C# Class

```
public class MyClass
{
    public string  myField = string.Empty;

    public MyClass()
    {
    }

    public void MyMethod(int parameter1, string parameter2)
    {
        Console.WriteLine("First Parameter {0}, second parameter {1}",
                                                    parameter1, parameter2);
    }

    public int MyAutoImplementedProperty { get; set; }

    private int myPropertyVar;

    public int MyProperty
    {
            get { return myPropertyVar; }
            set { myPropertyVar = value; }
    }
}
```

Property

A property can be defined using getters and setters, as below:

Example: Property in C#

```
private int _myPropertyVar;

public int MyProperty
{
    get { return _myPropertyVar; }
    set { _myPropertyVar = value; }
}
```

Property encapsulates a private field. It provides getters (get{}) to retrieve the value of the underlying field and setters (set{}) to set the value of the underlying field. In the above example, _myPropertyVar is a private field which cannot be accessed directly. It will only be accessed via MyProperty. Thus, MyProperty encapsulates _myPropertyVar.

You can also apply some addition logic in get and set, as in the below example.

Example: Property in C#

```
private int _myPropertyVar;

public int MyProperty
{
    get {
        return _myPropertyVar / 2;
    }

    set {
        if (value > 100)
            _myPropertyVar = 100;
        else
            _myPropertyVar = value; ;
    }
}
```

Auto-implemented Property

From C# 3.0 onwards, property declaration has been made easy if you don't want to apply some logic in get or set.

The following is an example of an auto-implemented property:

Example: Auto implemented property in C#

```
public int MyAutoImplementedProperty { get; set; }
```

Notice that there is no private backing field in the above property example. The backing field will be created automatically by the compiler. You can work with an automated property as you would with a normal property of the class. Automated-implemented property is just for easy declaration of the property when no additional logic is required in the property accessors.

C# - Object Initializer Syntax

C# 3.0 (.NET 3.5) introduced *Object Initializer Syntax*, a new way to initialize an object of a class or collection. Object initializers allow you to assign values to the fields or properties at the time of creating an object without invoking a constructor.

Example: Object Initializer Syntax

```
public class Student
{
    public int StudentID { get; set; }
```

```
    public string StudentName { get; set; }
    public int Age { get; set; }
    public string Address { get; set; }
}

class Program
{
    static void Main(string[] args)
    {
        Student std = new Student() { StudentID = 1,
                                      StudentName = "Bill",
                                      Age = 20,
                                      Address = "New York"
                                    };
    }
}
```

In the above example, Student class is defined without any constructors. In the Main() method, we have created Student object and assigned values to all or some properties in the curly bracket at the same time. This is called object initializer syntax.

The compiler compiles the above initializer into something like the following.

Example: Object Initializer Syntax at Compile time

```
Student __student = new Student();
__student.StudentID = 1;
__student.StudentName = "Bill";
__student.Age = 20;
__student.StandardID = 10;
__student.Address = "Test";

Student std = __student;
```

Collection can be initialized the same way as class objects using collection initializer syntax.

Example: Object initializer Syntax

```
var student1 = new Student() { StudentID = 1, StudentName = "John" };
var student2 = new Student() { StudentID = 2, StudentName = "Steve" };
var student3 = new Student() { StudentID = 3, StudentName = "Bill" } ;
var student4 = new Student() { StudentID = 3, StudentName = "Bill" };
var student5 = new Student() { StudentID = 5, StudentName = "Ron" };
```

```
IList<Student> studentList = new List<Student>() {
                                                    student1,
                                                    student2,
                                                    student3,
                                                    student4,
                                                    student5
                                                };


                Console.WriteLine("Total Students: {0}",studentList.Count);
        }
}


public class Student
{
    public int StudentID { get; set; }
    public string StudentName { get; set; }

}
```

You can also initialize collections and objects at the same time.

