आईएफटीएम विश्वविद्यालय, मुरादाबाद, उत्तर प्रदेश

IFTM University, Moradabad, Uttar Pradesh

NAAC ACCREDITED

E-Content

IFTM University, Moradabad

Unit-1
.Net Framework using C#

.Net Framework

.NET is a software framework which is designed and developed by **Microsoft**. This Framework is used for building and running applications on Windows. The first version of the .Net framework was 1.0 which came in the year 2002. It supports running websites, services, desktop apps, and more on Windows. It is used to develop **Form-based** applications, **Web-based** applications, and **Web services**. There is a variety of programming languages available on the .Net platform.

The .Net framework consists of developer tools, programming languages, and libraries to build desktop and web applications. It is also used to build websites, web services, and games.

In other words we can say, it is a virtual machine for compiling and executing programs written in different languages like C#, VB.Net etc. It provides a lot of functionalities and also supports industry standards.

.NET Framework supports more than 60 programming languages in which 11 programming languages are designed and developed by Microsoft.

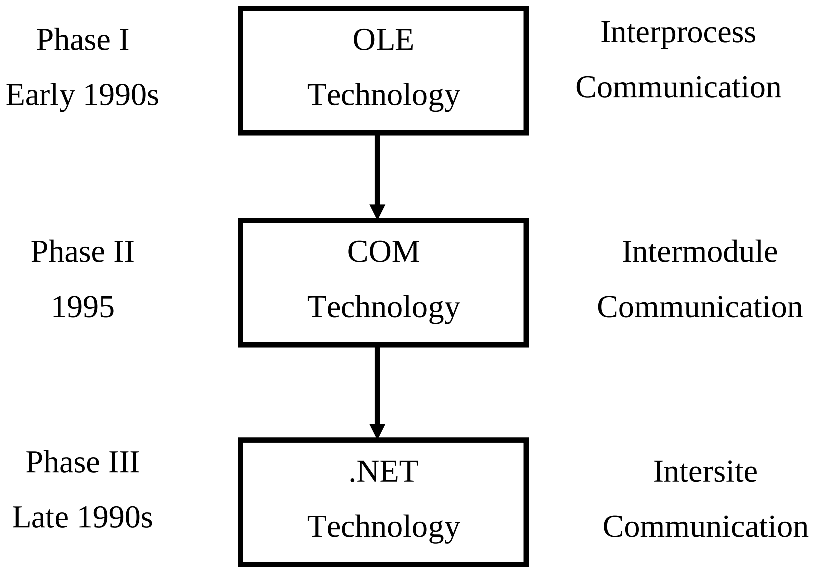
Programming Languages which are designed and developed by Microsoft are:

1. C#.NET
2. VB.NET
3. C++.NET
4. J#.NET
5. F#.NET
6. JSCRIPT.NET
7. WINDOWS POWERSHELL
8. IRON RUBY
9. IRON PYTHON
10. C OMEGA
11. ASML(Abstract State Machine Language)

Origin of .net Technology

The current technology of .NET has gone through three significant phases of developments:

1. OLE Technology
2. COM Technology
3. .NET Technology



OLE Technology (Object Linking and Embedding)

Object linking and embedding (OLE) is a Microsoft technology that facilitates the sharing of application data and objects written in different formats from multiple sources. **Linking** establishes a connection between two objects, and **embedding** facilitates application data insertion. OLE is used for compound document management, as well as application data transfer via drag-and-drop and clipboard operations.

OLE provides support to achieve the following:

– Easy inter-process communication

– Embed documents from one application into another application

– To enable one application to manipulate objects located in another application

– Ex: inter operability between various products such as MS word and MS-Excel

COM Technology (Component Object Model)

Unlike monolithic approach, in the component-based approach, a program is broken into a number of independent components where each one offers a particular service. Each component can be developed and tested independently and then integrated it into the main system.

COM is a simple Microsoft specification method that defines a binary standard for exchanging code between two systems, regardless of the OS or programming language. COM is used by developers to create re-usable software components, link components together to build applications, and take advantage of Windows services.

– Monolithic approach leads to many problems of maintainability and testing

– A program is broken into number of independent components where each one offers a particular service

– Each component can be developed and tested independently and then integrated into main system.

COM technology offers a number of benefits as:

– Reduces the overall complexity of software.

– Enables distributed development across multiple organization or departments.

– Enhances software maintainability

.NET Technology

.NET technology is a third generation component model. This provides a new level of inter-operability compared to COM technology. COM provides a standard binary mechanism for inter-module communication. This mechanism is replaced by an intermediate language called Microsoft Intermediate Language (MSIL) or simply IL in the .NET technology. Various .NET-language compilers enforce inter-operability by compiling code into IL, which is automatically compatible with other IL modules. All these functionality enables us to develop and implement Web-based applications easily.

.NET technology facilities:

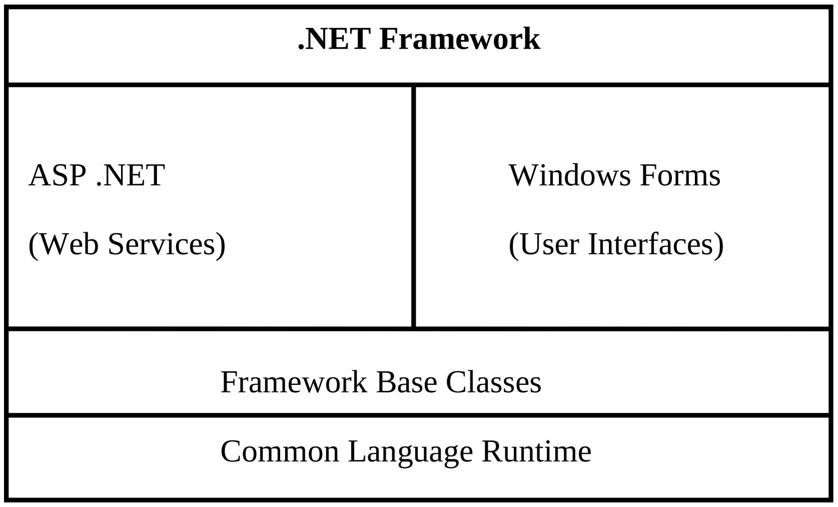
– Replacement of IMC (Inter Module Communication) in COM by Intermediate Language (IL or MSIL or CIL)

– Interoperability by compiling code into IL.

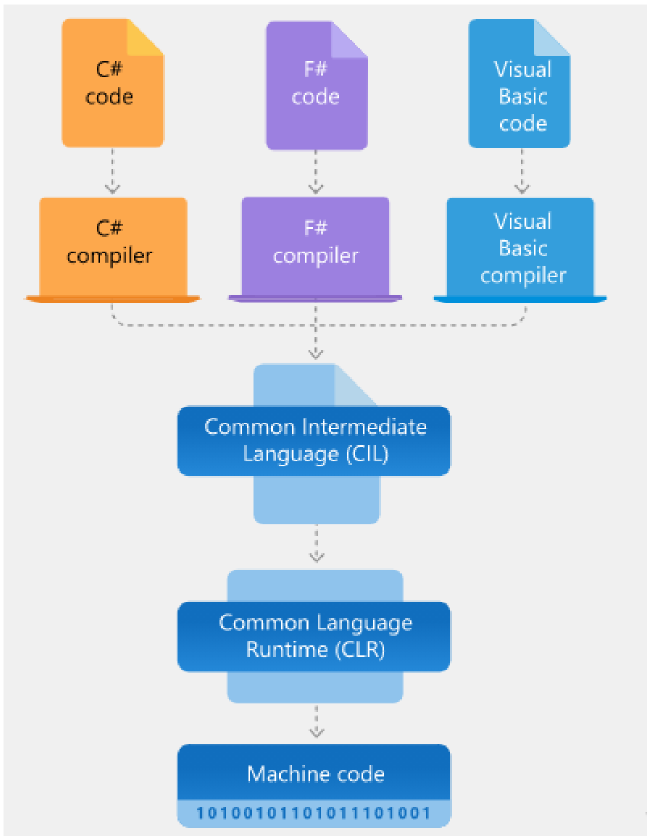
– Metadata (data about data)

.Net Framework Architecture

The basic architecture of the .Net framework is as shown in the figure below. The architecture of .Net framework is based on the following key components.



