Arrays

Array is a group of contiguous or related data items that share a common name and derived from the common base class System.Array. The complete set of values is referred to as an array while the individual values are called elements.

Even though the individual elements are primitive value types, the C# array is a reference type. By default, arrays always have a lower bound zero and elements are automatically set to their default values unless indicated otherwise.

One-Dimensional Arrays

A list of items can be given one variable name using only one subscript and such a variable is called a onedimensional array.

Declaration of Arrays:

Syntax: `type[] arrayname;`

Example:

```
int[] counter;
float[] marks;
string[] books
```

Creation of Arrays:

Syntax: `arrayname = new type[size];`

Example:

```
counter=new int[5];
marks=new float[4];
```

Notice that declaration and creation can be combined as shown below:

```
int[] counter=new int[5]; // semi initialized arrays
string[] books = new string[3];
```

Array Initialization

C# provides simple and straightforward ways to initialize arrays at declaration time by enclosing the initial values in curly braces ({}).

E.g. You can assign values to the array at the time of declaration using an array initializer as shown:

```
double[] balance = {340.0, 523.9, 421.0};
```

It is important to note that the context in which an array initializer is used determines the type of the array being initialized. You can also create and initialize an array as shown:

```
int [] marks = new int[5] { 99, 98, 92, 97, 95};
```

You may also omit the size of the array as shown:

```
int [] marks = new int[] { 99, 98, 92, 97, 95};
```

An array of strings may be initialized as shown:

```
string [] names = new string[3] {"Frank", "Elsie", "Robert"};
```

You can also copy an array variable into another target array variable. In such case, both the target and source point to the same memory location:

```
int [] marks = new int[] { 99, 98, 92, 97, 95};
int[] score = marks;  i.e arrays are reference types
```

Note If you do not initialize an array at the time of declaration, the array members are automatically initialized to the default initial value for the array type.

Assigning Values to an Array

You can assign values to individual array elements, by using the index number, like:

```
double[] balance = new double[10];
balance[0] = 4500.0;
```

Example1

The following is a C# program that initializes an array with the following numbers and then displays the same on the screen

10, 20, 30, 40 50

```
using System;
class InitArray   {
     static void Main() {
          int[]  numbers  =  new  int[]  {10,20,30,40,50};
          foreach (int i in numbers)
               Console.WriteLine("Value  is  {0}",  i);
          Console.ReadKey();
     }
}
```

Example2

Write a C# program that initializes an array to even numbers from 2 to 20 and then displays the same on the console window using an appropriate format

```
using System;
class  EvenNumbers
{
     static void Main( )
     {
          int[] numbers   =new int[10];
          for (int i=0; i<numbers.Length; i++)
          {
                numbers[i]=i*2+2;
          }
          for (int i=0; i<numbers.Length; i++)
          {
               Console.WriteLine("Element[{0}] = {1}",
                           i.ToString(), numbers[i].ToString());
          }
     }
}
```

Here we used the for loop to iterate through the array and the Console.WriteLine() method to print each individual element of the array. Note how the indexing operator [] is used.

Multidimensional Arrays

C# allows multidimensional arrays. Multi-dimensional arrays are also called rectangular array. The simplest form of the multidimensional array is the two-dimensional array. A two-dimensional array can be thought of as a table, which has x number of rows and y number of columns. Each dimension of the array is indexed from zero to its maximum size minus one. The first index specifies the row while the second index specifies the column within that row.