Example: Collection initializer Syntax

```
IList<Student> studentList = new List<Student>() {
                    new Student() { StudentID = 1, StudentName = "John"} ,
                    new Student() { StudentID = 2, StudentName = "Steve"} ,
                    new Student() { StudentID = 3, StudentName = "Bill"} ,
                    new Student() { StudentID = 3, StudentName = "Bill"} ,
                    new Student() { StudentID = 4, StudentName = "Ram" } ,
                    new Student() { StudentID = 5, StudentName = "Ron" }
                };
```

You can also specify null as an element:

Example: Collection initializer Syntax

```
IList<Student> studentList = new List<Student>() {
                                    new Student() { StudentID = 1, StudentName =
"John"} ,
                                    null
                                };
```

Namespace

Namespace is a container for a set of related classes and namespaces. Namespace is also used to give unique names to classes within the namespace name. Namespace and classes are represented using a dot (.).

In C#, namespace can be defined using the namespace keyword.

Example: Namespace

```
namespace CSharpTutorials
{
    class MyClass
    {

    }
}
```

In the above example, the fully qualified class name of `MyClass` is `CSharpTutorials.MyClass`.

A namespace can contain other namespaces. Inner namespaces can be separated using (.).

Example: Namespace

```
namespace CSharpTutorials.Examples
{
    class MyClassExample
    {

    }
}
```

In the above example, the fully qualified class name of `MyClassExample` is `CSharpTutorials.Examples.MyClassExample`.

C# Implicit Typed Local Variable – var

C# 3.0 introduced the implicit typed local variable "var". Var can only be defined in a method as a local variable. The compiler will infer its type based on the value to the right of the "=" operator.

Example: Explicitly Typed Variable

```
int i = 100;// explicitly typed
var j = 100; // implicityly type
```

The following example shows how ***var*** can have a different type based on its value:

Example: Implicit Typed Variable

```
static void Main(string[] args)
{
    var i = 10;
    Console.WriteLine("Type of i is {0}",i.GetType().ToString());

    var str = "Hello World!!";
    Console.WriteLine("Type of str is {0}", str.GetType().ToString());

    var d = 100.50d;
    Console.WriteLine("Type of d is {0}", d.GetType().ToString());

    var b = true;
    Console.WriteLine("Type of b is {0}", b.GetType().ToString());
}
```

C# Data Types

C# is a strongly-typed language.

Reserved Word	.NET Type	Type	Size (bits)	Range (values)
byte	Byte	Unsigned integer	8	0 to 255
sbyte	SByte	Signed integer	8	-128 to 127
short	Int16	Signed integer	16	-32,768 to 32,767
ushort	UInt16	Unsigned integer	16	0 to 65,535
int	Int32	Signed integer	32	-2,147,483,648 to 2,147,483,647
uint	UInt32	Unsigned integer	32	0 to 4294967295
long	Int64	Signed integer	64	-9,223,372,036,854,775,808 to 9,223,372,036,854,775,807
ulong	UInt64	Unsigned integer	64	0 to 18,446,744,073,709,551,615
float	Single	Single-precision floating point type	32	-3.402823e38 to 3.402823e38
double	Double	Double-precision floating point type	64	-1.79769313486232e308 to 1.79769313486232e308
decimal	Decimal	Precise fractional or integral type that can represent decimal	128	(+ or -)1.0 x 10e-28 to 7.9 x 10e28

Reserved Word	.NET Type	Type	Size (bits)	Range (values)
		numbers with 29 significant digits		
char	Char	A single Unicode character	16	Unicode symbols used in text
bool	Boolean	Logical Boolean type	8	True or False
object	Object	Base type of all other types		
string	String	A sequence of characters		
DateTime	DateTime	Represents date and time		0:00:00am 1/1/01 to 11:59:59pm 12/31/9999

Alias vs .Net Type

In the above table of data types, first column is for data type alias and second column is actual .Net type name. For example, int is an alias (or short name) for Int32. Int32 is a structure defined in System namespace. The same way, string represent String class.

Alias	Type Name	.Net Type
byte	System.Byte	struct
sbyte	System.SByte	struct
int	System.Int32	struct
uint	System.UInt32	struct
short	System.Int16	struct
ushort	System.UInt16	struct
long	System.Int64	struct
ulong	System.UInt64	struct
float	System.Single	struct
double	System.Double	struct
char	System.Char	struct
bool	System.Boolean	struct
object	System.Object	Class
string	System.String	Class
decimal	System.Decimal	struct
DateTime	System.DateTime	struct

Value Type and Reference Type

We have learned about the data types in the previous section. In C#, these data types are categorized based on how they store their value in the memory. C# includes following categories of data types:

1. Value type

2. Reference type
3. Pointer type

Here, we will learn about value types and reference types.

