Contents

Default Values and Resizing

When created, arrays have a fixed size and are populated with some *default values*. We discuss here what those default values are, how an array can be resized, and how we can avoid resizing an array.

Default Values

If we initialize an array but do not assign any values to its elements, each element will get the default value for that element's data type. (These are the same default values that are assigned to instance variables if we do not write a constructor, as we learned in "More Advanced Object Concepts"). In the following example, each element of `myArray` gets initialized to 0, the default value for `int`:

```
int[] myArray = new int[5];
Console.WriteLine(myArray[2]); // Displays "0"
myArray[1]++;
Console.WriteLine(myArray[1]); // Displays "1"
```

However, remember that the default value for any *object* data type is **`null`**, which is an object that does not exist. Attempting to call a method on a **`null`** object will cause a run-time error of the type `System.NullReferenceException`;

```
Rectangle[] shapes = new Rectangle[3];
shapes[0].SetLength(5);  // ERROR
```

Before we can use an array element that should contain an object, we must instantiate an object and assign it to the array element. For our array of `Rectangle` objects, we could either write code like this:

```
Rectangle[] shapes = new Rectangle[3];
shapes[0] = new Rectangle();
shapes[1] = new Rectangle();
shapes[2] = new Rectangle();
```

or use the abridged initialization syntax as follows:

```
Rectangle[] shapes = {new Rectangle(), new Rectangle(),
 ↪  new Rectangle()};
```

Changing the Size

There is a class named `Array` that can be used to resize an array. Upon expanding an array, the additional indices will be filled with the default value of the corresponding type. Shrinking an array will cause the data in the removed indices (those beyond the new length) to be lost.

Example

```
Array.Resize(ref myArray, 4); //myArray[3] now contains 0
myArray[3] = 40;
Array.Resize(ref myArray, 2);
```

In the above example, all data starting at index 2 is lost.

Partially Filled Arrays

To avoid resizing an array, it also possible to declare it *larger than it needs to be*, and then to manipulate an accompanying integer variable that holds the number of elements that are actually stored in the array. The solution to the todo list project illustrates this behavior in detail, the general idea is that you want to let the user store some elements without having to say ahead of time how many, and without having to resize the array constantly. The drawback is that the `Length` property becomes less useful, and that you have to manipulate a custom "accounting" variable to keep track of the actual number of elements manipulated.

```
using System;

public class Program
{
  public static void Main(string[] args)
  {
    // We decide that the maximum number of input is 10.
    const int MAXSIZE = 10;

    int[] inputs = new int[MAXSIZE];

    // The following variable will contain the number of
     ↪  input actually given.
    int numberOfInputs = 0;

    // The following variable will hold the user input.
```

```
    string uInput;

    do
    {
      Console.WriteLine(
        "What is your input #"
          + (numberOfInputs + 1)
          + "? Enter \"done\" when you are done."
      );
      uInput = Console.ReadLine();
      if (uInput != "done")
      {
        inputs[numberOfInputs] = int.Parse(uInput);
        numberOfInputs++; // We increment the number of
↪   items in the list.
      }
      if (numberOfInputs == MAXSIZE)
      {
        Console.WriteLine(
          "You have reached the maximum number of inputs."
        );
      }
    } while (uInput != "done" && numberOfInputs <
     ↪  MAXSIZE);
    /*
     * When the user enters "done", or if the user reached
↪   the maximum number of inputs, we exit this loop.
     */
  }
}
```