Syllogistek
Systems Private Limited

502 SreeNilayam, Lanco Hills Road,
Manikonda, Hyderabad – 500089
08413485772 (landline) / 9701377070
Email : hr@syllogistek.com

Contents

Day 1: Introduction of .Net Framework

1. The .NET Framework is set of technologies that form an integral part of the .NET Platform.
2. It is Microsoft's managed code programming model for building applications that have visually stunning user experiences, seamless and secure communication, and the ability to model a range of business processes.
3. The .NET Framework has two main components:
    a. Common language runtime (CLR)
    b. .NET Framework class library.
4. The CLR is the foundation of the .NET framework and provides a common set of services for projects that act as building blocks to build up applications across all tiers. It simplifies development and provides a robust and simplified environment which provides common services to build application.
5. The .NET framework class library is a collection of reusable types and exposes features of the runtime. It contains of a set of classes that is used to access common functionality.

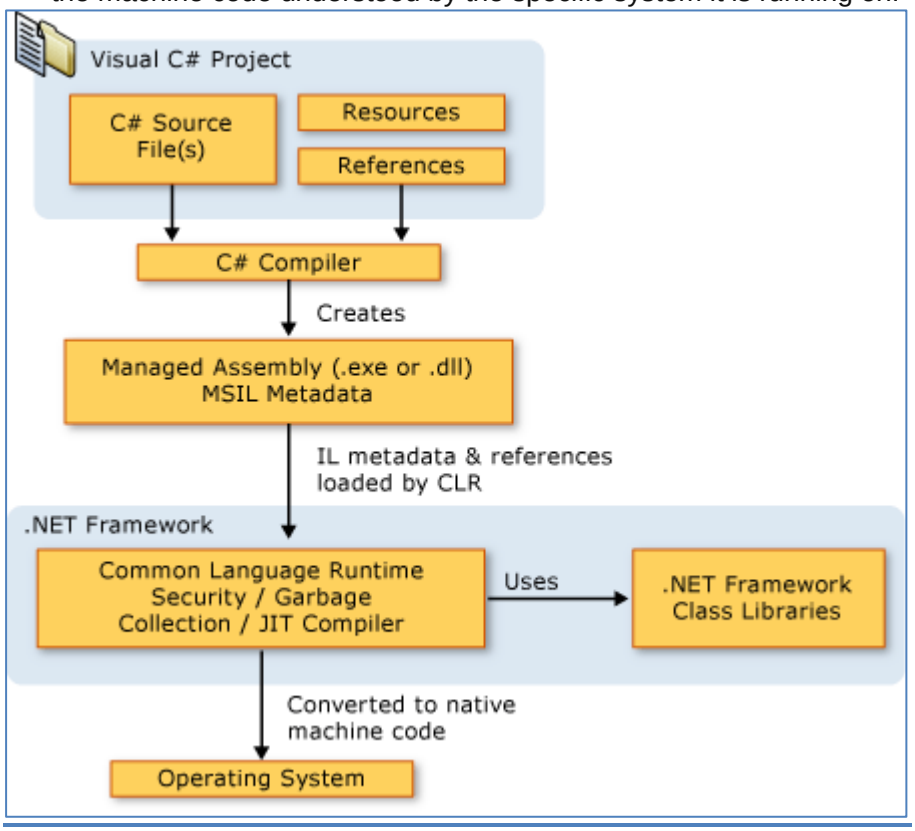
Generations of .NET Framework

Framework Version	CLR Version	Release Date	Support ended	Development tool
1	1	13-02-2002	14-07-2009	Visual Studio .NET
1.1	1.1	24-04-2003	14-06-2015	Visual Studio .NET 2003
2	2	07-11-2005	12-07-2011	Visual Studio 2005
3	2	06-11-2006	12-07-2011	Expression Blend
3.5	2	19-11-2007	10-10-2028	Visual Studio 2008
4	4	12-04-2010	12-01-2016	Visual Studio 2010
4.5	4	15-08-2012	12-01-2016	Visual Studio 2012
4.05.01	4	17-10-2013	12-01-2016	Visual Studio 2013
4.05.02	4	05-05-2014	NA	N/A
4.6	4	20-07-2015	NA	Visual Studio 2015
4.06.01	4	30-11-2015	NA	Visual Studio 2015 Update 1
4.06.02	4	02-08-2016	NA	
4.7	**4**	**05-04-2017**	**NA**	**Visual Studio 2017**
4.07.01	**4**	**17-10-2017**	**NA**	**Visual Studio 2017**
4.07.02	**4**	**30-04-2018**	**NA**	**Visual Studio 2017**

4.8	4	18-04-2019	NA	Visual Studio 2019

Common Language Infrastructure (CLI)

1. Common Language Infrastructure (CLI) enables an application program written in any of several commonly-used programming languages to be run on any operating system using a common runtime program rather than a language-specification.
2. Common Language Infrastructure provides a virtual execution environment comparable to the one provided by Sun Microsystems for Java programs.
3. Common Language Infrastructure has been accepted as an open standard by ECMA, an international organization for the promotion of technology standards.
4. CLI is designed to make it easier to develop programs that can be used as Web services.
5. Any programming language that conforms to CLI criteria can access a common class library and can be compiled into a common intermediate language (byte code).
6. Once compiled into the common intermediate language, any program can then be further compiled into the machine code understood by the specific system it is running on.



CLR

1. The .NET Framework provides a runtime environment called the Common Language Runtime or CLR.