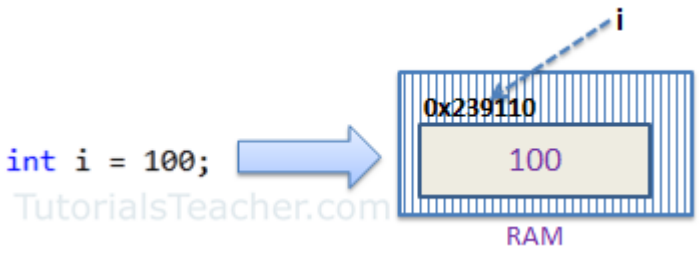
Value Type:

A data type is a value type if it holds a data value within its own memory space. It means variables of these data types directly contain their values.

All the value types derive from *System.ValueType*, which in-turn, derives from *System.Object*.

For example, consider integer variable `int i = 100;`

The system stores 100 in the memory space allocated for the variable 'i'. The following image illustrates how 100 is stored at some hypothetical location in the memory (0x239110) for 'i':



Memory allocation for Value Type

The following data types are all of value type:

- bool
- byte
- char
- decimal
- double
- enum
- float
- int
- long
- sbyte
- short
- struct
- uint
- ulong
- ushort

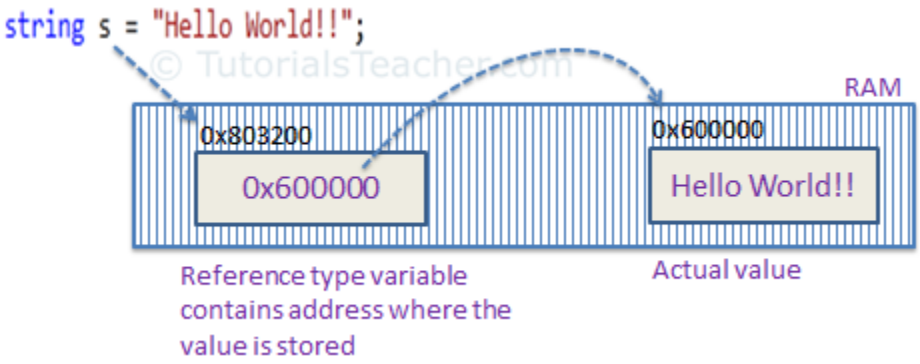
Reference Type

Unlike value types, a reference type doesn't store its value directly. Instead, it stores the address where the value is being stored. In other words, a reference type contains a pointer to another memory location that holds the data.

For example, consider following string variable:

```
string s = "Hello World!!";
```

The following image shows how the system allocates the memory for the above string variable.



Memory allocation for Reference type

As you can see in the above image, the system selects a random location in memory (`0x803200`) for the variable 's'. The value of a variable `s` is `0x600000` which is the memory address of the actual data value. Thus, reference type stores the address of the location where the actual value is stored instead of value itself.

The following data types are of reference type:

- String
- All arrays, even if their elements are value types
- Class
- Delegates

Pass by Reference

When you pass a reference type variable from one method to another, it doesn't create a new copy; instead, it passes the address of the variable. If we now change the value of the variable in a method, it will also be reflected in the calling method.