.NET Framework Architecture



1. **<u>CLR (Common Language Runtime)</u>:**-The Common Language Runtime, popularly known as CLR is the heart and soul of the .net Framework. It is a run-time environment which executes the code written in any .NET programming language. Basically, it is responsible for loading and running of *.NET programs*.

   It is .NET equivalent of Java Virtual Machine (JVM). It is the runtime that converts a MSIL (Micro Soft Intermediate Language) code into the host machine language code, which is then executed appropriately.

   The CLR provides a number of services that include:

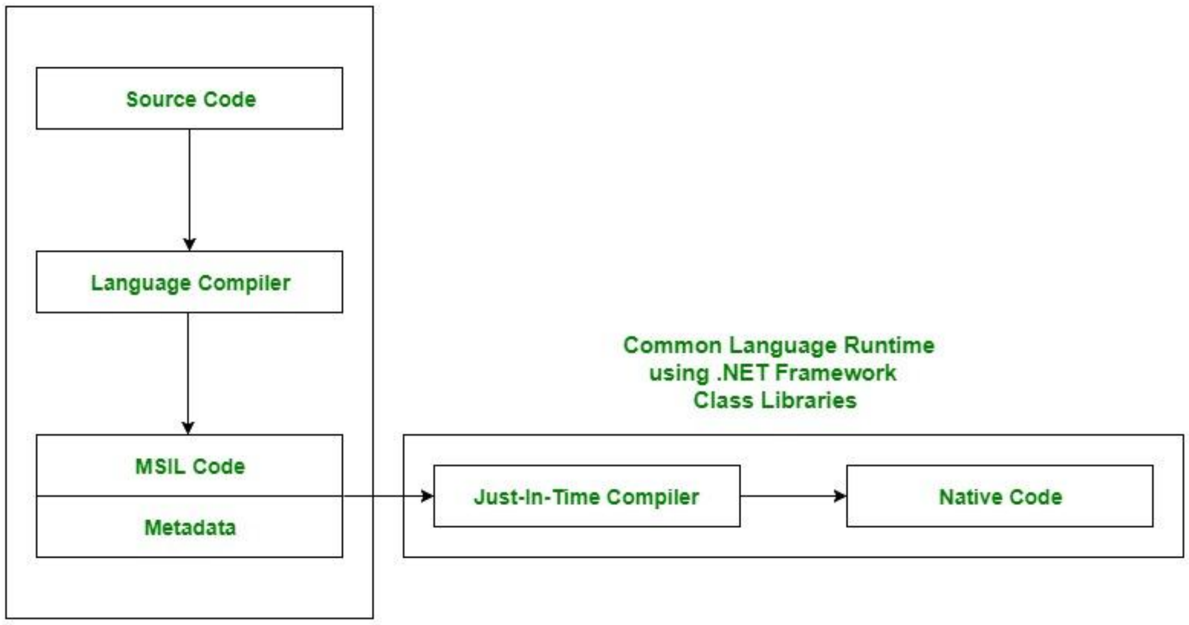
   - Loading and execution of codes
   - Memory isolation for application
   - Verification of type safety
   - Compilation of IL into native executable code
   - Providing metadata
   - Automatic garbage collection
   - Enforcement of Security
   - Interoperability with other systems
   - Managing exceptions and errors
   - Provide support for debugging and profiling

   <u>Role of CLR in the execution of a C# program</u>

- Suppose we have written a C# program and save it in a file which is known as the Source Code.

- Language specific compiler compiles the source code into the **MSIL (Microsoft Intermediate Language)** which is also known as the **CIL (Common Intermediate Language)** or **IL (Intermediate Language)** along with its metadata. **Metadata** includes the all the types, actual implementation of each function of the program. MSIL is machine independent code.
- Now CLR comes into existence. CLR provides the services and runtime environment to the MSIL code. Internally CLR includes the **JIT (Just-In-Time)** compiler which converts the MSIL code to machine code which further executed by CPU. CLR also uses the .NET Framework class libraries. Metadata provides information about the programming language, environment, version, and class libraries to the CLR by which CLR handles the MSIL code. As CLR is common so it allows an instance of a class that written in a different language to call a method of the class which written in another language.

  As the word specify, Common means CLR provides a common runtime or execution environment as there are more than 60 .NET programming languages.



Main Components of CLR

Main components of CLR:

- **Common Language Specification (CLS)**
- **Common Type System (CTS)**
- **Garbage Collection (GC)**
- **Just In – Time Compiler (JIT)**

<u>Common Language Specification (CLS)</u>

It defines a set of rules and restrictions that every language must follow which runs under the .NET framework. The languages which follow these set of rules are said to be CLS Compliant. In simple words, CLS enables cross-language integration or Interoperability.

For Example

1. If we talk about C# and VB.NET then, in C# every statement must have to end with a semicolon. It is also called a statement Terminator, but in VB.NET each statement should not end with a semicolon (;).

Explanation of the above Example

So these syntax rules which we have to follow from language to language differ but CLR can understand all the language Syntax because in .NET each language is converted into MSIL code after compilation and the MSIL code is language specification of CLR.

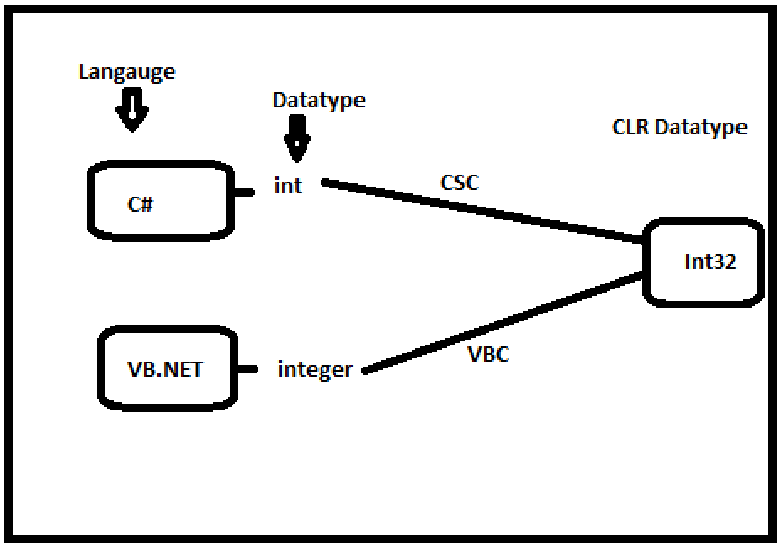
Common Type System (CTS)

Common Type System (CTS) describes the data types that can be used by managed code. CTS define how these types are declared, used and managed in the runtime. It facilitates cross-language integration, type safety, and high-performance code execution. The rules defined in CTS can be used to define our own classes and values.

CTS deals with the data type. So here we have several languages and each and every language has its own data type and one language data type cannot be understandable by other languages but .NET Framework language can understand all the data types.

For Example

C# has an int data type and VB.NET has Integer data type. Hence a variable declared as an int in C# and Integer in VB.NET, finally after compilation, uses the same structure Int32 from CTS.



Garbage Collector:

It is used to provide the *Automatic Memory Management* feature. If there was no garbage collector, programmers would have to write the memory management codes which will be a kind of overhead on programmers.

JIT (Just In Time Compiler):

It is responsible for converting the CIL(Common Intermediate Language) into machine code or native code using the Common Language Runtime environment.

Microsoft Intermediate Language (MSIL)

A .NET programming language (C#, VB.NET, J# etc.) does not compile into executable code; instead it compiles into an intermediate code called Microsoft Intermediate Language (MSIL). As a programmer one need not worry about the syntax of MSIL - since our source code in automatically converted to MSIL. The MSIL code is then send to the CLR (Common Language Runtime) that converts the code to machine language which is then run on the host machine.

MSIL is similar to Java Byte code. A Java program is compiled into Java Byte code (the .class file) by a Java compiler, the class file is then sent to JVM which converts it into the host machine language.