2. The CLR can be compared to the Java Virtual Machine or JVM in Java. CLR is the heart of .Net.
3. CLR handles the execution of code and provides useful services for the implementation of the program.
4. In addition to executing code, CLR provides services such as memory management, thread management, security management, code verification, compilation, and other system services.
5. It enforces rules that in turn provide a robust and secure execution environment for .NET applications.

CTS

1. Common Type System (CTS) describes the data types that can be used by managed code.
2. CTS define how these types are declared, used and managed in the runtime.

3. It facilitates cross-language integration, type safety and high performance code execution.
4. The rules defined in CTS can be used to define your own classes and values.

   *(Example: C# has **int** Data Type and VB.Net has **Integer** Data Type. Hence a variable declared as **int** in C# or **Integer** in vb.net, finally after compilation, uses the same structure **Int32** from CTS.)*

CLS

1. CLS stands for Common Language Specification and it is a subset of CTS.
2. It defines a set of rules and restrictions that every language must follow which runs under .NET framework.
3. The languages which follow these set of rules are said to be CLS Compliant. In simple words, CLS enables cross-language integration.
   (For Details: https://en.wikipedia.org/wiki/List_of_CLI_languages)

   *(Example: one rule is that you cannot use **multiple inheritances** within .NET Framework. As we know C++ supports multiple inheritances but; when you will try to use that C++ code within C#, it is not possible because C# doesn't support multiple inheritances.)*

   *(Example: One another rule is that you cannot have members with same name with case difference only i.e. you cannot have **add()** and **Add()** methods. This easily works in C# because it is case-sensitive but when you will try to use that C# code in VB.NET, it is not possible because VB.NET is not case-sensitive.)*

Managed Code

1. The .NET Framework provides a run-time environment called the Common Language Runtime, which manages the execution of code and provides services that make the development process easier.
2. Compilers and tools expose the runtime's functionality and enable you to write code that benefits from this managed execution environment.
3. The code that runs within the common language runtime is called managed code.

MSIL

1. When the code is compiled, the compiler translates your code into Microsoft intermediate language (MSIL).
2. The common language runtime includes a JIT compiler for converting this MSIL then to native code.
3. MSIL contains metadata that is the key to cross language interoperability. Since this metadata is standardized across all .NET languages, a program written in one language can understand the metadata and execute code, written in a different language.
4. MSIL includes instructions for loading, storing, initializing, and calling methods on objects, as well as instructions for arithmetic and logical operations, control flow, direct memory access, exception handling, and other operations.

