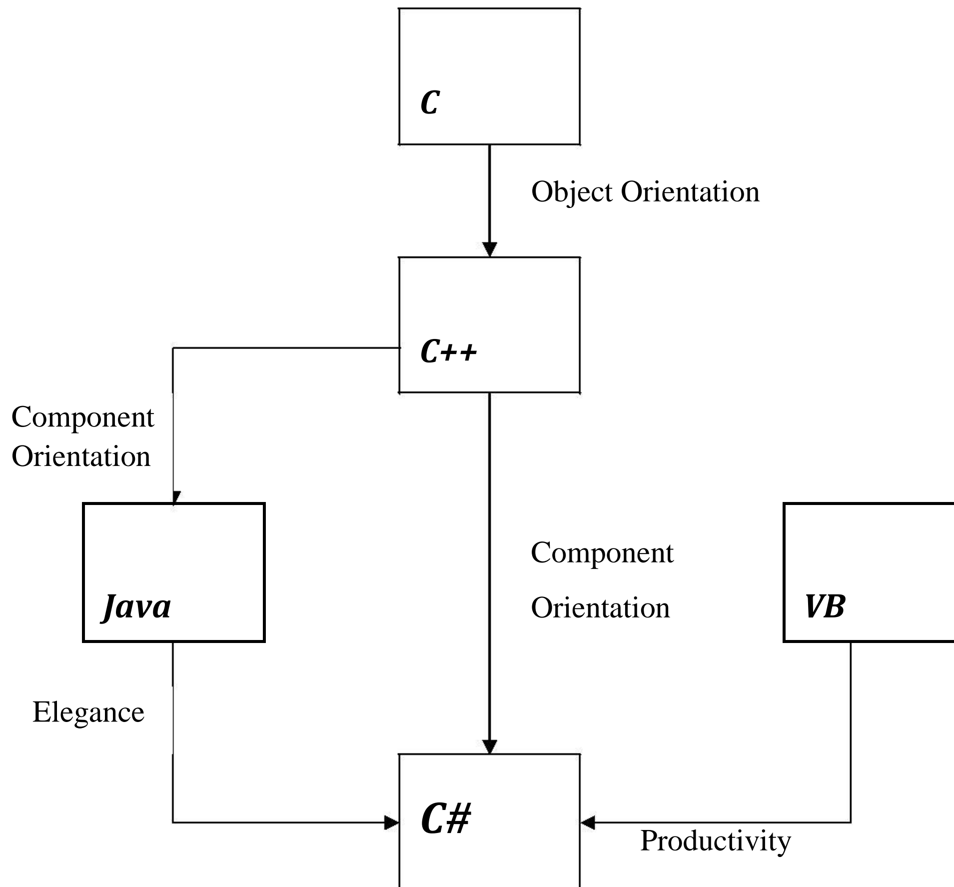
Unit-II

C -Sharp Language (C#)

2.1 Introduction

Microsoft Corporation, developed a new computer programming language C# pronounced as 'C-Sharp'. C# is a simple, modern, object oriented, and type safe programming language derived from C and C++. C# is a purely object-oriented language like as Java. It has been designed to support the key features of .NET framework.
Like Java, C# is a descendant language of C++ which is descendant of C language.



C# modernize C++ by enhancing some of its features and adding a few new features. C# borrows Java's features such as grouping of classes, interface and implementation together in one file so the programmers can easily edit the codes. C# also handles objects using reference, the same way as Java.
C# uses VB's approach to form designing, namely, dragging controls from a tool box, dropping them onto forms, and writing events handlers for them.