2. **Framework Base Classes:-** Base classes provide basic data types, collection of classes, method and other general classes for use by .NET languages.

The .NET Framework includes a set of standard class libraries. A class library is a collection of methods and functions that can be used for the core purpose.

For example, there is a class library with methods to handle all file-level operations. So there is a method which can be used to read the text from a file. Similarly, there is a method to write text to a file.

Much of the functionality in the base framework classes resides in the vast namespace called **System**. We can use the base classes in the system namespace for many different tasks including:

–Input/output Operations

–String Handling

–Managing arrays, lists, maps, etc

–Database Management

–Connecting to the Internet

–Accessing files and file systems

–And many more

3. **User and Program Interfaces**

The different types of applications that can be built in the .Net framework are classified broadly into the following categories.

- **Windows forms**
- **Web forms**
- **Console Applications**
- **Web Services**

  For example

WinForms – This is used for developing Forms-based applications, which would run on an end user machine. Notepad is an example of a client-based application.

ASP.Net – This is used for developing web-based applications, which are made to run on any browser such as Internet Explorer, Chrome or Firefox.

The Web application would be processed on a server, which would have Internet Information Services Installed.

Internet Information Services or IIS is a Microsoft component which is used to execute an Asp.Net application.

The result of the execution is then sent to the client machines, and the output is shown in the browser.

Features of .Net

Microsoft .NET provides a framework that facilitates designing and developing applications that are portable, scalable, and robust, and that can be executed in a distributed environment. It presents a platform and device-independent computing model in a managed environment.

The Microsoft .NET framework provides a lot of features. Microsoft has designed the features of the .NET framework by using the technologies that are required by software developers to develop applications for modern as well as future business needs. The key features of .NET are:

1. Common Executive Environment:-

All .NET applications run under a common execution environment, called the Common Language Runtime. The CLR facilitates the interoperability between different .NET languages such as C#, Visual Basic, Visual C++, etc. by providing a common environment for the execution of code written in any of these languages.

2. Common Type System:-

The .NET framework follows types of systems to maintain data integrity across the code written in different .NET compliant programming languages. CTS ensure that objects of the programs that are written in different programming languages can communicate with each other to share data.

CTS prevent data loss when a type in one language transfers data to its equivalent type in one language transfer data to its equivalent type in other languages. For example, CTS ensures that data is not lost while transferring an integer variable of visual basic code to an integer variable of C# code.

The common type system CTS defines a set of types and rules that are common to all languages targeted at the CLR. It supports both value and reference types. Value types are created in the stack and include all primitive types, structs, and enums. In contrast, reference types are created in the managed heap and include objects, arrays, collections, etc.

3. Multi-language support:-

.NET provides multi-language support by managing the compilers that are used to convert the source to intermediate language (IL) and from IL to native code, and it enforces program safety and security.

The basis for multiple language support is the common type system and metadata. The basic data types used by the CLR are common to all languages. There are therefore no conversion issues with the basic integer, floating-point and string types.

All languages deal with all data types in the same way. There is also a mechanism for defining and managing new types.

4. Tool Support:-

The CLR works hand-in-hand with tools like visual studio, compilers, debuggers, and profilers to make the developer's job much simpler.

5. Security:-

The CLR manages system security through user and code identity coupled with permission checks. The identity of the code can be known and permission for the use of resources granted accordingly. This type of security is a major feature of .NET. The .NET framework also provides support for role-based security using windows NT accounts and groups.

6. Automatic Resource Management:-

The .NET CLR provides efficient and automatic resource management such as memory, screen space, network connections, database, etc. CLR invokes various built-in functions of .NET framework to allocate and de-allocate the memory of .NET objects.

Therefore, programmers need not write the code to explicitly allocate and de-allocate memory to the program.

7. Easy and rich debugging support:-

The .NET IDE (integrated development environment) provides an easy and rich debugging support. Once an exception occurs at run time, the program stops and the IDE marks the line which contains the error along with the details of that error and possible solutions. The runtime also provides built-in stack walking facilities making it much easier to locate bugs and error.

8. Simplified development:-

With .NET installing or uninstalling, a window-based application is a matter of copying or deleting files. This possible because. .NET components are not referenced in the registry.

9. Framework class library:-

The framework class library (FCL) of the .NET framework contains a rich collection of classes that are available for developers to use these classes in code Microsoft has developed these classes to fulfill various tasks of applications, such as working with files and other data storages, performing input-output operations, web services, data access, and drawing graphics.

The classes in the FCL are logically grouped under various namespaces such as system, System.collections, system.diagnostics, system.Globalization, system.IO, system.text etc.

10. Portability:-

The application developed in the .NET environment is portable. When the source code of a program written in a CLR compliant language complies, it generates a machine-independent and intermediate code. This was originally called the Microsoft Intermediate Language (MSIL) and has now been renamed as the Common Intermediate Language (CIL). CIL is the key to portability in .NET.

Applications that can be developed using .Net

When we hear .NET, the first idea that comes to our mind will probably be internet or networked applications. Although this is absolutely true, there are many more types of applications to create with .NET.
So here is a more or less full list of various types of application that we can develop on .NET.

•**ASP.Net Web applications** are programs that used to run inside some web server to fulfill the user requests over the http. ASP.NET Web applications can range from simple Web sites that consist of HTML pages to advanced enterprise applications that run on local and remote networks. These enterprise applications also provide components for exchanging data using XML. This type includes dynamic and data driven browser based applications. (Ex: Hotmail and Google).

•**Web services** are “web callable” functionality available via industry standards like HTTP, XML and SOAP.

•**Windows applications** are form based standard Windows desktop applications for common day to day tasks. (Ex: Microsoft word). Run only under Windows environment. These applications consume the services provided by the Windows operating system.

•**Windows services** are long-running executable applications that run on the system as a background process. These applications do not interfere with the working of the other processes that run on the same computer. Windows services execute within separate Windows sessions created specifically for each Windows service. These services do not have a graphic user interface and are ideal for running on the server. Windows services were earlier called NT services.

•**Console applications** are light weight programs run inside the command prompt (DOS) window. They are commonly used for test applications.

•**Mobile applications** can run on multiple mobile devices, such as Pocket PCs, mobile phones, or personal digital assistants. These applications provide ubiquitous access to data from mobile devices. The .NET Framework automatically makes changes to these applications to enable them to run on multiple browsers, depending on the mobile device.

•**Class libraries** are components that we create once and reuse a number of times in multiple applications. Class libraries allow us to define several classes, along with their methods and interfaces, in one file. These libraries compile to .dll files and facilitate rapid development of new applications because of reusability of code. To access the functionality of the classes in a class library from our application, we need to include a reference to that library in our program.

All types of .NET applications use one or more .NET compliant languages for their design and development. The .NET Framework includes various technologies, such as ASP.NET, VB.NET, VC++.NET, and ADO.NET. We use ASP.NET to build Web applications and Web services, VB.NET and VC++.NET to create Windows applications, and ADO.NET for flexible access to databases.

Unit-2

.Net Framework using C#

C-Sharp Language (C#):

C# is a modern, general-purpose, object-oriented programming language developed by Microsoft and approved by European Computer Manufacturers Association (ECMA) and International Standards Organization (ISO).

C# was developed by **Anders Hejlsberg** and his team during the development of .Net Framework.

C# is used to develop web apps, desktop apps, mobile apps, games and much more. It is a modern, object-oriented, and type-safe programming language. C# has its roots in the C family of languages and will be immediately familiar to C, C++, Java, and JavaScript programmers.