JIT

1. JIT is a compiler that converts MSIL to native code. The native code consists of hardware specific instructions that can be executed by the CPU.
2. Rather than converting the entire MSIL (in a portable executable [PE] file) to native code, the JIT converts the MSIL as it is needed during execution. This converted native code is stored so that it is accessible for subsequent calls.

Day 1: Assembly

1. An assembly is a collection of one or more .exe or dll's.
2. An assembly is the fundamental unit for application development and deployment in the .NET Framework.
3. An assembly contains a collection of types and resources that are built to work together and form a logical unit of functionality.
4. An assembly provides the CLR with the information it needs to be aware of type implementations.

Contents of assembly

1. A static assembly can consist of four elements:
    I. Assembly manifest - Contains the assembly metadata. An assembly manifest contains the information about the identity and version of the assembly. It also contains the information required to resolve references to types and resources.
    II. Type metadata - Binary information that describes a program.
    III. Microsoft intermediate language (MSIL) code.
    IV. A set of resources.

Types of assembly

1. Assemblies can also be private or shared.
2. A private assembly is installed in the installation directory of an application and is accessible to that application only.
3. A shared assembly is shared by multiple applications. A shared assembly has a strong name and is installed in the GAC.
4. A satellite assembly that are often used to deploy language-specific resources for an application.

Day 1: C# introduction

1. C# syntax is highly expressive, yet it is also simple and **easy to learn**.
2. The **curly-brace syntax** of C# will be instantly recognizable to anyone familiar with C, C++ or Java.
3. Developers who know any of these languages are typically able to begin to work productively in C# within a very **short time**.
4. C# syntax simplifies many of the complexities of C++ and provides powerful features such as **nullable** value types, **enumerations**, **delegates**, **lambda expressions** and **direct memory access**, which are not found in Java.
5. C# supports **generic methods** and types, which provide increased type safety and performance, and iterators, which enable implementers of collection classes to define custom iteration behaviors that are simple to use by client code.
6. Language-Integrated Query (**LINQ**) expressions make the strongly-typed query a first-class language construct.
7. As an **object-oriented language**, C# supports the concepts of **encapsulation**, **inheritance**, **abstraction** and **polymorphism**.
8. All variables and methods, including the Main method, the **application's entry point, are encapsulated** within class definitions.
9. A class may inherit directly from one parent class, but it may **implement** any number of **interfaces**.
10. Methods that override virtual methods in a parent class require the **override** keyword as a way to avoid accidental redefinition. In C#, a struct is like a lightweight class; it is a stack-allocated type that can implement interfaces but does not support inheritance.

External Links:

https://docs.microsoft.com/en-us/dotnet/csharp/getting-started/introduction-to-the-csharp-language-and-the-net-framework

Introduction of Object Oriented Programming (OOP)

1. **OOP** is a design philosophy. It stands for **Object Oriented Programming**.
2. **OOP** uses a different set of programming languages than old procedural programming languages (C, Pascal, etc.).
3. Everything in OOP is grouped as self-sustainable "**objects**".
4. Hence, we gain reusability by means of four main object-oriented programming concepts.

In order to clearly understand the object orientation model, let's take humanoid robot "**hand**" as an example.
The "**hand**" is a class.
Our body has two **objects** of the type "hand", named "**lefthand**" and "**right hand**".
Their main functions are controlled or managed by a set of electrical signals sent through our **shoulders** (through an interface).
So the shoulder is an **interface** that your body uses to interact with our hands.
The hand is a well-architected class.
The hand is being reused to create the left hand and the right hand by slightly changing the properties of it.

