Array(Reference Type)

Arrays are sequence of data of the same data type. You can access an individual element of an array by means of its position (index).
In C#, array notation is very similar to that used by C/C++ but it differs in two things:

- You can't write square brackets to the right of the variable name.
- You DON'T specify the size of array when declaring it.

All C# array are derived from the System.Array base class.
To define an array, this is done in 2 steps:

1. declare array
2. create array

The general form of declaring an array:

Data_type[] array_name;

The size of an array is established when it is created not when it is declared.

- In C#, an array element access expression is automatically checked to ensure that the index is valid. This is used to ensure that C# is a type safe language. Otherwise, it will throw OutofRangeException.
- We can't shrink or expand the array
- Declaring an array doesn't mean creating an array instance, this is because array are reference type not value type.
- The array are implicitly initialized by 0 or false if it is of type boolean.
- When initializing array, you must explicitly initialize all array element.

Single Dimension Array

```
string[] books;              //declare
books = new string[3];       //create array of 3 element
```

OR

```
Books = new string[3] {"C#", "DotNet", "VB.Net"}; //create and init
```

OR(Declare and create at 1 step)

```
string[] books = {"C#", "DotNet", "VB.Net"}; //will set the array 3
```

OR(Declare and create at 1 step)

```
string[] books = new string[3]{"C#", "DotNet", "VB.Net"};
```

Two Dimension Array

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

```
grid[1, 1] = 1;
grid[1, 2] = 2;
```
Create and initialize:
```
int[,] grid = {{5, 4, 3}, {2, 1, 0}};
```
OR
```
int[,] grid = new int[2, 3]{{5, 4, 3}, {2, 1, 0}};
```

Declaration is important in compilation. While creation is done at runtime
So, you can make the size of array variable, which is very similar to Dynamic allocation.

Example:
```
int rows;
int col;
Console.WriteLine("Enter number of rows");
rows = int.Parse(Console.ReadLine());
Console.WriteLine("Enter number of columns");
col = int.Parse(Console.ReadLine());
int[,] ar;
ar = new int[rows, col];
```

As with other objects, when you assign one array reference variable to another, you are simply changing the object to which the variable refers. You are not causing a copy of the array to be made, nor are you causing the contents of one array to be copied to the other.

Example:
```
class AssignARef
{
    public static void Main()
    {
        int i;
        int[] nums1 = new int[10];
        int[] nums2 = new int[10];
        for(i=0; i < 10; i++) nums1[i] = i;
        for(i=0; i < 10; i++) nums2[i] = -i;
        nums2 = nums1; // now nums2 refers to nums1
        nums2[3] = 99;
        Console.Write("Here is nums1 after change through nums2: ");
        for(i=0; i < 10; i++)
            Console.Write(nums1[i] + " ");
        Console.WriteLine();
    }
```

}

Output :

Here is nums1 after change through nums2: 0 1 2 99 4 5 6 7 8 9

Jagged Array:

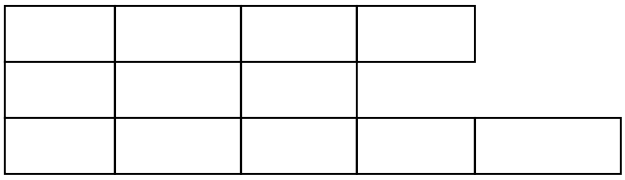
C# also allows you to create a special type of two-dimensional array called a *jagged array.* A jagged array is an *array of arrays* in which the length of each array can differ. Thus, a jagged array can be used to create a table in which the lengths of the rows are not the same.

Jagged arrays are declared by using sets of square brackets to indicate each dimension. For example, to declare a two-dimensional jagged array, you will use this general form:

*type[ ] [ ] ar*ray-name = new *type*[*size*][ ];

Here, *size* indicates the number of rows in the array. The rows, themselves, have not been allocated. Instead, the rows are allocated individually. This allows for the length of each row to vary. For example, the following code allocates memory for the first dimension of **jagged** when it is declared. It then allocates the second dimensions manually.

```
int[][] jagged = new int[3][];
jagged[0] = new int[4];
jagged[1] = new int[3];
jagged[2] = new int[5];
```



Once a jagged array has been created, an element is accessed by specifying each index within its own set of brackets. For example, to assign the value 10 to element 2, 1 of **jagged**, you would use this statement:

jagged[2][1] = 10;

Properties of Arrays:

Length: returns number of element in the array.

```
int[] x;
x = new int[7];
int y = x.Length;    //y = 7
int[,] a;
a = new int[3, 4];
int b = a.Length;    //b = 12 (3*4)
```

Rank: return the dimension of array

```
int z = x.Rank;      //z = 1;
int c = a.Rank;      //c = 2;
```

Methods of Arrays:

```
int[] data = {3, 4, 9, 1, 17};
System.Array.Sort(data);          //sort the array
```

Clear(): set a range of elements in the array to zero, false or null reference
System.Array.Clear(array_name, starting_index, numberofelement);

Clone(): this method creates a new array instance whose elements are copies of the elements of the cloned array.

```
int[] cl = (int[])data.Clone();
```

- If the array being copied contains references to objects, the references will be copied and not the objects. Both arrays will refer to the same objects

GetLength(): this method returns the length of a dimension provided as an integer.

```
Ex:  int [,] data = {{0, 1, 2, 3}, {4, 5, 6, 7}};
     int dim0 = data.GetLength(0); //2
     int dim1 = data.GetLength(1); //4
```

IndexOf(): this method returns the integer index of the first occurrence of a value provided as argument or -1 if not found. (used in 1-dimension array)

```
int[] data = {4, 6, 3, 8, 9, 3};
int Loc – System.Array.IndexOf(data, 9); //4
```

A method may return array:

```
class Example
{
    static void Main()
    {
        int []array = CreateArray(42);
    }
    static int[] CreateArray(int size)
    {
        int []created = new int[size];
        return created;
    }
}
```

You can also return arrays of rank greater than one.

