Chapter 10: Using Arrays

One-Dimensional Arrays in C#

C# has powerful facilities for handling arrays, which provide a way to store a large number of variables under the same name. Each variable, called an **element**, in an array must have the same data type, and they are distinguished from each other by an **array index**. In C#, the size of the array is static – once you declare an array of a specified size, it stays that size. If you need to add elements to the array, a new array will need to be declared.

Declaring a One-Dimensional Array

Arrays are declared in a manner similar to that used for regular variables. For example, to declare an integer array named **'salary'**, with dimension **5**, we use:

```
// create a new five element integer array and assign the memory address of this array to the name salary
int[] salary = new int[5];
// alternatively: using two lines
int[] salary;
salary = new int[5];
```

The index on an array variable begins at 0 and ends at the dimensioned value minus one. Hence, the **salary** array in the above example has **five** elements, ranging from salary [0] to salary [4]. You use array variables just like any other variable - just remember to include its name and its index.

Initializing an Array

Each element of the array should be initialized with data before it is used…

```
salary[0] = 2;   // the first element of the array has an index number of zero
salary[1] = 3;   // the second element of the array has an index number of one
salary[2] = 6;   // the third element of the array has an index number of two
salary[3] = 4;   // the fourth element of the array has an index number of three
salary[4] = 1;   // the fifth element of the array has an index number of four
```

If an array is not initialized by the programmer, C# will default the values in the array to the following…

Default Array Values:

Data Type	**Default Value**
Numeric (int, double, etc.)	0
Characters and Strings (char & string)	null ('' & "")
Boolean (bool)	false

DateTime	01/01/0001 00:00:00
Objects (reference types)	null

Declare and Initialize an Array in One Step

An array can be declared and initialized in one step for small arrays where the programmer is supplying the data…

```
int[ ] salary = new int[5] {2, 3, 6, 4, 1};
// alternatively: C# allows the programmer to simplify the initialization…
int[ ] salary = {2, 3, 6, 4, 1};   // The number of elements of the array matches the number of initial values used
```

Other Examples:

```
    decimal[ ] salary = {234.45m, 345.67m, 678.83m, 434.46m};
    string[ ] names = {"Cliff", "Melissa", "Peter", "Dave", "Marsha"};
    char[ ] codes = {'s', 'm', 't', 'c'};
```

Using Array Values:

To access the elements in an array, you must use the array name followed by the index number within square brackets…

```
decimal[ ] salary = {234.45m, 345.67m, 678.83m, 434.46m, 543.34};
decimal payrollBefore = salary[0] + salary[1] + salary[2] + salary[3] + salary[4];
// using "normal" variables:  payrollBefore = salary1 + salary2 + salary3 + salary4 + salary5;
```

Processing Arrays within a Loop:

The advantage of using arrays is that the index value can be stored by a whole number variable (like an *int*) and used in a loop – each iteration through the loop will increment the index variable by one.

Each array has a property associated with it called **Length**. The property contains the number of *elements* in the array (which is always one more than the highest valid index number). When looping through an array, it is this property that should be used in the boolean test expression of the loop and not a literal value.

Using a while loop…

```
decimal[ ] salary = {234.45m, 345.67m, 678.83m, 434.46m, 543.34m};
decimal payrollBefore = 0m;
int counter = 0;
while(counter < salary.Length)
{
   payrollBefore = payrollBefore + salary[counter];
   counter++;
}
```

salary.Length → array objects contain a property called Length that contains the number of array elements (in this case Length = 5)

salary[counter] → the first time through the loop, counter = 0, therefore salary[0]
→ the second time through the loop, counter = 1, therefore salary[1]

Using a for loop…

```
decimal[ ] salary = {234.45m, 345.67m, 678.83m, 434.46m, 543.34m};
decimal payrollBefore = 0m;
for(int counter = 0; counter < salary.Length; counter++)
{
   payrollBefore = payrollBefore + salary[counter];
}
```

Note: Variables in C# have block level scope. Any variable declared inside a loop – including the initialization section of the *for* loop – will not be accessible outside of the loop. If you need to access this variable, declare it outside of the loop (method level).

Another *for* loop example…

```
double[] myValues = new double[100];
// add code to store 100 numbers in the myValues array…
double sum = 0, average = 0;
for (int i = 0; i < myValues.Length; i++)
{
  sum += myValues[i];
}
average = sum / myValues.Length;
```

This code finds the average value of 100 numbers stored in the array **myValues**. It first sums each of the values in a **for** loop. That sum is then divided by the number of terms (100) to yield the average. Note the use of the **increment** operator. Also, notice the index variable is declared in the initialization step. This is a common declaration in a loop or block of code. Such **loop** or **block level variables** lose their values once the loop is completed.