Object:

1. An object can be considered a "thing" that can perform a set of related activities.
2. The set of activities that the object performs defines the object's behavior. For example, the Hand (object) can grip something, or a Student (object) can give their name or address.
3. In pure OOP terms an **object is an instance of a class**

Class:

A *class* is simply a representation of a type of ***object***. It is the blueprint, or plan, or template that describes the details of an ***object***. A class is the blueprint from which the individual **object**s are created. *Class* is composed of three things: a name, attributes, and operations.

```
public class Student
{
}
```

According to the sample given below we can say that the **Studentobject**, named **objectStudent**, has been created out of the **Student** class.

```
Student objectStudent = new Student();
```

External Link:
https://www.codeproject.com/Articles/22769/Introduction-to-Object-Oriented-Programming-Concep

Hello World Program

```
using System;
namespace Day1
{
classHelloWorld
  {
     public static void Main()
    {
Console.WriteLine("Hello, world!");
    }
  }
}
```

Understanding of Visual Studio:

1. Visual Studio is a complete set of development tools for building high performance Desktop applications, XML Web Services, mobile applications and ASP .Net web applications.
2. Visual Studio provides some ready-made tools for building .NET applications, such as:

I. Forms designer
II. Server Explorer
III. Property Window
IV. Tool-Box
V. Solution Explorer
VI. Compiler
VII. Debuggers
VIII. Output Window

The .NET Framework is multi-lingual, in other words it supports many types of languages.
e.g. C#, VB .Net, VC++ etc.

The .NET framework provides a multi-platform environment. It in other words we need to create: Web-Based, Windows-based, Console-Based, Web-Services etc.

External Link:
https://www.c-sharpcorner.com/UploadFile/d0a1c8/introduction-to-C-Sharp-net/

Day 2: 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:

- Value type
- Reference type

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 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 data types are of reference type:

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

Boxing

Boxing is the conversion of any value type to object type. Remember that boxing is an implicit conversion. Boxing a value of value type like int consists of allocating an object instance and copying the value of the value type into that object instance.
An example of boxing is shown below.

```
int x = 10;
object o = x; //Boxing
if(o is int)
Console.WriteLine("o contains int type");
```

Un-boxing

Un-boxing is the conversion of an object type to a value type. The casting operator () is necessary for unboxing. The example of unboxing is shown below.

```
int x = 100;
object o = x; // Boxing
int I (int) o; // un-boxing
```

if-else Condition

C# also provides for a second part to the if statement, that is else. The else statement must follow if or else if statement. Also, else statement can appear only one time in a if-else statement chain.

```
staticvoid Main(string[] args)
{
inti = 10, j = 20;

if (i> j)
   {
Console.WriteLine("i is greater than j");
   }
elseif (i< j)
   {
Console.WriteLine("i is less than j");
   }
else
   {
Console.WriteLine("i is equal to j");
   }
}
```

Output: i is less than j

Link: https://www.tutorialsteacher.com/csharp/csharp-if-else

Ternary operator (?):

C# includes a special type of decision making operator '?:' called the ternary operator.

Boolean Expression ? First Statement : Second Statement

As you can see in the above syntax, ternary operator includes three parts. First part (before ?) includes conditional expression that returns boolean value true or false. Second part (after ? and before :) contains a statement which will be returned if the conditional expression in the first part evalutes to true. The third part includes another statement which will be returned if the conditional expression returns false.

Example:

```
int x = 20, y = 10;
var result = x > y ? "x is greater than y" : "x is less than or equal to y";
Console.WriteLine(result);
```

Output: x is greater than y

switch

C# includes another decision making statement called switch. The switch statement executes the code block depending upon the resulted value of an expression.

```
switch(expression)
{
case<value1>
      // code block
break;
case<value2>
      // code block
break;
case<valueN>
      // code block
break;
default
      // code block
```

```
break;
}
```

As per the syntax above, switch statement contains an expression into brackets. It also includes multiple case labels, where each case represents a particular literal value. The switch cases are separated by a break keyword which stops the execution of a particular case. Also, the switch can include a default case to execute if no case value satisfies the expression.

```
int x = 10;

switch (x)
{
case 5:
Console.WriteLine("Value of x is 5");
break;
case 10:
Console.WriteLine("Value of x is 10");
break;
case 15:
Console.WriteLine("Value of x is 15");
break;
default:
Console.WriteLine("Unknown value");
break;
}
```

Output: Value of x is 10

for loop

The **for** keyword indicates a loop in C#. The for loop executes a block of statements repeatedly until the specified condition returns false.

```
for (variable initialization; condition; steps)
{
    //execute this code block as long as condition is satisfied
}
```

As per the syntax above, the for loop contains three parts: initialization, conditional expression and steps, which are separated by a semicolon.