Example: Reference Type Variable

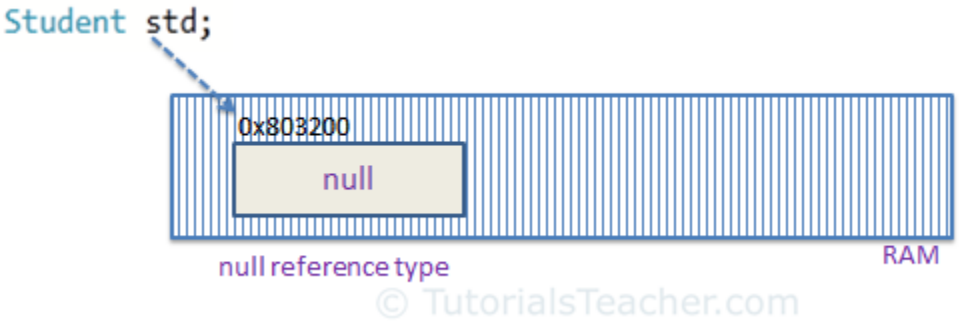
```
static void ChangeReferenceType(Student std2)
{
    std2.StudentName = "Steve";
}

static void Main(string[] args)
{
    Student std1 = new Student();
    std1.StudentName = "Bill";

    ChangeReferenceType(std1);

    Console.WriteLine(std1.StudentName);
}
```

Output

Steve

Null

Reference types have null value by default, when they are not initialized. For example, a string variable (or any other variable of reference type datatype) without a value assigned to it. In this case, it has a null value, meaning it doesn't point to any other memory location, because it has no value yet.



Null Reference type

A value type variable cannot be null because it holds a value not a memory address. However, value type variables must be assigned some value before use. The compiler will give an error if you try to use a local value type variable without assigning a value to it.

Example: Compile Time Error

```
void someFunction()
{
    int i;
```

```
    Console.WriteLine(i);
}
```

C# 2.0 introduced nullable types for value types so that you can assign null to a value type variable or declare a value type variable without assigning a value to it.

However, value type field in a class can be declared without initialization (field not a local variable in the function) . It will have a default value if not assigned any value, e.g., int will have 0, boolean will have false and so on.

Example: Value Type Field

```
class myClass
{
    public int i;
}

myClass mcls = new myClass();

Console.WriteLine(mcls.i);
```

Output

0

Points to Remember :

1. Value type stores the value in its memory space, whereas reference type stores the address of the value where it is stored.
2. Primitive data types and struct are of the 'Value' type. Class objects, string, array, delegates are reference types.
3. Value type passes byval by default. Reference type passes byref by default.
4. Value types and reference types stored in Stack and Heap in the memory depends on the scope of the variable.

C# - Struct

We have learned class in the previous section. Class is a reference type. C# includes a value type entity same as class called "structure". Structs are mainly useful to hold small data values. A structure can be defined using

the *struct* operator. It can contain parameterized constructor, static constructor, constants, fields, methods, properties, indexers, operators, events and nested types.

Structure Declaration

A structure is declared using struct keyword with public, private, or internal modifier. The default modifer is internal for the struct and its members.

The following declares the simple structure that holds data for employees.

Example: Structure

```
struct Employee
{
    public int EmpId;
    public string FirstName;
    public string LastName;
}
```

A struct object can be created with or without the `new` operator, same as primitive type variables. When you create a struct object using the `new` operator, an appropriate constructor is called.

Example: Create struct object using new keyword

```
struct Employee
{
    public int EmpId;
    public string FirstName;
    public string LastName;
}

Employee emp = new Employee();
Console.Write(emp.EmpId); // prints 0
```

In the above code, an object of the structure `Employee` is created using the `new`keyword. So, this calls the default parameterless constructor that initializes all the members to their default value.

When you create a structure object without using `new` keyword, it does not call any constructor and so all the members remain unassigned. So, you must assign values to each member before accessing them, otherwise it will give a compile time error.

Example: Create struct object without using new keyword

```
struct Employee
{
    public int EmpId;
    public string FirstName;
    public string LastName;
}

Employee emp;
Console.Write(emp.EmpId); // Compile time error

emp.EmpId = 1;
Console.Write(emp.EmpId); // prints 1
```

Constructors in Struct

A struct cannot contain parameterless constructor. It can only contain parameterized constructors or a static constructor. You can declare parameterized constructor to initialize struct members, as shown below.