Comparing C# to C++ and Java

C# versus Java

C# and Java are both new-generation languages descended from a line including C and C++. Each includes advanced features, like garbage collection, which remove some of the low level maintenance tasks from the programmer. In a lot of areas they are syntactically similar.
Both C# and Java compile initially to an intermediate language: C# to Microsoft Intermediate Language (MSIL), and Java to Java bytecode. In each case the intermediate language can be run -

by interpretation or just-in-time compilation - on an appropriate 'virtual machine'. In C#, however, more support is given for the further compilation of the intermediate language code into native code.
C# contains more primitive data types than Java, and also allows more extension to the value types. For example, C# supports 'enumerations', type -safe value types which are limited to a defined set of constant variables, and 'structs', which are user-defined value types.
Unlike Java, C# has the useful feature that we can overload various operators.
Like Java, C# gives up on multiple class inheritance in favour of a single inheritance model extended by the multiple inheritance of interfaces. However, polymorphism is handled in a more complicated fashion, with derived class methods either 'overriding' or 'hiding' super class methods C# also uses 'delegates' - type-safe method pointers. These are used to implement event-handling. In Java, multi -dimensional arrays are implemented solely with single-dimensional arrays (where arrays can be members of other arrays. In addition to jagged arrays, however, C# also implements genuine rectangular arrays.

C# versus C++

Although it has some elements derived from Visual Basic and Java, C++ is C#'s closest relative. In an important change from C++, C# code does not require header files. All code is written inline. As touched on above, the .NET runtime in which C# runs performs memory management, taking care of tasks like garbage collection. Because of this, the use of pointers in C# is much less important than in C++. Pointers can be used in C#, where the code is marked as 'unsafe', but they are only really useful in situations where performance gains are at an absolute premium.
Speaking generally, the 'plumbing' of C# types is different from that of C++ types, with all C# types being ultimately derived from the 'object' type. There are also specific differences in the way that certain common types can be used. For instance, C# arrays are bounds checked unlike in C++, and it is therefore not possible to write past the end of a C# array.
C# statements are quite similar to C++ statements. To note just one example of a difference: the 'switch' statements has been changed so that 'fall-through' behavior is disallowed.
As mentioned above, C# gives up on the idea of multiple class inheritance. Other differences relating to the use of classes are: there is support for class 'properties' of the kind found in Visual Basic, and class methods are called using the . operator rather than the :: operator.

Features of C#

1. Simplicity

All the Syntax of java is like C++. There is no preprocessor, and much larger library. C# code does not require header files. All code is written inline.

2. Consistent behavior

C# introduced an unified type system which eliminates the problem of varying ranges of integer types. All types are treated as objects and developers can extend the type system simply and easily.

3. Modern programming language

C# supports number of modern features, such as:

- Automatic Garbage Collection
- Error Handling features
- Modern debugging features
- Robust Security features

4. Pure Object- Oriented programming language

In C#, every thing is an object. There are no more global functions, variable and constants.
It supports all three object oriented features:

- Encapsulation

- Inheritance
- Polymorphism

5. Type Safety
Type safety promotes robust programming. Some examples of type safety are:
- All objects and arrays are initialized by zero dynamically
- An error message will be produced , on use of any uninitialized variable
- Automatic checking of array out of bound and etc.

6. Feature of Versioning
Making new versions of software module work with the existing applications is known as versioning. Its achieve by the keywords **new** and **override.**

7. Compatible with other language
C# enforces the .NET common language specifications (CLS) and therefore allows inter-operation with other .NET language.

8. Inter-operability
C# provides support for using COM objects, no matter what language was used to author them. C# also supports a special feature that enables a program to call out any native API.

A Simple C# Program
Let's begin in the traditional way, by looking at the code of a Hello World program (note that the tabulation and line numbers are included just for the sake of readability).

```
1.    Using System;
2.    public class HelloWorld
3.    {
4.        public static void Main()
5.        {
6.            // This is a single line comment
7.            /* This is a
8.            multiple
9.            line comment */
10.           Console.WriteLine("Hello World! ");
11.       }
12.   }
```

- The first thing to note about C# is that it is case-sensitive. You will therefore get compiler errors if, for instance, you write 'console' rather than 'Console'.
- The second thing to note is that every statement finishes with a semicolon (;) or else takes a code block within curly braces.