1. Variable initialization: Declare & initialize a variable here which will be used in conditional expression and steps part.
2. Condition: The condition is a boolean expression which will return either true or false.
3. steps: The steps defines the incremental or decremental part

Consider the following example of a simple for loop.

```
for (inti = 0; i< 10; i++)
{
Console.WriteLine("Value of i: {0}", i);
}
```

Output:
Value of i: 0
Value of i: 1
Value of i: 2
Value of i: 3
Value of i: 4
Value of i: 5
Value of i: 6
Value of i: 7
Value of i: 8
Value of i: 9

Link:https://www.tutorialsteacher.com/csharp/csharp-for-loop

do while& while loop

The do-while loop is the same as a 'while' loop except that the block of code will be executed at least once, because it first executes the block of code and then it checks the condition.

```
do
{
   //execute code block

} while(boolean expression);
```

As per the syntax above, do-while loop starts with the 'do' keyword followed by a code block and boolean expression with 'while'.
Example:

```
inti = 0;

do
{
Console.WriteLine("Value of i: {0}", i);

i++;

} while (i< 10);
```

Output:
Value of i: 0
Value of i: 1
Value of i: 2
Value of i: 3
Value of i: 4
Value of i: 5
Value of i: 6
Value of i: 7
Value of i: 8
Value of i: 9

Link:https://www.tutorialsteacher.com/csharp/csharp-do-while-loop

Day 2: Exception Handling

We have seen in the previous section that an exception is thrown by the CLR or program code if there is an error in the program. These exceptions need to be handling to prevent crashing of program. C# provides built-in support to handle the exception using try, catch & finally block.

```
try
{
   // code that may raise exceptions
}
catch(Exception ex)
{
   // handle exception
}
finally
{
   // final cleanup code
}
```

Throw Exception

We need to remember an exception is not an error, An exception is a situation where there is possibility of uncertain situation.

For example suppose the following function returning name of an employee, and it accepts employee email as an input parameter.

```
public string GetStudentNameByEmail(string email) {}
```

It can be assumed inside the function it will search some permanent storage like Database, but before the actual database calling there is possibility that the parameter “email” is null or blank string,
It also assumed that with null or empty string the actual database will not provide any result, but in general to avoid unexpected situation developer can handle the exception at the beginning.

```
public string GetStudentNameByEmail (string email) {

        if(string.IsNullOrEmpty(email){
                throw new ArgumentNullException(“email”,”email is required”);
        }

        // Rest of the logic
}
```

Day 2: Namespaces

Namespaces in C# are used to organize too many classes so that it can be easy to handle the application.

In a simple C# program, we use **System.Console** where System is the namespace and Console is the class. To access the class of a namespace, we need to use **namespacename.classname**. We can use using keyword so that we don't have to use complete name all the time.

In C#, global namespace is the root namespace. The global::System will always refer to the namespace "System" of .Net Framework.

System Namespace

The System namespace contains fundamental classes and base classes that define commonly-used value and reference data types, events and event handlers, interfaces, attributes, and processing exceptions.

```
using System;
namespace ConsoleApplication1
{
class Program
   {
static void Main(string[] args)
      {
Console.WriteLine("Hello Namespace!");
```

```
    }
  }
}
```

Day 2: String

Strings are one of the most important data types in any modern language including C#.

The Char data type represents a character in .NET. In .NET, the text is stored as a sequential read-only collection of Char data types. There is no null-terminating character at the end of a C# string; therefore a C# string can contain any number of embedded null characters ('\0').

The System.String data type represents a string in .NET. A string class in C# is an object of type System.String. The String class in C# represents a string.