Example: Parameterized Constructor in Struct

```
struct Employee
{
    public int EmpId;
    public string FirstName;
    public string LastName;

    public Employee(int empid, string fname, string lname)
    {
        EmpId = empid;
        FirstName = fname;
        LastName = lname;
    }
}

Employee emp = new Employee(10, "Bill", "Gates");

Console.Write(emp.EmpId); // prints 10
Console.Write(emp.FirstName); // prints Bill
Console.Write(emp.LastName); // prints Gates
```

Please note that you must assign values to all the members of a struct in parameterized constructor, otherwise it will give compile time error if any member remains unassigned.

A struct can include static parameterless constructor and static fields.

Example: Static Constructor in Struct

```
struct Employee
```

```
{
    public int EmpId;
    public string FirstName;
    public string LastName;

    static Employee()
    {
        Console.Write("First object created");
    }

    public Employee(int empid, string fname, string lname)
    {
        EmpId = empid;
        FirstName = fname;
        LastName = lname;
    }
}

Employee emp1 = new Employee(10, "Bill", "Gates");
Employee emp2 = new Employee(10, "Steve", "Jobs");
```

Methods and Properties in Struct

The structure can contain properties, auto-properties, or methods, same as class.

Example: Methods and Properties in Struct

```
struct Employee
{
    public int EmpId { get; set; }
    public string FirstName { get; set; }
    public string LastName { get; set; }

    public Employee(int empid, string fname, string lname)
    {
        EmpId = empid;
        FirstName = fname;
        LastName = lname;
    }

    public string GetFullName()
    {
        return FirstName + " " + LastName;
    }
}

Employee emp = new Employee(10, "Bill", "Gates");

Console.Write(emp.GetFullName()); // prints Bill Gates
```

C# - enum

In C#, enum is a value type data type. The enum is used to declare a list of named integer constants. It can be defined using the *enum* keyword directly inside a namespace, class, or structure. The enum is used to give a name to each constant so that the constant integer can be referred using its name.

Example: enum

```
enum WeekDays
{
    Monday = 0,
    Tuesday =1,
    Wednesday = 2,
    Thursday = 3,
    Friday = 4,
    Saturday =5,
    Sunday = 6
}

Console.WriteLine(WeekDays.Friday);
Console.WriteLine((int)WeekDays.Friday);
```

By default, the first member of an enum has the value 0 and the value of each successive enum member is increased by 1. For example, in the following enumeration, Monday is 0, Tuesday is 1, Wednesday is 2 and so forth.

Example: enum

```
enum WeekDays
{
    Monday,
    Tuesday,
    Wednesday,
    Thursday,
    Friday,
    Saturday,
    Sunday
}

Console.WriteLine((int)WeekDays.Monday);
Console.WriteLine((int)WeekDays.Friday);
```

An explicit cast is necessary to convert from enum type to an integral type. For example, to get the int value from an enum:

Example: enum

```
int dayNum = (int)WeekDays.Friday;

Console.WriteLine(dayNum);
```

A change in the value of the first enum member will automatically assign incremental values to the other members sequentially. For example, changing the value of Monday to 10, will assign 11 to Tuesday, 12 to Wednesday, and so on:

Example: enum

```
enum WeekDays
{
    Monday = 10,
    Tuesday,
    Wednesday,
    Thursday,
    Friday,
    Saturday,
    Sunday
}
Console.WriteLine((int)WeekDays.Monday);
Console.WriteLine((int)WeekDays.Friday);
```

Enum methods:

Enum is an abstract class that includes static helper methods to work with enums.

Enum method	Description
Format	Converts the specified value of enum type to the specified string format.
GetName	Returns the name of the constant of the specified value of specified enum.
GetNames	Returns an array of string name of all the constant of specified enum.
GetValues	Returns an array of the values of all the constants of specified enum.
object Parse(type, string)	Converts the string representation of the name or numeric value of one or more enumerated constants to an equivalent enumerated object.
bool TryParse(string, out TEnum)	Converts the string representation of the name or numeric value of one or more enumerated constants to an equivalent enumerated object. The return value indicates whether the conversion succeeded.