Explanation of Program
Line 1 : using System;
we are using the System namespace (namespaces are also covered in chapter 7). The point of this declaration is mostly to save ourselves time typing. Because the 'Console' object used in line 10 of the code actually belongs to the 'System' namespace, its fully qualified name is 'System.Console'. However, because in line 1 we declare that the code is using the System namespace, we can then leave off the 'System.' part of its name within the code.
Line 2: public class HelloWorld

As C# is an object-oriented language, C# programs must be placed in classes (classes are discussed in chapter 5 but if you are new to object orientation we suggest that you first read some introductory material). This line declares the class to be named 'HelloWorld'.

Line 4: public static void Main()

When compiled and run, the program above will automatically run the 'Main' method declared and begun in this line. Note again C#'s case-sensitivity - the method is 'Main' rather than 'main'.

Line 3,11 and 5,12 :

These lines are uses the '{' for starting braces and '}' for closing braces of block. **Lines 6-9** : Comments

('//' uses for single line and '/* -- - - */' uses for multiple line comments)

These lines of the program are ignored by the compiler, being comments entered by the programmer for his own benefit.

Line 6 shows a single line comment, in which everything on the line after the two forward slashes is ignored by the compiler.

Lines 7-9 demonstrate a multi-line comment, in which everything between the opening /* and closing */ is ignored, even when it spans multiple lines.

Line 10:

The statement on this line calls the 'WriteLine' method of the Console class in the System namespace. It should be obvious how this works in the given example - it just prints out the given string to the 'Console' (on PC machines this will be a DOS prompt).

Instruction for Saving the Program

In order to run the program, it must first be saved in a file. Unlike in Java, the name of the class and the name of the file in which it is saved do not need to match up, although it does make things easier if you use this convention. In addition, you are free to choose any extension for the file, but it is usual to use the extension '.cs'.

Writing program in Computer

There are two ways of program writing in computer • Using Text Editor

Using Visual Studio.NET

2.2 Data Types. Identifiers, Variables, Constants and Literals

Identifiers & Variables

Identifiers refer to the names of variables, functions arrays, classes, etc. created by programmer. They are fundamental requirement of any language. Each language has its own rules for naming these identifiers.

To name the variables of your program, you must follow strict rules. In fact, everything else in your program must have a name.

There are some rules you must follow when naming your objects. On this site, here are the rules we will follow:

- The name must start with a letter or an underscore
- After the first letter or underscore, the name can have letters, digits, and/or underscores
- The name must not have any special characters other than the underscore
- The name cannot have a space

C# is case-sensitive. This means that the names Case, case, and CASE are completely different. For example, the main function is always written Main.

C# Keywords

C# uses a series of words, called keywords, for its internal use. This means that you must avoid naming your objects using one of these keywords. They are:

abstract	const	extern	int	out	short	typeof
as	continue	false	interface	override	sizeof	uint
base	decimal	finally	internal	params	stackalloc	ulong
bool	default	fixed	is	private	static	unchecked
break	delegate	float	lock	protected	string	unsafe
byte	do	for	long	public	struct	ushort
case	double	foreach	namespace	readonly	switch	using
catch	else	goto	new	ref	this	virtual
char	enum	if	null	return	throw	void
checked	event	implicit	object	sbyte	true	volatile
class	explicit	in	operator	sealed	try	while

Data types

C# is a type-safe language. Variables are declared as being of a particular type, and each variable is constrained to hold only values of its declared type.

Variables can hold either value types or reference types, or they can be pointers. Here's a quick recap of the difference between value types and reference types.

- where a variable v contains a value type, it directly contains an object with some value. No other variable v' can directly contain the object contained by v (although v' might contain an object with the same value).

- where a variable v contains a reference type, what it directly contains is something which refers to an object. Another variable v' can contain a reference to the same object referred to by v.

Value Types

C# defines the following value types:

- Primitives int i;
- Enum enum state { off, on }
- Struct struct Point{ int x, y; }

It is possible in C# to define your own value types by declaring enumerations or structs. These user-defined types are mostly treated in exactly the same way as C#'s predefined value types, although compilers are optimized for the latter. The following table lists, and gives information about, the predefined value types. Because in C# all of the apparently fundamental value types are in fact built up from the (actually fundamental) object type, the list also indicates which System types in the .Net framework correspond to these pre-defined types.

