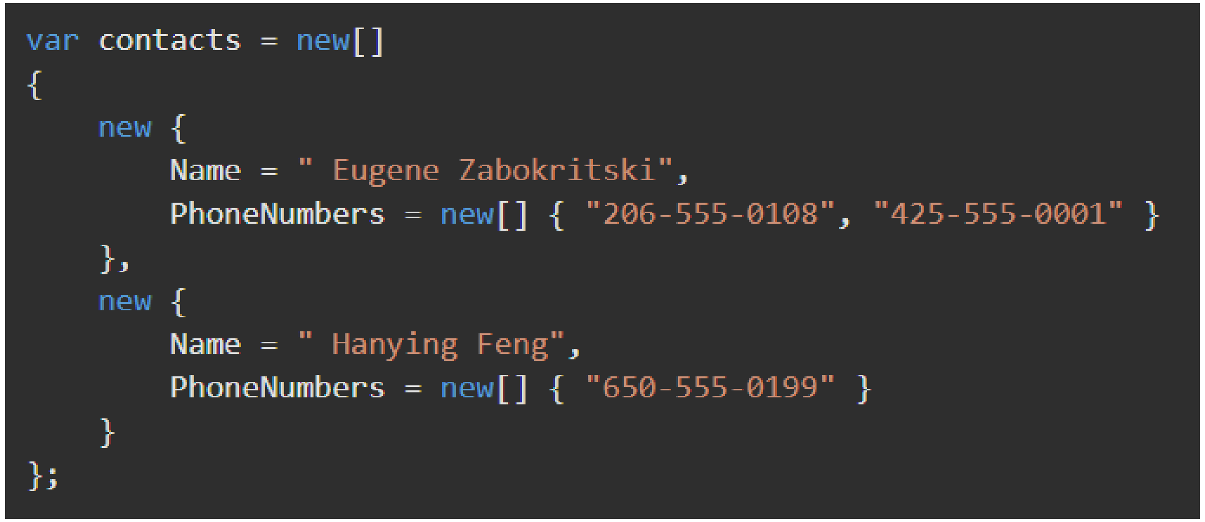
- You can create an implicitly-typed array in which the type of the array instance is inferred from the elements specified in the array initializer.
- The rules for **any implicitly-typed variable** also apply to implicitly-typed arrays.
  - Local variables can be declared without giving an explicit type: `var`
- Implicitly-typed arrays are **usually used** in query expressions together with anonymous types and object and collection initializers.



```
class ImplicitlyTypedArraySample
{
    static void Main()
    {
        var a = new[] { 1, 10, 100, 1000 }; // int[]
        var b = new[] { "hello", null, "world" }; // string[]

        // single-dimension jagged array
        var c = new[]
        {
            new[]{1,2,3,4},
            new[]{5,6,7,8}
        };

        // jagged array of strings
        var d = new[]
        {
            new[]{"Luca", "Mads", "Luke", "Dinesh"},
            new[]{"Karen", "Suma", "Frances"}
        };
    }
}
```

Implicitly-typed Arrays in Object Initializers

- When you create an anonymous type that contains an array, the array **must be** implicitly typed in the type's object initializer.
- Example, `contacts` is an implicitly-typed array of anonymous types, each of which contains an array named `PhoneNumbers`. Note that the `var` keyword is not used inside the object initializers.