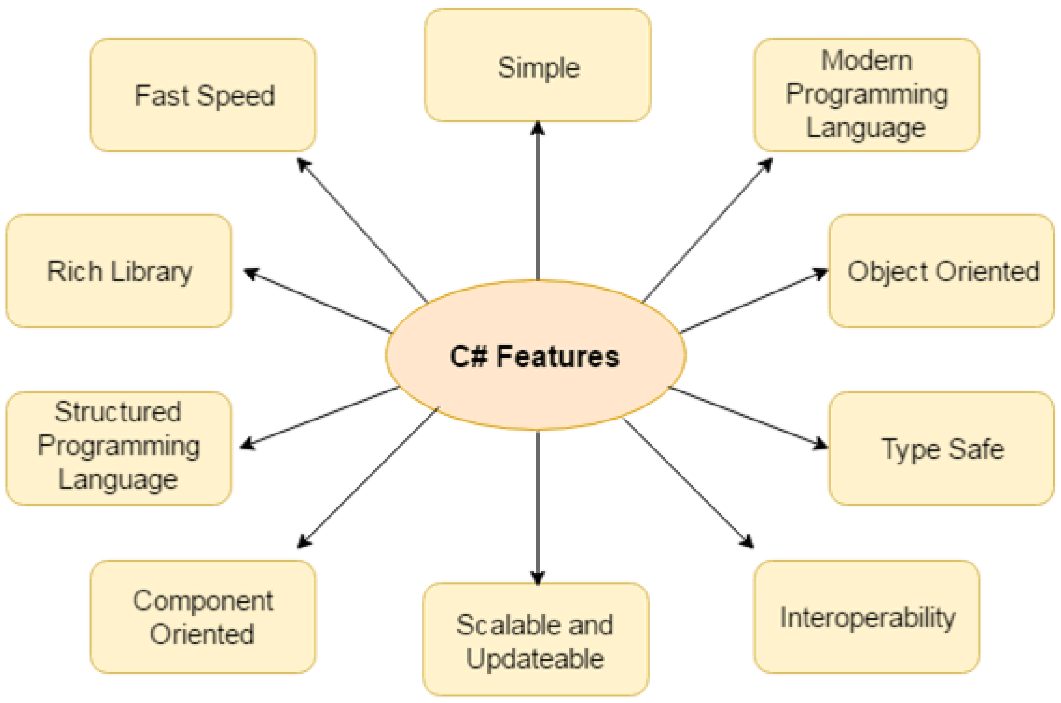
The following reasons make C# a widely used professional language −

- It is a modern, general-purpose programming language
- It is object oriented.
- It is component oriented.
- It is easy to learn.
- It is a structured language.
- It produces efficient programs.
- It can be compiled on a variety of computer platforms.
- It is a part of .Net Framework.

C# Features

C# is object oriented programming language. It provides a lot of **features** that are given below.

1. Simple
2. Modern programming language
3. Object oriented
4. Type safe
5. Interoperability
6. Scalable and Updateable
7. Component oriented
8. Structured programming language
9. Rich Library
10. Fast speed



1) Simple

It is quite simple to use C# as it has various features and functionalities. Moreover, it provides a structured approach that makes the program easier to understand. It provides **structured approach** (to break the problem into parts), rich set of **library functions**, **data types** etc. C# code does not require header files. All code is written inline.

2) Modern Programming Language

C# programming is based upon the current trend and it is very powerful and simple for building scalable, interoperable and robust applications.

C# supports number of modem features, such as:

- **Automatic Garbage Collection**
- **Error handling features**
- **Modern debugging features**
- **Robust Security features**

3) Object Oriented

C# is pure object oriented programming language. OOPs makes development and maintenance easier where as in Procedure-oriented programming language it is not easy to manage if code grows as project size grow.

In C#, everything is an object. There are no more global functions, variable and constants.

It supports all three object oriented features:

- Encapsulation
- Inheritance
- Polymorphism

4) Type Safe

Type safety promotes robust programming. Therefore it improves a security of the program. It will not allow us to mix up one data type with the other data type.

5) Interoperability

Language interoperability is the ability of code to interact with code that is written using a different programming language. Language interoperability can help maximize code reuse and, therefore, improve the efficiency of the development process. It is the ability for two or more languages to interact as part of the same system. In other words, two .NET compliant languages can interoperate with each other. This simply means a function in VB .NET can be called by C# and vice-versa.

6) Scalable and Updateable

C# is an automatically scalable as well as updatable language. .NET has introduced **assemblies** which are self describing by means of their manifest. Manifest establishes the assembly identity, version, culture and digital signature etc. Assemblies need not to be register anywhere.

To scale our application we delete the old files and updating them with new ones. No registering of dynamic linking library.

Updating software components is an error prone task. Revisions made to the code. This can affect the existing program. C# support versioning in the language. Native support for interfaces and method overriding enable complex frame works to be developed and evolved over time.

7) Component Oriented

C# is component oriented programming language. **Component-oriented** programming is a technique of developing software applications by **combining pre-existing** and **new components**.

8) Structured Programming Language

C# is a structured programming language in the sense that we can break the program into parts using functions. So, it is easy to understand and modify.

9) Rich Library

C# provides a lot of inbuilt functions that makes the development fast.

10) Fast Speed

The compilation and execution time of C# language is fast.

C# - Data Types

C# is a strongly-typed language. It means we must declare the type of a variable that indicates the kind of values it is going to store, such as integer, float, decimal, text, etc.

The following declares and initialized variables of different data types.

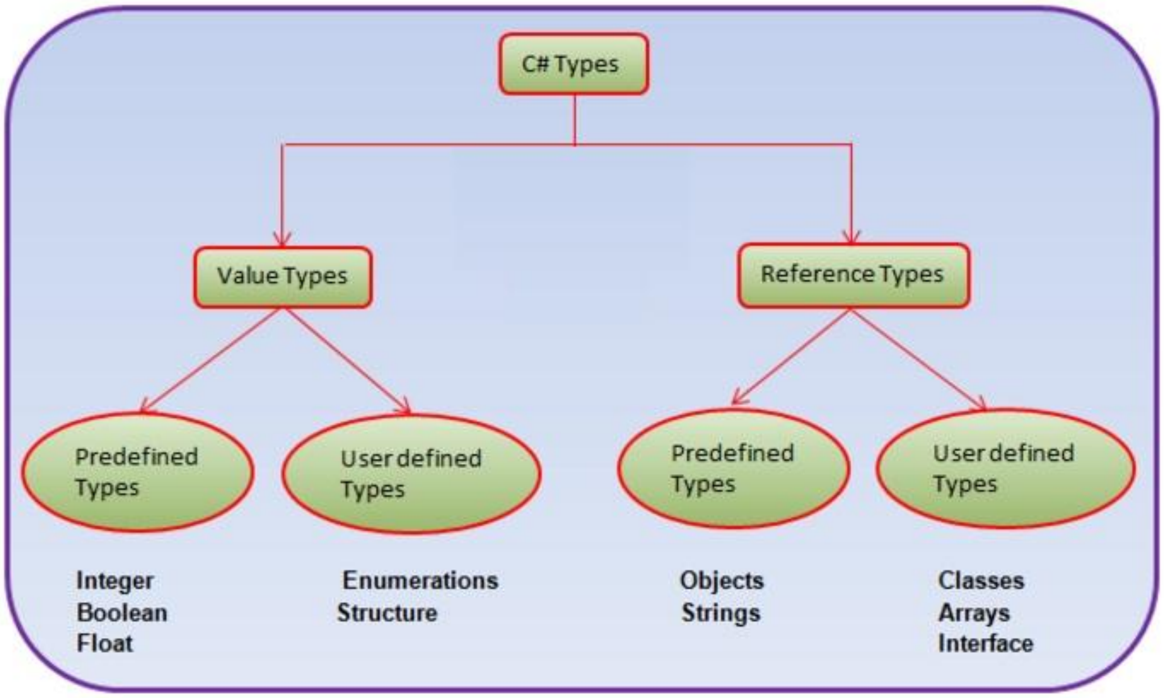
Example: Variables of Different Data Types

- string stringVar = "Hello World!!";
- int intVar = 100;
- float floatVar = 10.2f;
- char charVar = 'A';
- bool boolVar = true;

C# mainly categorized data types in two types:

Value types and **Reference types.**

Value types include simple types (such as int, float, bool, and char), enum types and struct types. Reference types include class types, object types, interface types, delegate types, and array types.