Declaration:

```
int[,] myArray;
```

Creation:

```
myArray=new int[3,4];
```

Combination:
```
          int[,] myArray=new int[3,4];
```

Initializing Two-Dimensional Arrays
Multidimensional arrays may be initialized by specifying bracketed values for each row. The following array is with 3 rows and each row has 4 columns.

```
int [ , ] a = new int [3,4]{
          {0, 1, 2, 3},   /* initializers for row indexed by 0 */
          {4, 5, 6, 7},   /* initializers for row indexed by 1 */
          {8, 9, 10, 11} /* initializers for row indexed by 2 */
};
```

Alternatively, an array can be initialize as follows
```
int[,] b ={{0, 1}, {2, 3}, {4, 5}};
```

A string may be initialized as shown below

```
string[,] siblings = new string[2, 2] {{"Frank","Elsie"},
                     {"Mary","Albert"}};
```

Accessing Two-Dimensional Array Elements
An element in two-dimensional array is accessed by using the subscripts. That is, row index and column index of the array. For example the following statement takes 4th element from the 3rd row of the array.
```
int val = a[2,3];
```

Example 1
The following is a C# program that initializes a five by two, two dimensional array and then displays the same numbers on the screen

```
using System;
class InitArray
{
     static void Main(string[] args)
     {
          /* an array with 5 rows and 2 columns*/
          int[,] numbers = new int[5, 2] {{0,0}, {1,2}, {2,4},
                         {3,6}, {4,8}};
          int i, j;
          /* output each array element's value */
          for (i = 0; i<numbers.GetLength(0); i++)
          {
               for (j = 0; j<numbers.GetLength(1); j++)
               {
                    Console.Write("{0}\t",numbers[i,j]);
               }
               Console.WriteLine("");
          }
          Console.ReadKey();
     }
}
```

Example 2
The following is a C# program that initializes a two dimensional array to 10 by 10 multiplication table and then displays the same on the screen using an appropriate format

```
using System;
class MultiplicationTable
{
     static int ROWS = 10;
     static int COLUMNS = 10;
     public static void Main( )
     {
          int[,] product  =new int[ROWS,COLUMNS];
          int i,j;
          for (i=0; i<= product. GetUpperBound(0); i++)
          {
               for (j=0; j<= product. GetUpperBound(1); j++)
               {
                    product[i,j] = (i+1)*(j+1);
                    Console.Write(product[i,j]+"\t");
               }
               Console.WriteLine();
          }
     }
}
```

Notice that it is possible to have other higher dimensions. For example, you can have a three-dimensional rectangular array as follows:

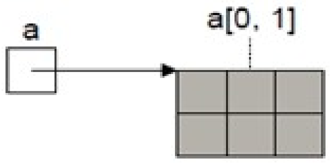
```
int[,,] numbers = new int[4,5,3];
```

Types of Multidimensional Arrays

C# supports two types of multidimensional arrays: rectangular and jagged.

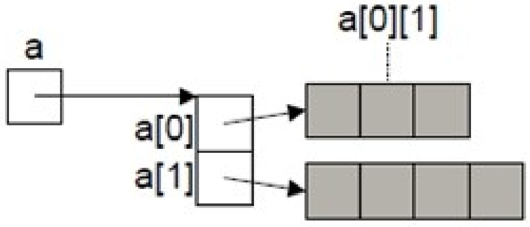
i). A rectangular array is a multidimensional array that has the fixed dimensions' sizes i.e. Rectangular arrays always have a rectangular shape and each row contains an equal number of columns

```
int[,] a = new int[2,3];
```



ii). A jagged arrays (array-of-arrays or variable-size arrays) is a multidimensional array that has the irregular dimensions' sizes i.e. each row does not necessarily contain an equal number of columns

```
int[][] a = new int[2][];
a[0] = new int [3]; //first row has three elements
a[1] = new int [4]; //second row has four elements
```



These statements creates and initializes a two-dimensional array having different length for each row. Specifically, a denotes an array of an array of int, or a single-dimensional array of type int[]. Each of these int[] variables can be initialized individually, and this allows the array to take on a jagged shape.