The following code creates three strings with a name, number, and double values.

```
// String of characters
System.String authorName = "Arghya Biswas";
// String made of an Integer
System.String age = "33";
// String made of a double
System.String numberString = "33.23";
```

Here is the complete example that shows how to use stings in C# and .NET.

```
using System;
namespace CSharpStrings
{
  class Program
  {
    static void Main(string[] args)
    {
      // Define .NET Strings
      // String of characters
      System.String authorName = "Arghya Biswas";
      // String made of an Integer
      System.String age = "33";
      // String made of a double
      System.String numberString = "33.23";
      // Write to Console.
      Console.WriteLine("Name: {0}", authorName);
      Console.WriteLine("Age: {0}", age);
      Console.WriteLine("Number: {0}", numberString);
      Console.ReadKey();
    }
  }
}
```

String class defined in the .NET base class library represents text as a series of Unicode characters. The String class provides methods and properties to work with strings.

The String class has methods to clone a string, compare strings, concatenate strings, and copy strings. This class also provides methods to find a substring in a string, find the index of a character or substring, replace characters, split a string, trim a string, and add padding to a string. The string class also provides methods to convert a string's characters to uppercase or lowercase.

Check out these links to learn about a specific operation or functionality of strings.

What is different between string and System.String?

.NET defines all data types as a class. The System.String class represents a collection of Unicode characters also known as a text. The System.String class also defines the properties and methods to work with string data types.

The string class is equivalent to the System.String in C# language. The string class also inherits all the properties and methods of the System.String class.

Create a string

There are several ways to construct strings in C# and .NET.

- Create a string using a constructor
- Create a string from a literal
- Create a string using concatenation
- Create a string using a property or a method
- Create a string using formatting

Create a string using its constructor

The String class has several overloaded constructors that take an array of characters or bytes. The following code snippet creates a string from an array of characters.

```
char[] chars = { 'M', 'a', 'h', 'e', 's', 'h' };
string name = new string(chars);
Console.WriteLine(name);
```

Create a string from a literal

This is the most common ways to instantiate a string.

You simply define a string type variable and assign a text value to the variable by placing the text value without double quotes. You can put almost any type of characters within double quotes accept some special character limitations.

The following code snippet defines a string variable named firstName and then assigns text value Arghya to it.

```
string firstName;
firstName = "Arghya";
```

Alternatively, we can assign the text value directly to the variable.

```
string firstName = "Arghya";
```

Here is a complete sample example of how to create strings using literals.

```
using System;
namespace CSharpStrings
{
    class Program
    {
        static void Main(string[] args)
        {
            string firstName = "Arghya";
            string lastName = "Biswas";
            string age = "33";
            string numberString = "33.23";
            Console.WriteLine("First Name: {0}", firstName);
            Console.WriteLine("Last Name: {0}", lastName);
            Console.WriteLine("Age: {0}", age);
            Console.WriteLine("Number: {0}", numberString);
            Console.ReadKey();
        }
    }
}
```

Create a string using concatenation

Sting concatenation operator (+) can be used to combine more than one string to create a single string. The following code snippet creates two strings. The first string adds a text Date and current date value from the DateTime object. The second string adds three strings and some hard coded text to create a larger string.

```
string nowDateTime = "Date: " + DateTime.Now.ToString("D");
string firstName = "Arghya";
string lastName = "Biswas";
string age = "33";
string authorDetails = firstName + " " + lastName + " is " + age + " years old.";
Console.WriteLine(nowDateTime);
Console.WriteLine(authorDetails);
```

Create a string using a property or a method

Some properties and methods of the String class returns a string object such as SubString method. The following code snippet takes one sentence string and finds the age within that string. The code returns 33.

```
string authorInfo = "Arghya Biswas is 33 years old.";
int startPosition = sentence.IndexOf("is ") + 1;
string age = authorInfo.Substring(startPosition +2, 2 );
Console.WriteLine("Age: " + age);
```