Example: enum methods

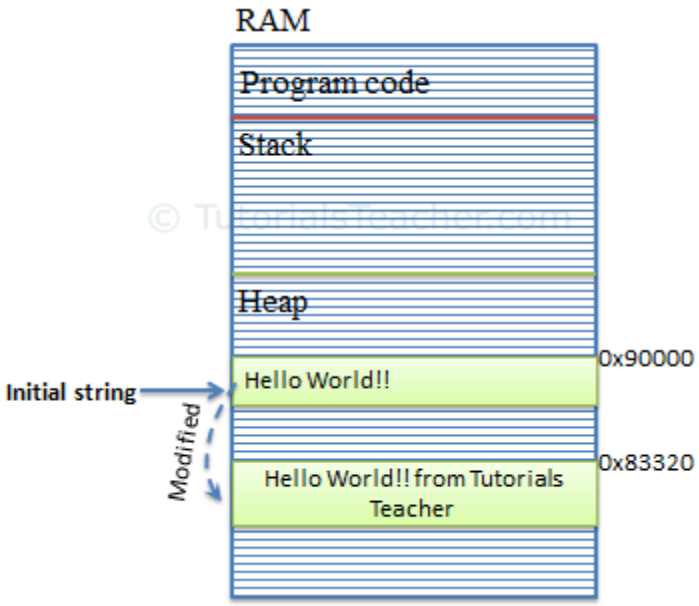
```
enum WeekDays
{
    Monday,
    Tuesday,
    Wednesday,
    Thursday,
    Friday,
    Saturday,
    Sunday
}

Console.WriteLine(Enum.GetName(typeof(WeekDays), 4));

Console.WriteLine("WeekDays constant names:");

foreach (string str in Enum.GetNames(typeof(WeekDays)))
            Console.WriteLine(str);

Console.WriteLine("Enum.TryParse():");

WeekDays wdEnum;
Enum.TryParse<WeekDays>("1", out wdEnum);
Console.WriteLine(wdEnum);
```

Output

```
Friday
WeekDays constant names:
Monday
Tuesday
Wednesday
Thursday
Friday
Saturday
Sunday
Enum.TryParse():
Tuesday
```

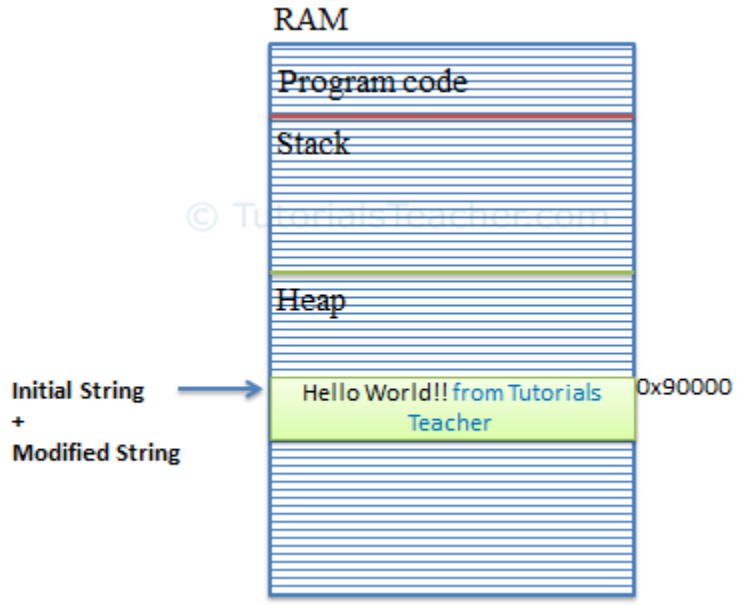
C# - StringBuilder

A String is immutable, meaning String cannot be changed once created. For example, new string "Hello World!!" will occupy a memory space on the heap. Now, by changing the initial string "Hello World!!" to "Hello World!! from Tutorials Teacher" will create a new string object on the memory heap instead of modifying the initial string at the same memory address. This behaviour will hinder the performance if the same string changes multiple times by replacing, appending, removing or inserting new strings in the initial string.



Memory allocation for String

To solve this problem, C# introduced StringBuilder. StringBuilder is a dynamic object that allows you to expand the number of characters in the string. It doesn't create a new object in the memory but dynamically expands memory to accommodate the modified string.



Memory allocation for StringBuilder

StringBuilder Initialization

StringBuilder can be initialized the same way as class.