Jagged arrays can also be created using a looping construct as shown below

```
int[][] jagArray = new int[5][];
for  (int  i=0;  i<jagArray.Length;  i++)
jagArray[i] = new int[i+7];

//Print length of each row
for (int i = 0; i < jagArray.Length; i++)
{
     Console.WriteLine("Length of row {0} is {1}", i,
     jagArray[i].Length);
}
```

You can initialize jagged arrays as shown in the following examples:

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

You can also omit the size of the first array as shown below:

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

or

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

Notice that you can mix rectangular and jagged arrays. For example, the following code declares a singledimensional array of three-dimensional arrays of two-dimensional arrays of type int:

```
          int[][,,][,] numbers;
```

Instantiating and accessing Jagged Arrays

A jagged array with 2 rows where the length of the first row is 3, and the second row is 5 can be instantiated as follows:

```
     int [][] myTable = new int[2][];
     myTable[0] = new int[3]{3 ,-2, 16};
     myTable[1] = new int[5]{1, 9, 5, 6, 98};
```

The foreach loop can be used to access the elements of the array as follows:

```
          foreach(int []row in myTable)
          {
               foreach(int  col  in  row)
               {
                    Console.WriteLine(col);
               }
               Console.WriteLine();
          }
```

In the above code you need to pick up each row (which is an int array) and then iterated through the row while printing each of its columns.

System.Array Class

In C# every array created is automatically derived from the System.Array class. The following are methods/properties present in this class:

Method/Property	Purpose
Clear()	Sets a range of array elements to empty values

CopyTo()	Copies elements from source array to destination array
GetLength()	Gives the number of elements in a given dimension of the array
GetLowerBound()	Gets the lower bound of the specified dimension in the Array.
GetUpperBound()	Gets the upper bound of the specified dimension in the Array.
GetValue()	Gets the value for a given index in the array
Length	Gets the lengths of an array
SetValue()	Sets the value for a given index in the array
IndexOf()	Searches for the specified value and returns the index of the first occurrence within the entire one-dimensional Array.
Reverse()	Reverses the contents of a one-dimensional array Gets
Rank	the rank (number of dimensions) of the Array. Sorts
Sort()	the elements in a one-dimensional array.

Example

```
using System;
class SortArray

     {
     public static void Main( )
     {
          int[] numbers={10,5,2,11,7};

          Console.WriteLine("Before Sort");
          foreach(int i in numbers)
               Console.WriteLine(" " + i);
          Console.WriteLine(" ");

          Array.Sort(numbers);
          Console.WriteLine("After Sort");
          foreach(int i in numbers)
          Console.WriteLine(" " + i);
          Console.WriteLine(" ");
      }
}
```

Collections

Although you can make collections of related objects using arrays, there are some limitations when using arrays for collections.

- The size of an array is always fixed and must be defined at the time of instantiation of an array.
- An array can only contain objects of the same data type, which we need to define at the time of its instantiation.
- Array does not impose any particular mechanism for inserting and retrieving the elements of a collection.

The .Net Framework Class Library (FCL) provides a number of classes to serve as a collection of different types. These classes are present in the System.Collections namespace. Some of the most common classes from this namespace are: ArrayList, Stack, Queue, HashTable and SortedList

Class	Description
ArrayList	Provides a collection similar to an array, but that grows dynamically as the number of elements change.
Stack	A collection that works on the Last In First Out (LIFO) principle, i.e., the last item inserted is the first item removed from the collection.
Queue	A collection that works on the First In First Out (FIFO) principle, i.e., the first item inserted is the first item removed from the collection.
HashTable	Provides a collection of key-value pairs that are organized based on the hash code of the key.
SortedList	Provides a collection of key-value pairs where the items are sorted according to the key. The items are accessible by both the keys and the index.