C# Type	**.Net Framework (System) type**	**Signed?**	**Bytes Occupied**	**Possible Values**
sbyte	System.Sbyte	Yes	1	-128 to 127
short	System.Int16	Yes	2	-32768 to 32767
int	System.Int32	Yes	4	-2147483648 to 2147483647
long	System.Int64	Yes	8	-9223372036854775808 to 9223372036854775807
byte	System.Byte	No	1	0 to 255
ushort	System.Uint16	No	2	0 to 65535
uint	System.UInt32	No	4	0 to 4294967295

ulong	System.Uint64	No	8	0 to 18446744073709551615
float	System.Single	Yes	4	Approximately ±1.5 x 10^{-45} to ±3.4 x 10^{38} with 7 significant figures
double	System.Double	Yes	8	Approximately ±5.0 x 10^{-324} to ±1.7 x 10^{308} with 15 or 16 significant figures
decimal	System.Decimal	Yes	12	Approximately ±1.0 x 10^{-28} to ±7.9 x 10^{28} with 28 or 29 significant figures
char	System.Char	N/A	2	Any Unicode character (16 bit)
bool	System.Boolean	N/A	1 / 2	true or false

In the following lines of code, two variables are declared and set with integer values.

```
int x = 10;
int y = x;
y = 20; // after this statement x holds value 10 and y holds value 20
```

Reference Types

The pre-defined reference types are object and string, where object - is the ultimate base class of all other types. New reference types can be defined using 'class', 'interface', and 'delegate' declarations. There fore the reference types are :

Predefined Reference Types

- Object
- String

User Defined Reference Types

- Classes
- Interfaces
- Delegates
- Arrays

Reference types actually hold the value of a memory address occupied by the object they reference. Consider the following piece of code, in which two variables are given a reference to the same object (for the sake of the example, this object is taken to contain the numeric property 'myValue').

```
object x = new
object(); x.myValue =
10; object y = x ;
y.myValue = 20; // after this statement both
            x.myValue // and y.myValue equal 20
```

This code illustrates how changing a property of an object using a particular reference to it is reflected in all other references to it. Note, however, that although strings are reference types, they work rather more like value types. When one string is set to the value of another, eg

```
string s1 = "hello";
string s2 = s1;
```

Then s2 does at this point reference the same string object as s1. However, when the value of s1 is changed, for instance with

```
s1 = "goodbye";
```

what happens is that a new string object is created for s1 to point to. Hence, following this piece of code, s1 equals "goodbye", whereas s2 still equals "hello".

The reason for this behaviour is that string objects are 'immutable'. That is, the properties of these objects can't themselves change. So in order to change what a string variable references, a new string object must be created.

Boxing

C# allows you convert any value type to a corresponding reference type, and to convert the resultant 'boxed' type back again. The following piece of code demonstrates boxing. When the second line executes, an object is initiated as the value of 'box', and the value held by i is copied across to this object. It is interesting to note that the runtime type of box is returned as the boxed value type; the 'is' operator thus returns the type of box below as 'int'.

```
int i = 123;
object box = i;
if (box is int)
{Console.Write("Box contains an int");} // this line is printed
```

When boxing occurs, the contents of value type are copied from stack into memory allocated into the managed heap. The new reference type created contains a copy of the value type, and can be used by other types that expect an object reference. The value contained in the value type and the created reference types are not associated in any way (except that they contain the same values). If we change the original value type, the refernce type is not affected.

The following code explicitly **unboxes** a reference type into a value type:

```
object o;
int i = (int) o;
```

When unboxing occurs, memory is copied from the managed heap to the stack.

2.3 Array and Strings

Arrays

An array is a group or collection of similar values. An array contains a number of variables, which are accessed through computed indices. The various value contained in an array are also called the elements of array. All elements of an array have to be of same type, and this type is called the element type of the array. The element of an array can be of any type including an array type.