Example: StringBuilder

```
StringBuilder sb = new StringBuilder();

//or

StringBuilder sb = new StringBuilder("Hello World!!");
```

You can give an initial capacity of characters by passing an int value in the constructor. For example, the following will allocate memory of 50 characters sequentially on the memory heap. The memory allocation automatically expands once it reaches the capacity.

Example: StringBuilder

```
StringBuilder sb = new StringBuilder(50);

//or

StringBuilder sb = new StringBuilder("Hello World!!",50);
```

Important Methods of StringBuilder

Method Name	Description
StringBuilder.Append(valueToAppend)	Appends the passed values to the end of the current StringBuilder object.
StringBuilder.AppendFormat()	Replaces a format specifier passed in a string with formatted text.
StringBuilder.Insert(index, valueToAppend)	Inserts a string at the specified index of the current StringBuilder object.
StringBuilder.Remove(int startIndex, int length)	Removes the specified number of characters from the given starting position of the current StringBuilder object.
StringBuilder.Replace(oldValue, newValue)	Replaces characters with new characters.

Append()/AppendLine()

Use Append method of StringBuilder to add or append a string to StringBuilder. AppendLine() method appends the string with a newline at the end.

Example: Append()

```
StringBuilder sb = new StringBuilder("Hello ",50);

sb.Append("World!!");
sb.AppendLine("Hello C#!");
sb.AppendLine("This is new line.");

Console.WriteLine(sb);
```

Output

```
Hello World!!Hello C#!
This is new line.
```

Array Declaration

An array can be declare using a type name followed by square brackets [].

Example: Array declaration in C#

```
int[] intArray;  // can store int values

bool[] boolArray; // can store boolean values

string[] stringArray; // can store string values

double[] doubleArray; // can store double values

byte[] byteArray; // can store byte values

Student[] customClassArray; // can store instances of Student class
```

Array Initialization

An array can be declared and initialized at the same time using the new keyword. The following example shows the way of initializing an array.

Example: Array Declaration & Initialization

```
// defining array with size 5. add values later on
int[] intArray1 = new int[5];

// defining array with size 5 and adding values at the same time
int[] intArray2 = new int[5]{1, 2, 3, 4, 5};

// defining array with 5 elements which indicates the size of an array
int[] intArray3 = {1, 2, 3, 4, 5};
```

In the above example, the first statement declares & initializes int type array that can store five int values. The size of the array is specified in square brackets. The second statement, does the same thing, but it also assignes values to each indexes in curley brackets { }. The third statement directly initializes an int array with the values without giving any size. Here, size of an array will automatically be number of values.

Initialization without giving size is **NOT** valid. For example, the following example would give compile time error.

Example: Invalid Initializing an Array

```
int[] intArray = new int[]; // compiler error: must give size of an array
```

Late Initialization

Arrays can be initialized after declaration. It is not necessary to declare and initialize at the same time using new keyword. Consider the following example.

Example: Late Initialization

```
string[] strArray1, strArray2;

strArray1 = new string[5]{ "1st Element",
                "2nd Element",
                "3rd Element",
                "4th Element",
                "5th Element"
                          };


strArray2 = new string[]{ "1st Element",
                "2nd Element",
                "3rd Element",
                "4th Element",
                "5th Element"
                          };
```

However, in the case of late initialization, it must be initialized with the new keyword as above. It cannot be initialize by only assigning values to the array.

The following initialization is **NOT** valid:

Example: Invalid Array Initializing

```
string[] strArray;

strArray = {"1st Element","2nd Element","3rd Element","4th Element" };
```

Use `foreach` loop to iterate an array.

Example: Accessing Array Elements using foreach Loop

```
int[] intArray = new int[3]{ 10, 20, 30};

foreach(var i in intArray)
    Console.WriteLine(i);
```

Array Properties and Methods

Method Name	Description
GetLength(int dimension)	Returns the number of elements in the specified dimension.
GetLowerBound(int dimension)	Returns the lowest index of the specified dimension.
GetUpperBound(int dimension)	Returns the highest index of the specified dimension.
GetValue(int index)	Returns the value at the specified index.