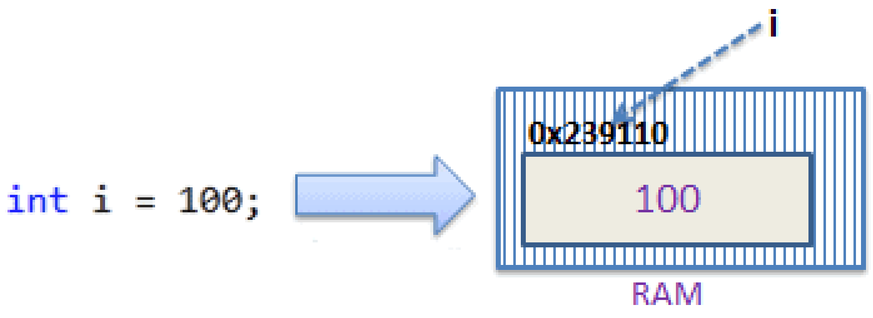


Value Type

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

For example, consider integer variable int i = 100;

The system stores 100 in the memory space allocated for the variable i.



The following data types (with size) are all of value type:

Data Type	Size
bool	1 bit
char	2 bytes
int	4 bytes
long	8 bytes
float	4 bytes
double	8 bytes
enum	User-defined data type
structure	User-defined data type

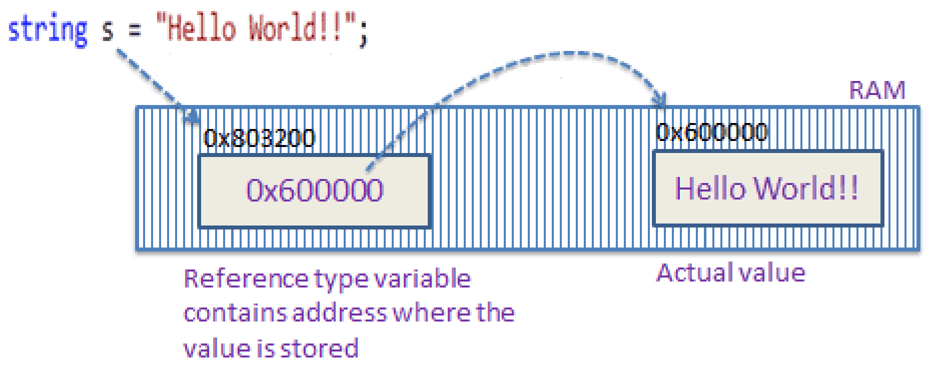
Reference Type

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 the following string variable:

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

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


string s = "Hello World!!";
RAM
0x803200
0x600000
0x600000
Hello World!!
Reference type variable contains address where the value is stored
Actual value

As we can see in the above image, the system selects a random location in memory (0x803200) for the variables. 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 the value itself.

The followings are reference type data types:

- String
- Object
- Arrays (even if their elements are value types)
- Class
- Delegate
- Interface

Numbers

Number types are divided into two groups:

Integer types stores whole numbers, positive or negative (such as 123 or -456), without decimals. Valid types are `int` and `long`. Which type we should use, depends on the numeric value.

Floating point types represents numbers with a fractional part, containing one or more decimals. Valid types are `float` and `double`.

Integer Types

Int

The `int` data type can store whole numbers from -2147483648 to 2147483647. In general, and in our tutorial, the `int` data type is the preferred data type when we create variables with a numeric value.

Example

```
int myNum = 100000;
Console.WriteLine(myNum);
```

Long

The `long` data type can store whole numbers from -9223372036854775808 to 9223372036854775807. This is used when `int` is not large enough to store the value. Note that we should end the value with an "L": (although not required):

Example

```
long myNum = 15000000000L;
Console.WriteLine(myNum);
```

Floating Point Types

We should use a floating point type whenever we need a number with a decimal, such as 9.99 or 3.14515.

Float

The `float` data type can store fractional numbers from 3.4e−038 to 3.4e+038. It has 7 digits Precision. Note that we should end the value with an "F":

Example

```
float myNum = 5.75F;
Console.WriteLine(myNum);
```

Double

The `double` data type can store fractional numbers from 1.7e−308 to 1.7e+308. It has 14 – 15 digit Precision. Note that we can end the value with a "D" (although not required because by default floating data types are the double type.):

Example

```
double myNum = 19.99D;
```

```
Console.WriteLine(myNum);
```

Booleans

A boolean data type is declared with the `bool` keyword and can only take the values `true` or `false`:

Example

```
bool isCSharpFun = true;
bool isFishTasty = false;
Console.WriteLine(isCSharpFun);   // Outputs True
Console.WriteLine(isFishTasty);   // Outputs False
```

Characters

The `char` data type is used to store a **single** character. The character must be surrounded by single quotes, like 'A' or 'c':

Example

```
char myGrade = 'B';
Console.WriteLine(myGrade);
```

Strings

The `string` data type is used to store a sequence of characters (text). String values must be surrounded by double quotes:

Example

```
string greeting = "Hello World";
Console.WriteLine(greeting);
```

Default values of C# data types

C# does not allow us to use an uninitialized variable, which means the variable must have a value before we use it. Although this idea of **definite assignment** helps reduce errors, because it is enforced by the compiler, it can be cumbersome if we have to explicitly provide a default value for every field.

To alleviate this burden, fields, or member variables, are always initially assigned with an appropriate default value. The following table shows the default value for the different predefined data types.

Type	**Default**
sbyte, byte, short, ushort, int, uint, long, ulong	0
char	'\x0000'

float	0.0f
double	0.0d
decimal	0.0m
bool	false
object	null
string	null

As we can see, for the integral value types, the default value is zero. The default value for the char type is the character equivalent of zero and false for the bool type. The object and string types have a default value of null,
Or we can say that all value type has default value zero and all reference type default value is null.

C# | Identifiers

In programming languages, identifiers are used for identification purposes. Or in other words, identifiers are the user-defined name of the program components. In C#, an identifier can be a class name, method name, variable name or label.

Example:

```
public class program
{
   public  static void Main()
   {
        int x;
   }
}
```

Here the total number of identifiers present in the above example is 3 and the names of these identifiers are:

- **program:** Name of the class
- **Main:** Method name
- **x:** Variable name

Rules for defining identifiers in C#:

There are certain valid rules for defining a valid C# identifier. These rules should be followed; otherwise, we will get a compile-time error.

- The only allowed characters for identifiers are all alphanumeric characters (**[A-Z]**, **[a-z]**, **[0-9]**), '**_**' (underscore). For example "name@" is not a valid C# identifier as it contain '@' – special character.
- Identifiers should not start with digits ([0-9]). For example "123name" is a not a valid in C# identifier.
- Identifiers should not contain white spaces.
- Identifiers are not allowed to use as keyword unless they include @ as a prefix. For example, **@as** is a valid identifier, but "**as**" is not because it is a keyword.
- C# identifiers are case-sensitive.
- C# identifiers cannot contain more than 512 characters.
- Identifiers do not contain two consecutive underscores.

C# Variables

Variables are containers for storing data values.

In C#, there are different **types** of variables (defined with different keywords), for example:

- int - stores integers (whole numbers), without decimals, such as 123 or -123
- double - stores floating point numbers, with decimals, such as 19.99 or -19.99
- char - stores single characters, such as 'a' or 'B'. Char values are surrounded by single quotes
- string - stores text, such as "Hello World". String values are surrounded by double quotes
- bool - stores values with two states: true or false

Declaring (Creating) Variables

To create a variable, we must specify the type and assign it a value:

Syntax

```
type variableName = value;
```

Where *type* is a C# type (such as `int` or `string`), and *variableName* is the name of the variable (such as **x** or **name**).

The **equal sign** is used to assign values to the variable.

To create a variable that should store text, look at the following example:

Example

Create a variable called **name** of type `string` and assign it the value "**Ashish**":

```
string name = "Ashish";
Console.WriteLine(name);
```

To create a variable that should store a number, look at the following example:

Example

Create a variable called **myNum** of type `int` and assign it the value **15**:

```
int myNum = 15;
Console.WriteLine(myNum);
```

We can also declare a variable without assigning the value, and assign the value later:

Example

```
int myNum;
myNum = 15;
Console.WriteLine(myNum);
```

Note that if we assign a new value to an existing variable, it will overwrite the previous value:

Example

Change the value of `myNum` to 20:

```
int myNum = 15;
myNum = 20; // myNum is now 20
Console.WriteLine(myNum);
```

Constants

However, we can add the `const` keyword if we don't want others (or our self) to overwrite existing values (this will declare the variable as "constant", which means unchangeable and read-only):

Example

```
const int myNum = 15;
myNum = 20; // error
```

The const keyword is useful when we want a variable to always store the same value.