Create a string with Format

The String.Format method returns a string. The following code snippet creates a new string using the Format method.

```
string name = "Arghya Biswas";
int age = 33;
string authorInfo = string.Format("{0} is {1} years old.", name, age.ToString());
Console.WriteLine(authorInfo);
```

Create a string using ToString Method

The ToString method returns a string. We can apply ToString on pretty much any data type that can be converted to a string. The following code snippet converts an int data type to a string.

```
string name = "Arghya Biswas";
int age = 33;
string authorInfo = string.Format("{0} is {1} years old.", name, age.ToString());
Console.WriteLine(authorInfo);
```

Get all characters of a string using C#

A string is a collection of characters.
The following code snippet reads all characters of a string and displays on the console.

```
string nameString = "Arghya Biswas";
for (int counter = 0; counter <= nameString.Length - 1; counter++)
Console.WriteLine(nameString[counter]);
```

Size of string

The Length property of the string class returns the number of characters in a string including white spaces.

The following code snippet returns the size of a string and displays on the console.

```
string nameString = "Arghya Biswas";
Console.WriteLine(nameString);
Console.WriteLine("Size of string {0}", nameString.Length);
```

Number of characters in a string

We can use the string.Length property to get the number of characters of a string but it will also count an empty character. So, to find out the exact number of characters in a string, we need to remove the empty character occurrences from a string.

The following code snippet uses the Replace method to remove empty characters and then displays the non-empty characters of a string.

```
string name = "Arghya Biswas";
// Get size of string
Console.WriteLine("Size of string: {0}", name.Length );
// Remove all empty characters
string nameWithoutEmptyChar = name.Replace(" ", "");
// Size after empty characters are removed
Console.WriteLine("Size of non empty char string: {0}", nameWithoutEmptyChar.Length);
// Read and print all characters
for (int counter = 0; counter <= nameWithoutEmptyChar.Length - 1; counter++)
Console.WriteLine(nameWithoutEmptyChar[counter]);
```

Convert String to Char Array

ToCharArray method converts a string to an array of Unicode characters. The following code snippet converts a string to char array and displays them.

```
string sentence = "Arghya Biswas is an author of this document";
char[] charArr = sentence.ToCharArray();
foreach (char ch in charArr)
{
  Console.WriteLine(ch);
}
```

Empty String

An empty string is a valid instance of a System.String object that contains zero characters. There are two ways to create an empty string. We can either use the string.Empty property or we can simply assign a text value with no text in it.

The following code snippet creates two empty strings.

```
string empStr = string.Empty;
string empStr2 = "";
```

Both of the statements above generate the same output.

An empty string is sometimes used to compare the value of other strings. The following code snippet uses an empty string to compare with the name string.

```
string name = "Arghya Biswas";
if (name != empStr)
{
  Console.WriteLine(name);
}
```

In real-world coding, we will probably never create an empty string unless you plan to use it somewhere else as a non-empty string. We can simply use the string.Empty direct to compare a string with an empty string.

```
if (name != string.Empty)
{
  Console.WriteLine(name);
}
```

Here is a complete example of using an empty string.

```
string empStr = string.Empty;
string empStr2 = "";
string name = "Arghya Biswas";
if (name != empStr)
{
  Console.WriteLine(name);
}
if (name != string.Empty)
{
  Console.WriteLine(name);
}
```

Null String

A null string is a string variable that has not been initialized yet and has a null value. If you try to call any methods or properties of a null string, you will get an exception. A null string valuable is exactly the same as any other variable defined in your code.

A null string is typically used in string concatenation and comparison operations with other strings.

The following code example shows how to use a null string.

```
string nullStr = null;
string empStr = string.Empty;
string name = "Arghya Biswas";
if ((name != nullStr) || (name != empStr))
{
  Console.WriteLine(name + " is neither null nor empty");
}
```

String interpolation

The $ special character identifies a string literal as an *interpolated string*. An interpolated string is a string literal that might contain *interpolation expressions*. When an interpolated string is resolved to a result string, items with interpolation expressions are replaced by the string representations of the expression results. This feature is available in C# 6 and later versions of the language.