Property	Description
Length	Returns the total number of elements in the array.

Array Helper Class

.NET provides an abstract class, Array, as a base class for all arrays. It provides static methods for creating, manipulating, searching, and sorting arrays.

For example, use the Array.Sort() method to sort the values:

Example: Array Helper Class

```
int[] intArr = new int[5]{ 2, 4, 1, 3, 5};

Array.Sort(intArr);

Array.Reverse(intArr);
```

Points to Remember :

1. An Array stores values in a series starting with a zero-based index.
2. The size of an array must be specified while initialization.
3. An Array values can be accessed using indexer.
4. An Array can be single dimensional, multi-dimensional and jagged array.
5. The Array helper class includes utility methods for arrays.

C# - Multi-dimensional Array

We have learned about single dimensional arrays in the previous section. C# also supports multi-dimensional arrays. A multi-dimensional array is a two dimensional series like rows and columns.

Example: Multi-dimensional Array:

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

// or
int[,] intArray = { {1, 1}, {1, 2}, {1, 3} };
```

As you can see in the above example, multi dimensional array is initialized by giving size of rows and columns. [3,2] specifies that array can include 3 rows and 2 columns.

The values of a multi-dimensional array can be accessed using two indexes. The first index is for the row and the second index is for the column. Both the indexes start from zero.

Example: Access Multi-dimensional Array

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

intArray[0,0]; //Output: 1
intArray[0,1]; // 2

intArray[1,0]; // 3
intArray[1,1]; // 4

intArray[2,0]; // 5
intArray[2,1]; // 6
```

C# - Jagged Array

A jagged array is an array of an array. Jagged arrays store arrays instead of any other data type value directly.

A jagged array is initialized with two square brackets [][]. The first bracket specifies the size of an array and the second bracket specifies the dimension of the array which is going to be stored as values. (Remember, jagged array always store an array.)

The following jagged array stores a two single dimensional array as a value:

Example: Jagged Array

```
int[][] intJaggedArray = new int[2][];

intJaggedArray[0] = new int[3]{1, 2, 3};

intJaggedArray[1] = new int[2]{4, 5 };

Console.WriteLine(intJaggedArray[0][0]); // 1

Console.WriteLine(intJaggedArray[0][2]); // 3

Console.WriteLine(intJaggedArray[1][1]); // 5
```

You can also initialize a jagged array upon declaration like the below.

Example: Jagged Array

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

                             new int[2]{4, 5, 6}
                         };

Console.WriteLine(intJaggedArray[0][0]); // 1

Console.WriteLine(intJaggedArray[0][2]); // 3

Console.WriteLine(intJaggedArray[1][1]); // 5
```

The following jagged array stores a multi-dimensional array as a value. Second bracket [,] indicates multi-dimension.

Example: Jagged Array

```
int[][,] intJaggedArray = new int[3][,];

intJaggedArray[0] = new int[3, 2] { { 1, 2 }, { 3, 4 }, { 5, 6 } };
intJaggedArray[1] = new int[2, 2] { { 3, 4 }, { 5, 6 } };
intJaggedArray[2] = new int[2, 2];

Console.WriteLine(intJaggedArray[0][1,1]); // 4

Console.WriteLine(intJaggedArray[1][1,0]); // 5

Console.WriteLine(intJaggedArray[1][1,1]); // 6
```

Try it

If you add one more bracket then it will be array of array of arry.

Example: Jagged Array

```
int[][][] intJaggedArray = new int[2][][]
                           {
                               new int[2][]
                               {
```

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

Console.WriteLine(intJaggedArray[0][0][0]); // 1

Console.WriteLine(intJaggedArray[0][1][1]); // 5

Console.WriteLine(intJaggedArray[1][0][2]); // 9
```

In the above example of jagged array, three bracket [][][] means array of array of array. So, intJaggedArray will contain 2 elements, means 2 arrays. Now, each of these arrays also contains array (single dimension). `intJaggedArray[0][0][0]` points to the first element of first inner array of `intJaggedArray`. `intJaggedArray[1][0][2]` points to the third element of second inner array of `intJaggedArray`. The following figure illustrates this.