Note: We cannot declare a constant variable without assigning the value. If we do, an error will occur: A const field requires a value to be provided.

Other Types

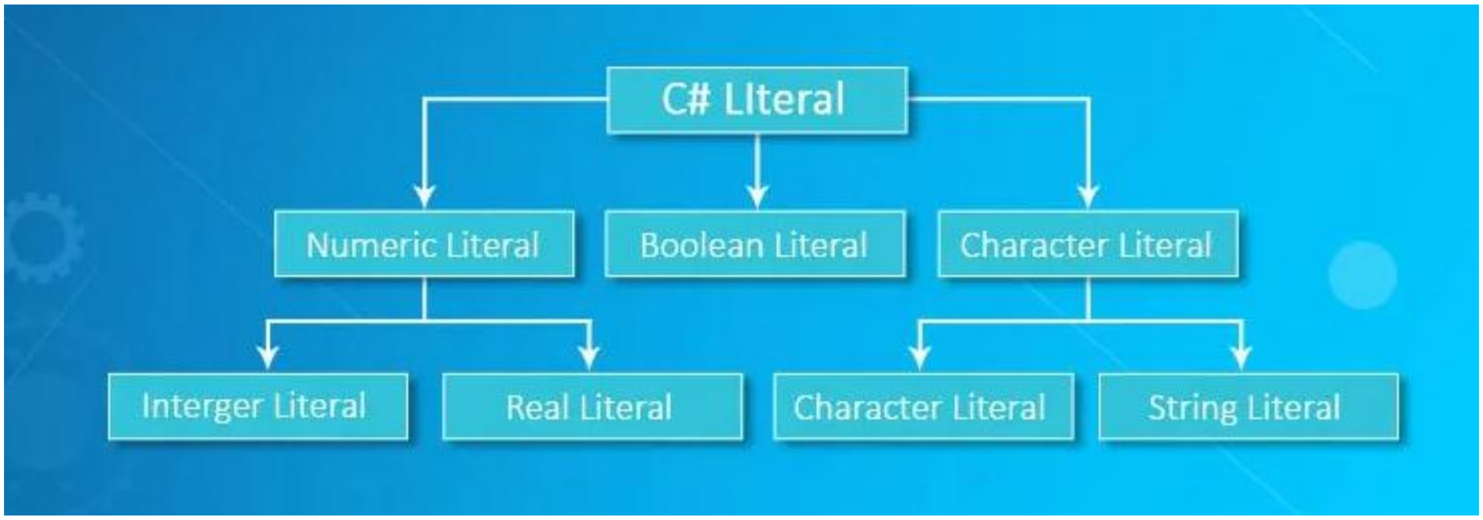
A demonstration of how to declare variables of other types:

Example

```
int myNum = 5;
double myDoubleNum = 5.99D;
char myLetter = 'D';
bool myBool = true;
string myText = "Hello";
```

C# Literals

Literals in C# are the fixed value used by a variable that is predefined and cannot be modified during the execution of the code. These are the convenient form of constant values like other variables but their values cannot be changed. The value used by a variable can be integer, decimal, floating type or string. There are different types of literals in C# with different forms. There are various types of literals in C#.



1. Integer Literals
2. Floating-point Literals (Real Literals)
3. Character Literals
4. String Literals
5. Boolean Literals

Following are the different types of literals in C#.

1. Integer Literals

The literal of integer type can be octal, decimal or hexadecimal. The prefix is used to specify whether it is decimal, octal or hexadecimal. U and u are also used as a suffix with integer type literals for unsigned numbers and l and L used for long numbers. Every literal is of integer type by default.

- **Decimal Literals:** In the decimal type of literals 0-9 digits are allowed. No prefix is required for the decimal type of literals.

  int x = 100; // decimal type

- **Octal Literals:** In the octal type of literals **0-7** digits are allowed. **0** is used as a prefix to specify the form of octal type literals.

  int x = 072; // octal type

- **Hexa-decimal literals:** In the hexadecimal type of literals, digits from **0- 9** and characters from **A-f** are allowed. Uppercase and lowercase both types of characters are allowed in this case. **0X** or **0x** is used as a prefix to specify the form of hexadecimal type of literals.

  int x = 0x123f;   // hexadecimal type

2. String Literals

The string type literals are enclosed in (“”)/ double quotes and also can be started with @””. Long lines can be broken into multiple lines with string literals and be separated by using blank spaces.

string s= “Hi”;   // string literals

3. Character Literals

The character type literals are enclosed in (‘’)/single quotes. There are three ways to specify character literals.

- **Single Quote:** Characters literals can be specified as single char using a single quote.
- **Unicode Representation:** Character literals can be specified using Unicode Representation ‘\uxxxx’ where xxxx are the hexadecimal numbers.
- **Escape Sequence:** There are some escape characters known as char literals.

  char c = ‘\n’;

Following are some escape sequence literals explained with there meanings.

Escape Sequence	**Meaning**
\\	Character
\’	Character
\’’	Character
\?	Character
\a	Alert
\b	Backspace
\n	Newline
\f	Form feed
\v	Vertical tab
\xhh	Hexadecimal number

4. Floating Point Literals

In the floating type of literal, there is an integer part, a fractional part, decimal part and exponent part. The floating type literal is of double type. F or f can be used as a suffix to specify the value as it cannot be assigned directly to the float variable.

5. Boolean Literals

In the Boolean type of literals, true and false will be the only two values.

Examples of C# Literals

Example #1 – Integer Literal

```
using System;
namespace Literals
{
class Program
{
static void Main(string[] args)
{
int x = 212;   // decimal literal
int y = 0145;  // octal literal
int z = 0x4b;  // hexadecimal literal
Console.WriteLine(x);
Console.WriteLine(y);
Console.WriteLine(z);
Console.ReadLine();
}
}
}
```

Output:

```
212
145
75
```

Explanation: In the above example, there are various forms of integer type literals. In this no prefix is used for decimal form, 0 is used to specify the octal form and 0x is used for specifying the hexadecimal number. Using prefix we can define the form of integer type literal. In this code, first, there is a literal of decimal type with no prefix, a second type is an octal form with 0 as prefix and at last, we have a hexadecimal type with 0x as a prefix.

Example #2 – Floating Point Literal

```
using System;
namespace Literals
{
class Program
{
static void Main(string[] args)
{
double x = 187.231;
double y = 0141.361;
double z = 374159E-4F;
Console.WriteLine(x);
Console.WriteLine(y);
Console.WriteLine(z);
Console.ReadLine();
}
}
}
```

Output:

```
187.231
141.361
37.415901184082
```

Explanation: In the above example, floating-point literals are implemented. It can be a decimal number, fractional or any exponent. So we can represent it either in decimal or in exponential form. The floating type literal is of double type. F or F can be used as a suffix to specify the value as it cannot be assigned directly to the float variable.

Example #3 – Character Literals

```
using System;
namespace Literals
{
class Program
{
static void Main(string[] args)
{
char c = 'b';
char ch = '\u0071';
Console.WriteLine(c);
Console.WriteLine(ch);
Console.WriteLine("\nHello World\t!");
Console.ReadLine();
}
}
}
```

Output:

```
b
q

Hello World	!
```

Explanation: In the above example, character type literals are implemented. In the above code, all three forms of character type are shown. We can specify the character literal using a single quote, Unicode representation, and escape sequence. We have multiple types of escape characters with their own meanings. In this code, the first single quote character is specified where the second one has Unicode representation and then, at last, we have escape form type of character literals.

Example #4 – String Literal

```
using System;
namespace Literals
{
class Program
{
static void Main(string[] args)
{
String s1 = "This is C# programming";
String s2 = @"This is C# programming";
Console.WriteLine(s1);
Console.WriteLine(s2);
Console.ReadLine();
}
}
}
```

Output:

```
This is C# programming
This is C# programming
```

Explanation: In the above example, string literals are implemented. There are two ways to specify string literals as shown in the code. First, it is implemented using double quotes and then @ symbol is used to specify the string.