String interpolation provides a more readable and convenient syntax to create formatted strings than a string composite formatting feature. The following example uses both features to produce the same output:

```
string name = "Mark";
var date = DateTime.Now;

// Composite formatting:
Console.WriteLine("Hello, {0}! Today is {1}, it's {2:HH:mm} now.", name, date.DayOfWeek, date);
// String interpolation:
Console.WriteLine($"Hello, {name}! Today is {date.DayOfWeek}, it's {date:HH:mm} now.");
// Both calls produce the same output that is similar to:
// Hello, Mark! Today is Wednesday, it's 19:40 now.
```

To include a brace, "{" or "}", in the text produced by an interpolated string, use two braces, "{{" or "}}".

As the colon (":") has special meaning in an interpolation expression item, in order to use a conditional operator in an interpolation expression, enclose that expression in parentheses.

The following example shows how to include a brace in a result string and how to use a conditional operator in an interpolation expression:

```
string name = "Horace";
int age = 34;
Console.WriteLine($"He asked, \"Is your name {name}?\", but didn't wait for a reply :-{{");
Console.WriteLine($"{name} is {age} year{(age == 1 ? "" : "s")} old.");
```

Output:

```
// He asked, "Is your name Horace?", but didn't wait for a reply :-{
// Horace is 34 years old.
```

Verbatim identifier

The @ special character serves as a verbatim identifier.

@ character to define an identifier named for that it uses in a for loop.

```
string[] @for = { "John", "James", "Joan", "Jamie" };
for (intctr = 0; ctr< @for.Length; ctr++)
{
Console.WriteLine($"Here is your gift, {@for[ctr]}!");
}
```

Output:

```
//     Here is your gift, John!
//     Here is your gift, James!
//     Here is your gift, Joan!
//     Here is your gift, Jamie!
```

Simple escape sequences (such as "\\" for a backslash),

```
string filename1 = @"c:\documents\files\u0066.txt";
```

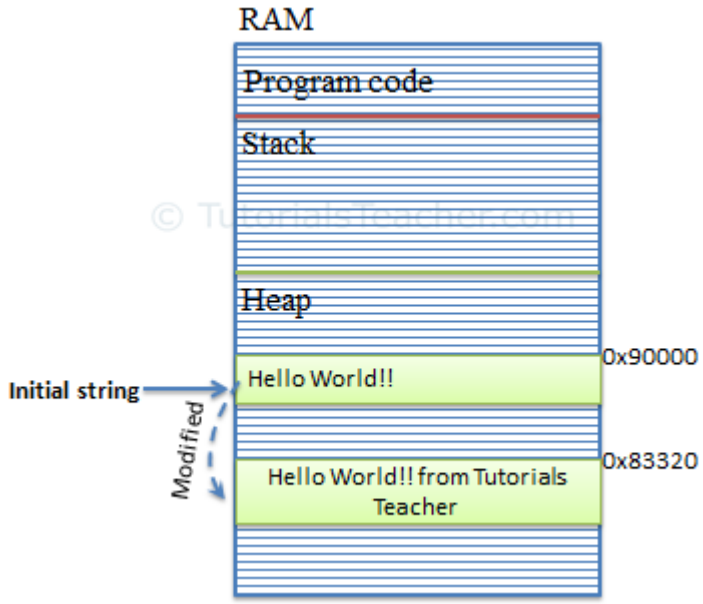
```
string filename2 = "c:\\documents\\files\\u0066.txt";

Console.WriteLine(filename1);
Console.WriteLine(filename2);
```

Output :

```
//     c:\documents\files\u0066.txt
//     c:\documents\files\u0066.txt
```

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.



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.

```
StringBuildersb = newStringBuilder();

//or

StringBuildersb = newStringBuilder("Hello World!!");
```

```
StringBuildersb = newStringBuilder("Hello ",50);

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

Console.WriteLine(sb);
```