Using a foreach loop

a special looping structure for arrays…

A *foreach* loop allows you **read-only** access (you can't change the value of any of the array elements) to each element in an array using the following syntax:

```
foreach (<data type> <variable name> in <array name>)
{
    // use <variable name> for each element
}
```

This loop will cycle through each element in the array. With each iteration of the loop, the variable will hold one element of the array. You don't have to worry about how many elements there are in the array, and you can be sure that you'll get to use each one in the loop. Using this approach, you can modify the code if the preceding examples as follows:

```
foreach(int pay in salary)
{
    payrollBefore = payrollBefore + pay;
}
```

Another *foreach* loop example…

```
double[] myValues = new double[100];
// add code to store 100 numbers in the myValues array…
double sum = 0, average = 0;
foreach(double temp in myValues)
{
    sum += temp;
}
average = sum / myValues.Length;
```

Some useful methods of the Array class…

- **Array.BinarySearch(**array, value**):** Searches a one-dimensional array that's in **ascending order** for an element with a specified value and returns the index for that element. If the value is not found, a negative value is returned (the negative value of the length of the array).
- **Array.Sort(**array**):** Sorts the elements in a one-dimensional array into ascending order.

Code that uses the **Sort** method…

```
string[] lastNames = {"Patrcik", "Vanscoy", "Cappellazzo"};
Array.Sort(lastNames);
string message = "";
foreach (string temp in lastNames)
{
   message += temp + "\n";
}
Console.WriteLine(message);
```

Code that uses the **BinarySearch** method…

```
    string[] employees = {"AbeL", "BenF", "JohnA", "JohnK"};
    decimal[] salesAmounts = {3275.68m, 4298.55m, 5289.57m, 1933.98m};
    int index = Array.BinarySearch(employees, "BenF");
    decimal salesAmount = salesAmounts[index];            // salesAmount = 4298.55
```

Parallel One-Dimensional Arrays

Parallel arrays are simply two or more arrays that have the same number of elements and each element in one array parallels the data in the other array(s).

Example: Design a program that accepts an item number from the user. The program will then search an array of item numbers to determine if the item number exists – and if it does; the index number of that item. If the number exists, use the index number to reference the corresponding elements in 2 other arrays: an array of item prices and an array of item descriptions. If the item number does not exist, display an error.

Item Number	Item Price	Item Description
1	$12.99	Blank Media
2	$23.25	Video
3	$4.75	Storage
4	$2.65	Pen
5	$15.25	Accessory
6	$63.45	Electronics

```
// declare and initialize 'parallel' arrays
int[] itemNumber = { 1, 2, 3, 4, 5, 6 };
double[] itemPrice = { 12.99, 23.25, 4.75, 2.65, 15.25, 63.45 };
string[] itemDescription = { "Blank Media", "Video", "Storage", "Pen", "Accessory", "Electronics" };

// declare method variables
int index = 0;
int userInput;

// get user input
userInput = Convert.ToInt32(txtItemNumber.Text);

// loop through array looking for the item number entered by user
// Note: can't use a foreach loop since the index is required after the loop is finished
while (index < itemNumber.Length)
{
        // if a match is found, exit the loop
        if(userInput == itemNumber[index])
                break;
        index++;
}

// if the index number is valid, a match was found use the index number to access the other 2 arrays
// else display a message that the item number does not exist
if (index < itemNumber.Length)
{
        lblPriceOfItem.Text = itemPrice[index].ToString("C2");
        lblDescriptionOfItem.Text = itemDescription[index];
}
else
{
        Console.WriteLine("Item number " + userInput.ToString() + " is not valid.");
}
```

Multi-Dimensional Arrays in C#

Rectangular Arrays

- Each row has the same number of columns

How to create a rectangular array:

```
type[,] arrayName = new type[numberOfRows, numberOfColumns];
```

Example: A statement that creates a 3x2 array: `int[,] numbers = new int[3,2];`

How to assign values to a rectangular array:

arrayName[rowIndex, columnIndex] = *data;*

The index values for the elements of a 4x4 rectangular array:

0,0	0,1	0,2	0,3
1,0	1,1	1,2	1,3
2,0	2,1	2,2	2,3
3,0	3,1	3,2	3,3

Example: Code that assigns values to a numbers array

```
int[,] numbers = new int[3,2];
numbers[0,0] = 1;
numbers[0,1] = 2;
numbers[1,0] = 3;
numbers[1,1] = 4;
numbers[2,0] = 5;
numbers[2,1] = 6;
```

1	2
3	4
5	6

How to create a rectangular array and assign values with one statement:

Example: Code that creates a 3x2 array and assigns values…

```
int[,] numbers = { {1,2}, {3,4}, {5,6} };
```

Example: Code that creates and assigns values to a 3x2 array of strings…

```
string[,] courses = { {"PROG1195", "JavaScript"},
                      {"PROG1853", "C#"},
                      {"PROG1196", "Computer Architecture"} };
```

GetLength( ) Method – Rectangular Arrays

Syntax: arrayName.GetLength(dimensionIndex)

The GetLength( ) method is used to get the number of rows or columns in a rectangular array. To get the number of rows, specify 0 for the *dimensionIndex* argument. To get the number of columns, specify 1 for this argument. *Note:* The *Length* property stores the total number of elements in the array – which can't be used to control a loop.