Example #5 – Boolean Type Literal

```
using System;
namespace Literals
{
class Program
{
static void Main(string[] args)
{
bool x = true;
bool y = false;
Console.WriteLine(x);
Console.WriteLine(y);
Console.ReadLine();
}
}
}
```

Output:

```
True
False
```

Explanation: In the above example, Boolean type literals are implemented which has two value either true or false.

C# Syntax

Example

```
using System;
namespace HelloWorld
{
  class Program
  {
   public static void Main(string[] args)
    {
      Console.WriteLine("Hello World!");
    }
  }
}
```

Output

```
Hello World!
```

Example explained

Line 1: `using System` means that we can use classes from the `System` namespace.

Line 2: `namespace` is a used to organize our code, and it is a container for classes and other namespaces.

Line 3: The curly braces `{}` marks the beginning and the end of a block of code.

Line 4: `class` is a container for data and methods, which brings functionality to our program. Every line of code that runs in C# must be inside a class. In our example, we named the class Program.

Line 5: Another thing that always appears in a C# program is the `Main` method. Any code inside its curly brackets `{}` will be executed. Here M of Main should be capital.

Line 6: `Console` is a class of the `System` namespace, which has a `WriteLine()` method that is used to output/print text. In our example it will output "Hello World!".

If we omit the `using System` line, we would have to write `System.Console.WriteLine()` to print/output text.

Note: Every C# statement ends with a semicolon ;.

Note: C# is case-sensitive: "MyClass" and "myclass" has different meaning.

static: It means Main Method can be called without an object. We need an entry point into our program. Static means that we can call the function without having instance of a class (or creating object of class).

public: It is access modifiers which means the compiler can execute this from anywhere.

void: The Main method doesn’t return anything. Main can either have a void or int.

Main(): It is the configured name of the Main method.

String []args: For accepting the *zero-indexed command line arguments*. args is the user-defined name. So we can change it by a valid identifier. [] must come before the args otherwise compiler will give errors. The Main method can be declared with or without a string [] args parameter that contains command-line arguments.

Applicable Access Modifiers: public, private, protected, internal, protected internal access modifiers can be used with the Main() method. The **private protected** access modifier cannot be used with it.

Display Variables

The `WriteLine()` method is often used to display variable values to the console window.

To combine both text and a variable, use the `+` character:

Example

```
string name = "Mohan";
Console.WriteLine("Hello " + name);
```

Output

```
Hello Mohan
```

We can also use the `+` character to add a variable to another variable:

Example

```
string firstName = "Mohan ";
string lastName = "Sharma";
string fullName = firstName + lastName;
Console.WriteLine(fullName);
```

Output

```
Mohan Sharma
```

For numeric values, the `+` character works as a mathematical operator (notice that we use `int` (integer) variables here):

Example

```
int x = 5;
int y = 6;
Console.WriteLine(x + y); // Print the value of x + y
```

Output

```
11
```

C# Type Conversion

Type casting is when we assign a value of one data type to another type.

In C#, there are two types of casting:

- **Implicit Conversion** (automatically) - converting a smaller type to a larger type size. These conversions are performed by C# in a type-safe manner.

  char -> int -> long -> float -> double
- **Explicit Conversion** (manually) - converting a larger type to a smaller size type

  double -> float -> long -> int -> char

Implicit Casting

Implicit casting is done automatically when passing a smaller size type to a larger size type:

Example

```
int myInt = 9;
double myDouble = myInt;       // Automatic casting: int to double
Console.WriteLine(myInt);      // Outputs 9
Console.WriteLine(myDouble);   // Outputs 9
```

Explicit Casting

Explicit casting must be done manually by placing the type in parentheses () in front of the value:

Example

```
double myDouble = 9.78;
int myInt = (int) myDouble;    // Manual casting: double to int
Console.WriteLine(myDouble);   // Outputs 9.78
Console.WriteLine(myInt);      // Outputs 9
```

Type Conversion Methods

It is also possible to convert data types explicitly by using built-in methods, such as Convert.ToBoolean, Convert.ToDouble, Convert.ToString, Convert.ToInt32 (int) and Convert.ToInt64 (long):

Example

```
int myInt = 10;
double myDouble = 5.25;
bool myBool = true;
Console.WriteLine(Convert.ToString(myInt));    // convert int to string
Console.WriteLine(Convert.ToInt32(myDouble));  // convert double to int
Console.WriteLine(Convert.ToString(myBool));   // convert bool to string
```

C# User Input

We have already learned that Console.WriteLine() is used to output (print) values. Now we will use Console.ReadLine() to get user input.

Example

```
Console.WriteLine("Enter username:");
string userName = Console.ReadLine();
Console.WriteLine("Username is: " + userName);
```

User Input and Numbers

The Console.ReadLine() method returns a string. Therefore, we cannot get information from another data type, such as int. The following program will cause an error:

Example

```
Console.WriteLine("Enter your age:");
int age = Console.ReadLine();
Console.WriteLine("Your age is: " + age);
```

We cannot implicitly convert type 'string' to 'int'.

We can convert any type explicitly, by using one of the `Convert.To` methods:

Example

```
Console.WriteLine("Enter your age:");
int age = Convert.ToInt32(Console.ReadLine());
Console.WriteLine("Your age is: " + age);
```

Output

```
Enter your age:
18
Your age is: 18
```

Exercise1: Write a C# Sharp program to print the sum of two numbers.

```
class Exercise1
{
   public static void Main()
   {
      System.Console.WriteLine(15+17);
   }
}
```

Output:

```
32
```

Exercise2: Write a C# Sharp program to swap two numbers.

```
using System;
class Exercise2
{
    public static void Main(string[] args)
     {
        int number1, number2, temp;
        Console.WriteLine("Input the First Number : ");
        number1 = Convert.ToInt32 (Console.ReadLine());
        Console.WriteLine("Input the Second Number : ");
        number2 = Convert.ToInt32 (Console.ReadLine());
        temp = number1;
        number1 = number2;
        number2 = temp;
        Console.WriteLine("After Swapping : ");
        Console.WriteLine("First Number : "+number1);
        Console.WriteLine("Second Number : "+number2);
    }
}
```

Output:

```
Input the First Number : 2
Input the Second Number : 5
After Swapping :
First Number : 5
Second Number : 2
```

Exercise3: Write a C# Sharp program to print on screen the output of adding, subtracting, multiplying and dividing of two numbers which will be entered by the user.

```
using System;
class Exercise3
{
  public static void Main()
```

```
    {
        Console.Write("Enter a number: ");
        int num1= Convert.ToInt32(Console.ReadLine());

        Console.Write("Enter another number: ");
        int num2= Convert.ToInt32(Console.ReadLine());

        Console.WriteLine("{0} + {1} = {2}", num1, num2, num1+num2);
        Console.WriteLine("{0} - {1} = {2}", num1, num2, num1-num2);
        Console.WriteLine("{0} x {1} = {2}", num1, num2, num1*num2);
        Console.WriteLine("{0} / {1} = {2}", num1, num2, num1/num2);
        Console.WriteLine("{0} mod {1} = {2}", num1, num2, num1%num2);
    }
}
```

Output:

```
Enter a number: 10
Enter another number: 2
10 + 2 = 12
10 - 2 = 8
10 x 2 = 20
10 / 2 = 5
10 mod 2 = 0
```

Exercise4: Write a C# Sharp program that takes four numbers as input to calculate and print the average.

```
using System;
class Exercise4
{
  public static void Main()
  {
     double number1,number2,number3,number4;

     Console.WriteLine("Enter the First number: ");
     number1 = Convert.ToDouble(Console.ReadLine());
```

```
    Console. WriteLine ("Enter the Second number: ");
    number2 = Convert.ToDouble(Console.ReadLine());

    Console. WriteLine ("Enter the third number: ");
    number3 = Convert.ToDouble(Console.ReadLine());

    Console. WriteLine ("Enter the fourth number: ");
    number4 = Convert.ToDouble(Console.ReadLine());

    double result = (number1 + number2 + number3 + number4) / 4;
    Console.WriteLine("The average of {0}, {1}, {2}, {3} is: {4} ",
  number1, number2, number3, number4, result);
   }
}
```