Using foreach statement

The C# allows you to iterate over all items within a collection (ex: array,…)
Ex:

```
int []ar = new int[]{0, 1, 2, 3, 4, 5};
foreach(int I in ar)
{
        Console.WriteLine(i);
}
```

Note:

1. You can't modify the elements in a collection by using a foreach statement because the iteration variable is implicitly readonly.

```
foreach(int i in ar)
{
        i++ ;  //compile time error
        Console.WriteLine(i);
}
```

2. you can use foreach statement to iterate through the values of an enumerator by using the Enum.GetValues() method, which returns an array of objects

```
public class GetValuesTest
{
 enum Colors { Red, Green, Blue, Yellow };
 enum Styles { Plaid = 0, Striped = 23, Tartan = 65, Corduroy = 78 };
        public static void Main()
        {
          Console.WriteLine("The values of the Colors Enum are:");
          foreach(int i in Enum.GetValues(typeof(Colors)))
                        Console.WriteLine(i);
          Console.WriteLine();
          Console.WriteLine("The values of the Styles Enum are:");
          foreach(int i in Enum.GetValues(typeof(Styles)))
                        Console.WriteLine(i);
        }
}
```

Classes and Object Oriented

Is a named syntactic construct that describes common behavior and attribute.
It is a data structure that includes both data and function.
All classes are reference type and inherit from object class directly or indirectly.
There is also: class member(or static members).
Ex:

```
class MyClass
{
        private int x;
        public int GetX()
        {
                return x;
        }
}
```

Each member in the class (class member or static member) must have an access modifier (public, private or protected)
The class itself is a template, so to deal with the class we must have an object from this class.
To have an object of a class:
1) Declaration : MyClass m;
2) Creation : m = new MyClass(); //Done at run time.

Note: object of a class is a reference type.

Inheritance:"Is Kind of"

No multiple inheritance in C #, only single inheritance
Class hierarch, many level of inheritance may be used. Optimum number of levels : 5 ->7

Polymorphism:

It is done by function overloading, and is better used in the late binding.

Early and Late binding:

When making a method call directly on an object not through a base class operation, the method call is resolved at compile time. This is called early binding or static binding.
When making a method call indirectly on an object through a base class operation, the method calls is resolved at run time. This is called late binding or dynamic binding.

Passing Reference type variable to a function:

Ex:

```
class C1
{
}

class C2
{
        public void MyFunction(C1 x)
        {

        }
}

class C3
{
        public static void Main()
        {
                C1 L;
                C2 M;
                L = new C1();
                M = new C2();
                M.MyFunction(L);
        }
}
```

With reference type variable: either call by value or call by reference it will be considered as a call by reference

The this object is sent in calling the function of an object, it is the reference to the calling object.

1) It is used in chaining (Multiple assignments)

2)
```
        class C2
        {
                int x;
                void MyFunction()
                {
                        int x;
                        x = 3;          //local variable
                        this.x = 3;     //member variabe
                }
        }
```

Static classes:

C# 2.0 added the ability to create static classes. There are two key features of a static class. First, no object of a static class can be created. Second, a static class must contain only static members. The main benefit of declaring a class static is that it enables the compiler to prevent any instances of that class from being created. Thus, the addition of static classes does not add any new functionality to C#. It does, however, help prevent errors.

Within the class, all members must be explicitly specified as static. Making a class static does not automatically make its members static.

The main use of a static class is to contain a collection of related static methods.

static class Myclass{}

String Class

Methods

1) Insert: insert characters into string variable and return the new string
   Ex: String S = "C is great"
   S = S.Insert(2, "Sharp");
   Console.WriteLine(S); //C Sharp is great
2) Length : return the length of a string.
   Ex : String msg = "Hello";
   int slen = msg.Length;
3) Copy: creates new string by copying another string (static)
   Ex: String S1 = "Hello";
   String S2 = String.Copy(S1);
4) Concat: Creates new string from one or more string (static)
   Ex: String S3 = String.Concat("ä", "b", "c", "d"); or use + operator
   S3 = "a" + "b" + "c" +"d";
5) ToUpper, ToLower: return a string with characters converted upper or lower
   Ex: String sText = "Welcome";
   Console.WriteLine(sText.ToUpper()); //WELCOME
6) Compare: compares 2 strings and return int.(static)
   -ve (first<second) 0 (first = second) +ve (first>second)
   Ex:String S1 = "Hello"; S2 = "Welcome";
   int comp = String.Compare(S1, S2); //-ve
   there is another version of Compare that takes 3 parameters, the third is a boolean :true (ignore case) false (Check case)
7) Equals: used to compare 2 string and return a Boolean either (true) if they are the same OR (false) if they are different.

Note:

All the classes must inherit from object class either implicitly or explicitly.
The object class provides a number of common methods that are inherited by all reference type.
ToString() : return a string that represents the current object (the type name of the class), but it also can be overridden.