Example: Using the numbers array (from above) …

```
int numberOfRows = numbers.GetLength(0);
int numberOfColumns = numbers.GetLength(1);
int sumOfFirstRow = numbers[0,0] + numbers[0,1];
```

Example: Display all the values in the numbers array …

```
string display = "";
for (int rows = 0; rows < numbers.GetLength(0); rows++)
{
        for (int columns = 0; columns < numbers.GetLength(1); columns++)
        {
                display += numbers[rows, columns] + " ";
        }
        display += "\n";
}
Console.WriteLine(display);
```

Example: Display all the values in a string array called courses (from above) using a **foreach** loop…

```
string display = "";
foreach (string temp in courses)
{
        display += temp + "\n";
}
Console.WriteLine(display);
```

Jagged Arrays

- Each row can have a different number of columns
- Sometimes referred to as an array of arrays

How to create a jagged array:

```
type[][] arrayName = new type[numberOfRows][];
arrayName[0] = new type[numberOfColumns];
arrayName[1] = new type[numberOfColumns];
…
```

Example: Create a jagged array with three rows of different lengths

```
int[][] numbers = new int[3][];    // number of rows
numbers[0] = new int[3];           // number of columns for row 1
numbers[1] = new int[4];           // number of columns for row 2
numbers[2] = new int[2];           // number of columns for row 3
```

How to refer to an element of a jagged array: arrayName[rowIndex][columnIndex]

The index values for the elements of a jagged array:

0,0	0,1	0,2	
1,0	1,1	1,2	1,3
2,0	2,1		

Example: Assigns values to a numbers array:

```
numbers[0][0] = 1;
numbers[0][1] = 2;
numbers[0][2] = 3;
numbers[1][0] = 4;
numbers[1][1] = 5;
numbers[1][2] = 6;
numbers[1][3] = 7;
numbers[2][0] = 8;
numbers[2][1] = 9;
```

1	2	3	
4	5	6	7
8	9		

How to create a jagged array and assign values with one statement:

Example: Create and initialize the numbers jagged array…

```
int[][] numbers = { new int[] {1, 2, 3},
                    new int[] {4, 5, 6, 7},
                    new int[] {8, 9} };
```

Example: Create and initialize a jagged array of strings…

```
string[][] names = { new string[3] {"Dave", "Braden", "Rick"},
                     new string[4] {"Dino", "Cliff", "Marsha", "Dave"},
                     new string[2] {"Linda", "Roberto"} };
```

Using Jagged Arrays:

- The *Length* property of the array only stores the number of rows in that array…

  Example: numbers.*Length* stores the value 3 which is the number of rows in the *numbers* array.
- The *Length* property of each row stores the number of columns in that row…

  Example:
  - numbers[0].*Length* stores the value 3 which is the number of columns in the 1st row of the *numbers* array.
  - numbers[1].*Length* stores the value 4 which is the number of columns in the 2nd row of the *numbers* array.

- The *GetLength( )* method works only for the 1st dimension of the array – the rows…if you try to use it on the 2nd dimension – the columns – an *IndexOutOfRangeException* occurs.

  Example:

  - ♦ numbers.*GetLength(0)* returns the value 3 – the number of rows in the *numbers* array.
  - ♦ numbers.*GetLength(1)* crashes the program – the number of columns are different for each row.

  Note: The *GetLength( )* method can work for each row to determine the number of columns in that row. *Example:* numbers[1].*GetLength(0)* returns the value 4 – the number of columns in row 2 of the *numbers* array.

Example: Display the values of the numbers array (from above)...

```
string display = "";
for (int i = 0; i < numbers.Length; i++)
{
        for (int j = 0; j < numbers[i].Length; j++)
        {
                display += numbers[i][j] + " ";
        }
        display += "\n";
}
Console.WriteLine(display);
```

Note: A *foreach* loop can only be used for each row of the array and not the entire array.

This will work…

```
// loop through all columns in the first row
foreach (int temp in numbers[0])
{
        display += temp + " ";
}
```

This will **not** work…(you still need two loops)…

```
// can't loop through entire jagged array
foreach (int temp in numbers)  // Cannot convert type int[] to int
{
        display += temp + " ";
}
```