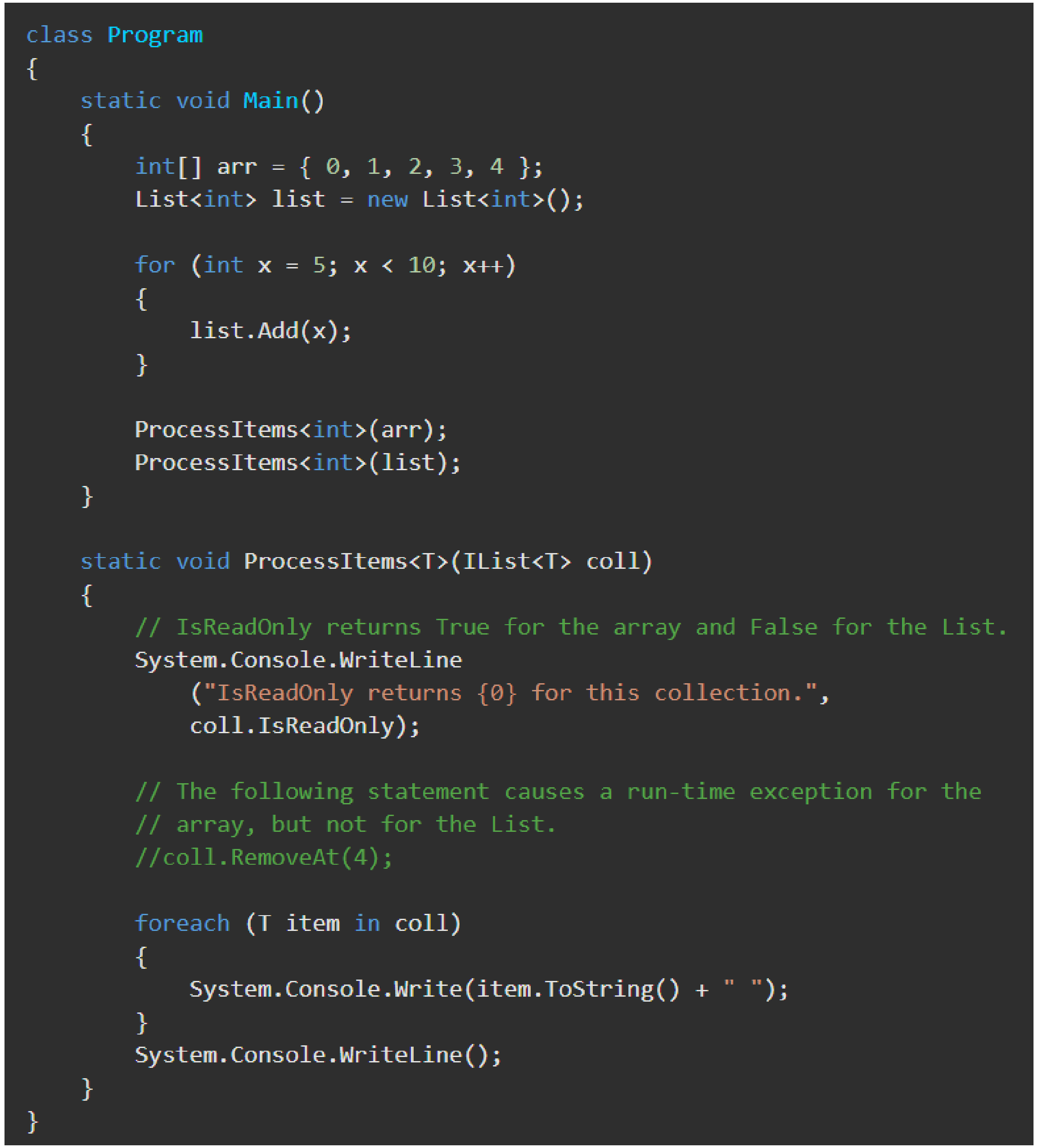
```
var contacts = new[]
{
    new {
        Name = " Eugene Zabokritski",
        PhoneNumbers = new[] { "206-555-0108", "425-555-0001" }
    },
    new {
        Name = " Hanying Feng",
        PhoneNumbers = new[] { "650-555-0199" }
    }
};
```

Bonus: Generic Classes

- Generic classes encapsulate operations that are not specific to a particular data type.
- The most common use for generic classes is with collections like linked lists, hash tables, stacks, queues, trees, and so on.
- Generic classes are invariant.
- In other words, if an input parameter specifies a `List<BaseClass>`, you will get a compile-time error if you try to provide a `List<DerivedClass>`.

Generics and Arrays

- single-dimensional arrays that have a lower bound of zero automatically implement IList<T>
- This enables you to create generic methods that can use the same code to iterate through arrays and other collection types.
- This technique is primarily useful for reading data in collections.



```
class Program
{
    static void Main()
    {
        int[] arr = { 0, 1, 2, 3, 4 };
        List<int> list = new List<int>();

        for (int x = 5; x < 10; x++)
        {
            list.Add(x);
        }

        ProcessItems<int>(arr);
        ProcessItems<int>(list);
    }

    static void ProcessItems<T>(IList<T> coll)
    {
        // IsReadOnly returns True for the array and False for the List.
        System.Console.WriteLine
            ("IsReadOnly returns {0} for this collection.",
            coll.IsReadOnly);

        // The following statement causes a run-time exception for the
        // array, but not for the List.
        //coll.RemoveAt(4);

        foreach (T item in coll)
        {
            System.Console.Write(item.ToString() + " ");
        }
        System.Console.WriteLine();
    }
}
```

Terima kasih.