ArrayList Class

ArrayList is present in System.Collections namespace and used to store a dynamically sized array of objects i.e. has an ability to grow dynamically. ArrayList are created as shown below:

```
ArrayList city=new ArrayList(30);
```

Example

The following is a C# program that populates an ArrayList with the four major country cities and then displays the same on the screen

```
using System;
using   System.Collections;
class ArryList
{
   public static void Main( )
   {
       ArrayList  cities=new  ArrayList();
       cities.Add("Nairobi");
       cities.Add("Nakuru");
       cities.Add("Mombasa");
       cities.Add("Kisumu");
       Console.WriteLine("Capacity="  +  cities.Capacity);
       foreach(string city in cities)
            Console.WriteLine(" " + city);
   }
}
```

The ArrayList class can also implement the indexer property (or index operator) which allow its elements to be accessed using the [] operators, similar to arrays. The following program is similar to the above code but uses the indexers to access the elements of the ArrayList.

```
using System;
using   System.Collections;
class ArryList
{
      public static void Main( )
      {
           ArrayList  cities=new  ArrayList();
           cities.Add("Nairobi");
           cities.Add("Nakuru");
```

```
                cities.Add("Mombasa");
                cities.Add("Kisumu");
                Console.WriteLine("Capacity="  +  cities.Capacity);
                for(int i=0;i< cities.Count;i++)
                    Console.WriteLine(" " + cities[i]);
            }
        }
```

ArrayList Property & Methods
The following is a table of a list of some important properties and methods of the ArrayList:

Property or Method	Description
Capacity	Gets or sets the number of elements the ArrayList can contain. Gets
Count	the exact number of elements in the ArrayList.
Add(object)	Adds an element at the end of an ArrayList.
Remove(object)	Removes an element from the ArrayList.
RemoveAt(int)	Removes an element at the specified index from the ArrayList.
Insert(int, object)	Inserts an object in the ArrayList at the specified index.
Clear()	Removes all the elements from the ArrayList
Contains(object)	Returns a boolean value indicating whether the ArrayList contains the supplied element or not.
CopyTo()	Copies the elements of the ArrayList to the array supplied as a parameter. This method is overloaded and one can specify the range to be copied and also from which index of the array copy should start.
IndexOf(object)	Returns the zero based index of the first occurrence of the object in ArrayList. If the object is not found in the ArrayList, it returns -1.
LastIndexOf(object)	Returns the zero based index of the last occurrence of the object in the ArrayList.

String
The string type represents a string of Unicode characters. The string keyword is an alias for the System.String class. In C#, you can use strings as array of characters, however, more common practice is to use the string keyword to declare a string variable. The objects of the String class (or string) are immutable by nature i.e. the value of a string cannot be modified once established. Thus modifying a string will in fact return a new object containing the modification
Even though string is a reference type, the equality operators "==" and "!=" are defined to compare the value with the string objects

Creating a String Object
You can create string object using one of the following methods:
- By assigning a string literal to a String variable
- By using a String class constructor
- By using the string concatenation operator (+)
- By calling a formatting method to convert a value or an object to its string representation

Parse Method
.NET data types provide the ability to parse a string to corresponding value
```
    Bool myBool = bool.Parse ("True ");
    int myInt = int.Parse ("8 ");
    char myChar = char.Parse ("w ");
```

string Operators
- The + operator concatenates strings:

```
     string a = "good " + "morning";
```

- The [] operator accesses individual characters of a string:

```
     char x = "test"[2]; // x = 's';
```

The string class and its members

The following is description of some common properties and methods of the String class

Property or Method	Description
Length	Gets the number of characters the String object contains.
Compare(string s1, string s2)	Compares two specified string objects and returns an integer that indicates their relative position in the sort order.
Equals(string s)	Returns true if the supplied string is exactly the same as this string, else returns false.
Concat(string s)	Returns a new string that is the concatenation (addition) of this and the supplied string s.
Insert(int index, string s)	Returns a new string by inserting the supplied string s at the specified index of this string.
Copy(string s)	This static method returns a new string that is the copy of the supplied
StartsWith(string s)	Returns true if this string starts with the supplied string s.
EndsWith(string s)	Returns true if this string ends with the supplied string s.
Format(string format, Object arg0)	Replaces one or more format items in a specified string with the string representation of a specified object.
IndexOf(string s) IndexOf(char ch)	Returns the zero based index of the first occurrence of the supplied string s or supplied character ch. This method is overloaded and more versions are available.

