ARRAY, LINKED LIST AND HASHSET DATA STRUCTURE

What is Array?

An **array** is an ***indexed sequence*** of components. Typically, the array occupies **sequential** storage locations. The **length** of the array is determined when the array is created, and *cannot be changed*. Each component of the array has a *fixed, unique* ***index***.

Arrays works as ***collections*** of items. You can use them to gather items in a single group, and perform various operations on them, e.g. **sorting**. Besides that, several methods within the framework work on arrays, to make it possible to accept a range of items instead of just one.

Array Index

Array **indices** are *integers*. The bracket notation **a[i]** is used (and not overloaded). Bracket operator performs *bounds checking*. An *array of length* n has **bounds 0** and **n-1**. Any component of the array can be inspected or updated by using its index

```
int[] a = new int[10];

int[] b = a;

int[] c = new int[3] {1,2,3};
```

Arrays are **homogeneous**. However, an array of an object type may contain objects of any subtype of that object. For example, an array of Animal may contain objects of type Cat and objects of type Dog. An array of Object may contain any type of object (but cannot contain primitives)

Declaring Array

Arrays are declared much like variables, with a set of [] brackets after the datatype, like this:

```
string[] names;
```

You need to instantiate the array to use it, which is done like this:

```
string[] names = new string[2];
```

The number (2) is the size of the array, that is, the number of items we can put in it. Putting items into the array is pretty simple as well:

names[0] = "John Doe";

Example Program 1

```
using System;
using System.Collections;

namespace ConsoleApplication1
{
  class Program
  {
  static void Main(string[] args)
  {
    string[] names = new string[2];

    names[0] = "John Doe";
    names[1] = "Jane Doe";

    foreach(string s in names)
    Console.WriteLine(s);

    Console.ReadLine();
  }
  }
}
```

for(int i = 0; i < names.Length; i++)

We use the ***foreach loop***, because it's the easiest, but of course we could have used one of the other types of loop instead. The for loop is good with arrays as well, for instance if you need to count each item, like this:

It's actually very simple. We use the **Length** property of the array to decide how many times the loop should iterate, and then we use the **counter (i)** to output where we are in the process, as well as get the item from the array. Just like we used a number, a so-called **index**, to put items into the array, we can use it to get a specific item out again.

The Array class contains a bunch of smart methods for working with arrays. This example will use numbers instead of strings, just to try something else,

but it could just as easily have been strings.

Take a look:

```
int[] numbers = new int[5] { 4, 3, 8, 0, 5 };
```

With one line, we have created an array with a size of 5, and filled it with 5 integers. By filling the array like this, you get an extra advantage, since the compiler will check and make sure that you don't put too many items into the array. Try adding a number more - you will see the compiler complain about it.

Actually, it can be done even shorter, like this:

```
int[] numbers = { 4, 3, 8, 0, 5 };
```

This is short, and you don't have to specify a size. The first approach may be easier to read later on though.

Example Program 2

```
using System;
using System.Collections;

namespace ConsoleApplication1
{
    class Program
    {
    static void Main(string[] args)
    {
        int[] numbers = { 4, 3, 8, 0, 5 };

        Array.Sort(numbers);

        foreach(int i in numbers)
        Console.WriteLine(i);

        Console.ReadLine();
    }
    }
}
```

The only real new thing here is the **Array.Sort** command. It can take various parameters, for various kinds of sorting, but in this case, it simply takes our array.

As you can see from the result, our array has been sorted. The Array class has other methods as well, for instance the **Reverse()** method.

The arrays we have used so far have only had **one dimension**. However, C# arrays can be ***multidimensional***, sometimes referred to as *arrays in arrays*. **Multidimensional arrays** come in two flavors with C#: *Rectangular arrays* and *jagged arrays*. The difference is that with rectangular arrays, all the dimensions have to be the ***same size***, hence the name rectangular. A jagged array can have dimensions of ***various sizes***.

Implementation of Array in C#

An array is a group of like-typed variables that are referred to by a common name. And each data item is called an element of the array. The data types of the elements may be any valid data type like char, int, float, etc. and the elements are stored in a contiguous location. Length of the array specifies the number of elements present in the array. In C# the allocation of memory for the arrays is done dynamically. And arrays are kind of objects, therefore it is easy to find their size using the predefined functions. The variables in the array are ordered and each has an index beginning from 0. Arrays in C# work differently than they do in C/C++.