An array has a rank that determines the number of indices associated wth each array elements. The rank of an array is also referred as the dimension of the array. An array may be :

- Single Dimensional
- Multi Dimensional

An array with a rank of one is called single-dimensional array, and an array with a rank greater than one is called a multi dimensional array.
Each dimension of array has an associated length, which is an integer number greater than or equal to zero. For a dimension of length n, indices can range from 0 to n-1. in C#, array types are categorized under the reference types alongside with classes and interfaces.

Single Dimensional Array

Single -dimensional arrays have a single dimension (ie, are of rank 1). The process of creation of arrays is basically divided into three steps:

1. Declaration of Array
2. Memory Allocation for Array
3. Initialization of Array

Declaration of Array

To declare an array in C# place a pair of square brackets after the variable type. The syntax is given below :

```
type[] arrayname;
```

For Example:

```
int[] a; float[]
marks;
double[] x;
int[] m,n;
```

You must note that we do not enter the size of the arrays in the declaration.

Memory Allocation for Array

After declaring an array, we need to allocate space and defining the size. Declaring arrays merely says what kind of values the array will hold. It does not create them. Arrays in C# are objects, and you use the new keyword to create them. When you create an array, yu must tell the compiler how many components will be stored in it. Here is given the syntax:

arrayname = **new** type[size];

For Example:

```
a = new int[5];
      marks = new float[6];
      x = new double[10];
      m = int[100];
n = int [50];
```

It is also possible to combine the two steps, declaration and memory allocation of array, into one as shown below:

```
int[] num = new int [5];
```

Initialization of Array

This step involves placing data into the array. Arrays are automatically assigned the default values associated with their type. For example, if we have an array of numerical type, each element is set to number 0. But explicit values can be assigned as and when desired.
Individual elements of an array are referenced by the array name and a number that represents their position in the array. He number you use to identify them are called subscripts or indexes into the array.
Subscripts are consecutive integers beginning with 0. thus the array “num” above has components **num[0], num[1], num[2], num[3],** and **num[4].**
The initialization process is done using the array subscripts as shown:

```
arrayname[subscript] = value;
```

For Example:

```
num[0] = 5;
num[1] = 15;
num[2] = 52;
num[3] = 45;
num[4] = 57;
```

We can also initialize arrays automatically in the same way as the ordinary variables when they are declared, as shown below:

```
type[] arrayname = { list of values };
```

the list of variables separated by commas and defined on both ends by curly braces. You must note that no size is given in this syntax. The compiler space for all the elements specified in the list.

For Example:

```
int[] num = {5,15,52,45,57};
```

You can combine all the steps, namely declaration, memory allocation and initialization of arrays like as:

```
int[] num = new int [5] {5,15,52,45,57};
```

You can also assign an array object to another. For Example

```
int[] a = { 10, 20,
30}; int[] b;
b=a;
```

The above example is valid in C#. Both the array will have same values.

Example

```
using system;
class Number
{
	public static void Main()
	{
		int [] num = {10, 20, 30, 40,
		50}; int n = num.Length;
		// Length is predefined attribute to access the size of
		array Console.Write(" Elements of array are :");
		for(int i=0; i<n; i++)
		{
			Console.WriteLine(num[i]);
		}

		int sum =0;
		for(int i=0; i<n; i++)
		{
			sum = sum + num[i]);
		}
		Console.WriteLine(" The sum of elements :"+sum);
}
```

OUTPUT:

Elements of array
are: 10 20 30 40 50

The sum of elements :150

Multi Dimensional Array

C# supports two types of multidimensional arrays:

- Rectangular Array
- Jagged Array

Rectangular Arrays

A rectangular array is a single array with more than one dimension, with the dimensions' sizes fixed in the array's declaration. The following code creates a 2 by 3 multi-dimensional array:

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

As with single-dimensional arrays, rectangular arrays can be filled at the time they are declared. For instance, the code

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

creates a 2 by 3 array with the given values. It is, of course, important that the given values do fill out exactly a rectangular array.