Replace(char, char)
Replace(string, string)
	Returns a new string by replacing all occurrences of the first char with the second char (or first string with the second string).

Split(params char[])	Identifies those substrings (in this instance) which are delimited by one or more characters specified in an array, then places the substrings into a String array and returns it.

Substring(int i1)
Substring(int i2, int i3)
	Retrieves a substring of this string starting from the index position i1 till the end. In the second overloaded form, it retrieves the substring starting from the index i2 and which has a length of i3 characters.

ToCharArray()	Returns an array of characters of this string.
ToUpper()	Returns a copy of this string in uppercase.
ToLower()	Returns a copy of this string in lowercase.
Trim()	Returns a new string by removing all occurrences of a set of specified characters from the beginning and at end of this instance.

Examples

```
using System;
class DemoString
{
     public static void Main()
     {
          string s1="Lean";
          string    s2=s1.Insert(3,"r");
          string   s3=s2.Insert(5,"er");
          string s4="Learner";
          string s5=s4.Substring(4);
          Console.WriteLine(s2);
          Console.WriteLine(s3);
```

```
            if(s3.Equals(s4))
            Console.WriteLine("Two Strings are Equal");
            Console.WriteLine("Substring="+s5);
        }
}
```

Example 2

```
using System;
class TestString
{
     static void Main()
     {
          string[] sarray = new string[]{ "Hello", "From", "Author" };
          string message = String.Join("  ", sarray);
          Console.WriteLine("Message:             {0}",
          message); //formatting method to convert a
          value
          string chat = String.Format("Message sent at {0:t} on {0:D}",
          DateTime.Now);
          Console.WriteLine("Message: {0}", chat);
     }
}
```

Verbatim strings

The @-prefixed string is called verbatim string, which is used to disable the processing of escaped characters in the string

For example:

```
Console.WrteLine (@"c:\My Documents\My Videos ");
Console.WriteLine (@"This is a ""value-type"" variable!");
```

The StringBuilder class

The System.Text.StringBuilder class is very similar to the System.String class with the difference that it is mutable i.e. the internal state of its objects can be modified by its operations. Unlike in the string class, you must first call the constructor of a StringBuilder to instantiate its object.

```
string s = "This is held by string";
StringBuilder sb = new StringBuilder("This is held by StringBuilder");
```

StringBuilder is similar to ArrayList and it grows automatically as the size of the string it contains changes. Hence, the Capacity of a StringBuilder may be different from its Length. Some of the more common properties and methods of the StringBuilder class are listed in the following table:

Property or Method	Description
Append()	Appends the string representation of the specified object at the end of this StringBuilder instance. The method has a number of overloaded forms.
Insert()	Inserts the string representation of the specified object at the specified index of this StringBuilder object.

Replace(char, char)
Replace(string, string)
	Replaces all occurrences of the first supplied character (or string) with the second supplied character (or string) in this StringBuilder object.

Remove(int st, int length)	Removes all characters from the index position st of specified length in the current StringBuilder object.
Equals(StringBuilder)	Checks the supplied StringBuilder object with this instance and returns true if both are identical; otherwise, it returns false.

Example

```
using System;
using System.Text;
class  TestStringBuilder
{
   static void Main()
   {
   StringBuilder  sb  =  new  StringBuilder("The  text");
   string s = " is complete";
   Console.WriteLine("StringBuilder before appending is '{0}'", sb);
   Console.WriteLine("StringBuilder after appending '{0}' is '{1}'", s,
                      sb.Append(s));
   Console.WriteLine("StringBuilder after inserting 'now' is '{0}'",
                      sb.Insert(11,"Now"));
   Console.WriteLine("StringBuilder after removing 'is' is '{0}'",
                      sb.Remove(8, 3));
   Console.WriteLine("StringBuilder after replacing 'e' with 'x' {0}",
                      sb.Replace('e', 'x'));
   }
}
```