Important Points to Remember About Arrays in C#

- In C#, all arrays are dynamically allocated.
- Since arrays are objects in C#, we can find their length using member length. This is different from C/C++ where we find length using size of operator.
- A C# array variable can also be declared like other variables with [] after the data type.

The variables in the array are ordered and each has an index beginning from 0.

- C# array is an object of base type System.Array.
- Default values of numeric array and reference type elements are set to be respectively zero and null.
- A jagged array element are reference types and are initialized to null.
- Array elements can be of any type, including an array type.

- Array types are reference types which are derived from the abstract base type Array. These types implement IEnumerable and for it, they use foreach iteration on all arrays in C#.

Table 1. Array Properties

Property	Description
IsFixedSize	It is used to get a value indicating whether the Array has a fixed size or not.
IsReadOnly	It is used to check that the Array is read-only or not.
IsSynchronized	It is used to check that access to the Array is synchronized or not.
Length	It is used to get the total number of elements in all the dimensions of the Array.
LongLength	It is used to get a 64-bit integer that represents the total number of elements in all the dimensions of the Array.
Rank	It is used to get the rank (number of dimensions) of the Array.
SyncRoot	It is used to get an object that can be used to synchronize access to the Array.

Table 2. Array Methods

Method	Description
AsReadOnly<T>(T[])	It returns a read-only wrapper for the specified array.
BinarySearch(Array,Int32,Int32,Object)	It is used to search a range of elements in a one-dimensional sorted array for a value.
BinarySearch(Array,Object)	It is used to search an entire one-dimensional sorted array for a specific element.
Clear(Array,Int32,Int32)	It is used to set a range of elements in an array to the default value.
Clone()	It is used to create a shallow copy of the Array.
Copy(Array,Array,Int32)	It is used to copy elements of an array into another array by specifying starting index.
CopyTo(Array,Int32)	It copies all the elements of the current one-dimensional array to the specified one-dimensional array starting at the specified destination array index

CreateInstance(Type,Int32)	It is used to create a one-dimensional Array of the specified Type and length.
Empty<T>()	It is used to return an empty array.
Finalize()	It is used to free resources and perform cleanup operations.
Find<T>(T[],Predicate<T>)	It is used to search for an element that matches the conditions defined by the specified predicate.
IndexOf(Array,Object)	It is used to search for the specified object and returns the index of its first occurrence in a one-dimensional array.
Initialize()	It is used to initialize every element of the value-type Array by calling the default constructor of the value type.
Reverse(Array)	It is used to reverse the sequence of the elements in the entire one-dimensional Array.
Sort(Array)	It is used to sort the elements in an entire one-dimensional Array.
ToString()	It is used to return a string that represents the current object.

Example Program 3

```
using System;
namespace CSharpProgram
{
   class Program
   {
      static void Main(string[] args)
      {
         // Creating an array
         int[] arr = new int[6] { 5, 8, 9, 25, 0, 7 };
         // Creating an empty array
         int[] arr2 = new int[6];
         // Displaying length of array
         Console.WriteLine("Length of first array: "+arr.Length);
         // Sorting array
         Array.Sort(arr);
         Console.Write("First array elements: ");
         // Displaying sorted array
         PrintArray(arr);
         // Finding index of an array element
         Console.WriteLine("\nIndex position of 25 is "+Array.IndexOf(arr,25));
         // Coping first array to empty array
         Array.Copy(arr, arr2, arr.Length);
         Console.Write("Second array elements: ");
         // Displaying second array
         PrintArray(arr2);
         Array.Reverse(arr);
         Console.Write("\nFirst Array elements in reverse order: ");
         PrintArray(arr);
      }
      // User defined method for iterating array elements
      static void PrintArray(int[] arr)
      {
         foreach (Object elem in arr)
         {
            Console.Write(elem+" ");
         }
      }
   }
}
```

```
Length of first array: 6
First array elements: 0 5 7 8 9 25
Index position of 25 is 5
Second array elements: 0 5 7 8 9 25
First Array elements in reverse order: 25 9 8 7 5 0
```