The **System.Array** class includes a number of methods for determining the size and bounds of arrays. These include the methods **GetUpperBound(int i)** and **GetLowerBound(int i)**, which

return, respectively, the upper and lower subscripts of dimension i of the array (note that i is zero based, so the first array is actually array 0).
For instance, since the length of the second dimension of squareArray is 3, the

```
expression squareArray.GetLowerBound(1)
```

returns 0, and the expression

```
squareArray.GetUpperBound(1)
```

returns 2.
System.Array also includes the method **GetLength(int i)**, which returns the number of elements in the ith dimension (again, zero based).
The following piece of code loops through squareArray and writes out the value of its elements.

```
for(int i = 0; i < squareArray.GetLength(0); i++)
 for (int j = 0; j < squareArray.GetLength(1); j++)
    Console.WriteLine(squareArray[i,j]);
```

A foreach loop can also be used to access each of the elements of an array in turn, but using this construction one doesn't have the same control over the order in which the elements are accessed.

Jagged Arrays

Using jagged arrays, one can create multidimensional arrays with irregular dimensions. This flexibility derives from the fact that multidimensional arrays are implemented as arrays of arrays. The following piece of code demonstrates how one might declare an array made up of a group of 4 and a group of 6 elements:

```
int[][] jag = new int[2][];
jag[0] = new int [4];
jag[1] = new int [6];
```

The code reveals that each of jag[0] and jag[1] holds a reference to a single-dimensional int array. To illustrate how one accesses the integer elements: the term jag[0][1] provides access to the second element of the first group.
To initialise a jagged array whilst assigning values to its elements, one can use code like the following:

```
int[ ][ ] jag = new int[ ][ ] {new int[ ] {1, 2, 3, 4}, new int[ ] {5, 6, 7, 8, 9, 10}};
```

Be careful using methods like GetLowerBound, GetUpperBound, GetLength, etc. with jagged arrays. Since jagged arrays are constructed out of single-dimensional arrays, they shouldn't be treated as having multiple dimensions in the same way that rectangular arrays do.
To loop through all the elements of a jagged array one can use code like the following:

```
for (int i = 0; i < jag.GetLength(0); i++)
   for (int j = 0; j < jag[i].GetLength(0);
      j++) Console.WriteLine(jag[i][j]);
```

or

```
for (int i = 0; i < jag.Length; i++)
   for (int j = 0; j < jag[i].Length; j++)
      Console.WriteLine(jag[i][j]);
```

Strings

A string is an empty space, a character, a word, or a group of words that you want the compiler to consider "as is", that is, not to pay too much attention to what the string is made of, unless you explicitly ask it to. This means that, in the strict sense, you can put in a string anything you want. Primarily, the value of a string starts with a double quote and ends with a double-quote. An example of a string is "Welcome to the World of C# Programming!". You can include such a string in the **Console.Write()** method to display it on the console. Here is an example:

Example using
System; class
BookClub

```
{
 static void Main()
  {
   Console.WriteLine("Welcome to the World of C# Programming!");
  }
}
```

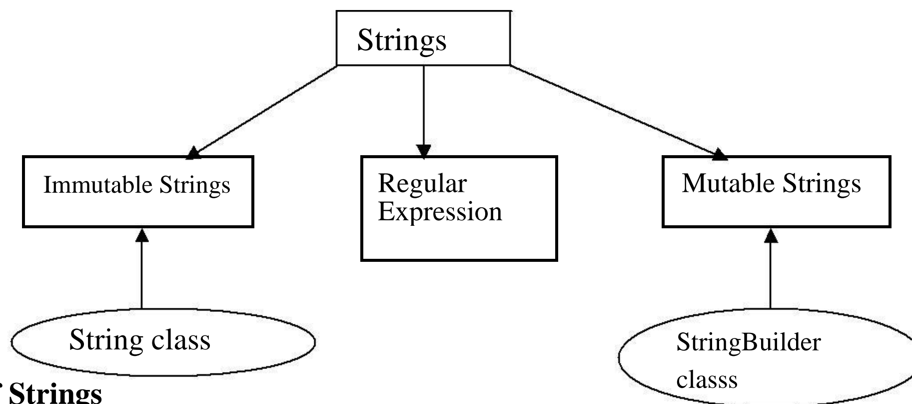
OUTPUT:
Welcome to the World of C# Programming!

Types of String

There are two types of string in C#:

1) **Immutable strings**
2) **Mutable strings**

The immutable strings are can't be modify and mutable strings are modifiable.C# also supports a feature of *regular expression* that can be used for complex strings manipulations and pattern matching.



Handling of Strings

We can create immutable strings using **string** or **String** objects in number of ways. There are a some techniques to handling the immutable strings:

Assigning String

```
string s1;
s1 = "Welcome";
```

or

```
string s1 = "Welcome";
```

Copying String

```
string s2 = s1;
```

or

```
string s2 = string.Copy(s1);
```

Concatenating Strings

```
string s3 = s1 + s2;
```

or

```
string s3 = string.Concat(s1,s2);
```

Reading from Console

```
string s1 = Console.ReadLine();
```

Converting Number to String

```
int num = 100;
string s1= num.ToString();
```

Inserting String

```
string s1 = Wel;
string s2 = s1.insert(3,"come");
// s2 = Welcome
string s3 = s1.insert(3,"don");
// s3 = Weldon;
```

Comparing Strings

```
int n = string. Compare(s1,s2);
```

This statement will perform case-sensitive comparison and returns integer values for different conditions. Such as:

- If s1 is equal to s2 it will return zero.
- If s1 is greater than s2 it will return positive integer (1).
- If s1 is less than s2 it will return negative integer(-

1). Or you can use following statement:

```
bool a = s2.Equals(s1); bool
b = string.Equal(s1,s2);
```

Above statements will return a Boolean value **true** (if equal) or **false**(if not equal).

Or you can also use the "==" operator for comparing the strings. Like as:

```
if ( s1 == s2)
   Console.Write(" both are equal");
```

In this statement, it will return a Boolean value **true** (if equal) or **false**(if not equal).

Mutable String

Mutable strings are those strings, which can be modify dynamically. This type of strings are created using **StringBuilder** class. For Example:

```
StringBuilder s1 = new StringBuilder("Welcome");
StringBuilder s2 = new StringBuilder( );
```

The string **str1** is created with an initial size of seven characters and **str2** is created as an empty string. They can grow dynamically as more character added to them. Mutual string are referred as a *dynamic strings*.
The StringBuilder class supports many methods that are useful for manipulating dynamic strings. Some of the most common methods are listed below:

Method	Operation
Append( )	Append a string
AppendFormat( )	Append string using specific format
EnsureCapacity( )	Ensure sufficient size
Insert( )	Insert a string at a specified position
Remove( )	Remove specified character
Replace( )	Removes previous string with new one

StringBuilder also provides some attributes to access some properties of strings, such as:

Attributes	Puspose
Capacity	To retrieve or set the number of characters the object can hold
Length	To retrieve or set the length
MaxCapacity	To retrieve maximum capacity of the object
[ ]	To get or set a character at a specified position

Example

```
using System.Text;    //For using StringBuilder
using System;

class StrMethod
{
        public static void Main( )
        {
                StringBuilder s = new StringBuilder("C");
                Console.WriteLine(" Stored String is :"+ s);
                Console.WriteLine("Length of string is :"+s.Length);

                s.Append("Sharp ");
                // appending the string s

                Console.WriteLine(" After Append String is :"+ s);
                Console.WriteLine("Length of string is :"+s.Length);

                s.Insert(7,"Language");
                // inserting the string at last in s

                Console.WriteLine("After Insertion String is:"+ s);
                Console.WriteLine("Length of string is :"+s.Length);

                int n = s.Length;

                s[n] = "!";
```

```
            Console.WriteLine(" At Last String is :"+ s);
        }
}
```

OUTPUT:

Stored String is : C
Length of String is : 1

After Append string is : CSharp
After Insertion String is : CSharp Language
At Last String is : CSharp Language!