Output:

```
Enter the First number: 17
Enter the Second number: 17
Enter the third number: 517
Enter the fourth number: 51
The average of 17, 17, 517, 51 is: 150.5
```

Exercise 5: Write a program in C# Sharp to read 10 numbers from keyboard and find their sum and average.

```
using System;
class Exercise4
{
   public static void Main()
{
   int i,n,sum=0;
   double avg;
            Console.WriteLine("Input the 10 numbers : \n");
            for (i=1;i<=10;i++)
            {
             Console.WriteLine("Number-{0} :",i);
```

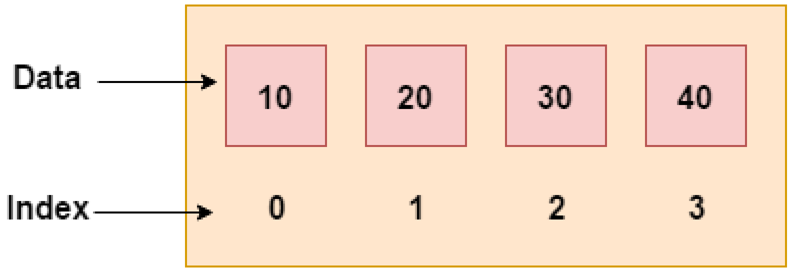
```
                 n= Convert.ToInt32(Console.ReadLine());
                                 sum +=n; // sum=sum+n
                }
                avg=sum/10.0;
                Console.WriteLine("The sum of 10 no is : {0} The Average is : {1}\n",sum,avg);
    }
}
```

Output:

```
Input the 10 numbers :
Number-1 :2
Number-2 :4
Number-3 :6
Number-4 :8
Number-5 :10
Number-6 :12
Number-7 :14
Number-8 :16
Number-9 :18
Number-10 :20
The sum of 10 no is : 110
The Average is : 11
```

C# Arrays

Like other programming languages, array in C# is a group of similar types of elements that have contiguous memory location. That is, arrays are used to store multiple values in a single variable, instead of declaring separate variables for each value. In C#, array is an *object* of base type **System.Array**. In C#, array index starts from 0. We can store only fixed set of elements in C# array.



Declaring Arrays

To declare an array in C#, we can use the following syntax −

datatype[] arrayName;

where,

- *datatype* is used to specify the data type of elements in the array.
- [ ] specifies the rank of the array. The rank specifies the size of the array.
- *arrayName* specifies the name of the array.

For example,

double[] balance;

int [] marks;

Note: We cannot place square brackets after the identifier.

Initializing an Array

Declaring an array does not initialize the array in the memory. When the array variable is initialized, we can assign values to the array.

Array is a reference type, so we need to use the **new** keyword to create an instance of the array. For example,

double[] balance = new double[10];

Assigning Values to an Array

We can assign values to individual array elements, by using the index number, like −

double[] balance = new double[10];

balance[0] = 4500.0;

We can assign values to the array at the time of declaration, as shown −

double [] balance = {2340.0, 4523.69, 3421.0};

We can also create and initialize an array, as shown −

int [] marks = new int[5] { 99, 98, 92, 97, 95};

We may also omit the size of the array, as shown −

int [] marks = new int[] { 99, 98, 92, 97, 95};

In C#, there are different ways to create an array:

```
// Create an array of four elements, and add values later
string[] cars = new string[4];

// Create an array of four elements and add values right away
string[] cars = new string[4] {"Volvo", "BMW", "Ford", "Mazda"};

// Create an array of four elements without specifying the size
string[] cars = new string[] {"Volvo", "BMW", "Ford", "Mazda"};

// Create an array of four elements, omitting the new keyword, and without specifying the size
string[] cars = {"Volvo", "BMW", "Ford", "Mazda"};
```

We should note that if we declare an array and initialize it later, we have to use the new keyword:

Array Length

To find out how many elements an array has, use the Length property: as arrayName.Length

Example

```
using System;
  class Program
  {
    public static void Main()
    {
      int[] marks = {1,10,15,20,25,30};
      Console.WriteLine(marks.Length);
    }
  }
```

Output

```
6
```

Array Index

The **IndexOf()** method of array class in C# searches for the specified object and returns the index of the first occurrence within the entire one-dimensional Array.

We have set the array.

```
int[] arr = new int[10];
arr[0] = 100;
arr[1] = 200;
arr[2] = 300;
arr[3] = 400;
arr[4] = 500;
```

Now use the IndexOf() method and set the element for which we want the index,

The following is the example showing the usage of IndexOf(,) method in C#.

Example

```
using System;
class Program {
  static void Main() {
    int[] arr = new int[5];
    arr[0] = 100;
    arr[1] = 200;
    arr[2] = 300;
    arr[3] = 400;
    arr[4] = 500;
    int a = Array.IndexOf(arr, 400);
    Console.WriteLine(a);
  }
}
```

Output

```
3
```

Loop Through an Array

We can loop through the array elements with the for loop, and use the Length property to specify how many times the loop should run.

The following **example** outputs all elements in the **marks** array:

```
using System;
namespace MyApplication
{
  class Program
  {
    static void Main(string[] args)
    {
      int[] marks = {10,23,12,25,16};
      for (int i = 0; i < marks.Length; i++)
      {
        Console.WriteLine(marks[i]);
      }
    }
  }
}
```

Output

```
10
23
12
25
16
```

The foreach Loop

There is also a foreach loop, which is used exclusively to loop through elements in an **array**:

Syntax

```
foreach (type variableName in arrayName)
{
  // code block to be executed
}
```

The following example outputs all elements in the **marks** array, using a foreach loop:

```
using System;
  class Program
  {
    static void Main(string[] args)
    {
      int[] marks = {12,12,13,34};
      foreach (int i in marks)
      {
        Console.WriteLine(i);
      }
    }
  }
```

Output

```
 12
 12
 13
 34
```

Another example with foreach loop

```
using System;
namespace MyApplication
{
  class Program
  {
    static void Main(string[] args)
    {
      string[] cars = {"Volvo", "BMW", "Ford", "Mazda"};
      foreach (string b in cars)
      {
        Console.WriteLine(b);
      }
    }
  }
  }
```

Output

```
    Volvo
    BMW
    Ford
    Mazda
```

The example above can be read like this: **for each** string element (called **i** - as in **i**ndex) in **cars**, print out the value of **i**.

If we compare the for loop and foreach loop, we will see that the foreach method is easier to write, it does not require a counter (using the Length property), and it is more readable.

Sort Arrays

There are many array methods available, for example Sort(), which sorts an array alphabetically or in an ascending order: **Example**

```
using System;
  class Program
  {
    static void Main(string[] args)
    {
      string[] cars = {"Volvo", "BMW", "Ford", "Mazda"};
      Array.Sort(cars);
      foreach (string i in cars)
      {
        Console.WriteLine(i);
      }
      int[] myNumbers = {5, 1, 8, 9};
      Array.Sort(myNumbers);
      foreach (int i in myNumbers)
      {
        Console.WriteLine(i);
      }
    }
  }
```

Output

```
BMW
Ford
Mazda
Volvo
1
5
8
9
```

System.Linq Namespace

Other useful array methods, such as Min, Max, and Sum, can be found in the System.Linq namespace:

Example

```
using System;
using System.Linq;
  class Program
  {
    static void Main(string[] args)
    {
      int[] myNumbers = {5, 1, 25, 9};
      Console.WriteLine(myNumbers.Max());  // returns the largest value
      Console.WriteLine(myNumbers.Min());  // returns the smallest value
      Console.WriteLine(myNumbers.Sum());  // returns the sum of elements
    }
  }
```

Output

```
25
1
40
```

Example: All array functions in a single program:

```
using System;
using System.Linq;
 class Program
{
  static void Main()
```

```
        {
         int[] a = new int[5] {5,10,45,6,2};
        foreach (int i in a)
        Console.WriteLine(i);
        Array.Sort(a);
        Console.WriteLine("Sorted array elements are");
        foreach (int i in a)
        {
        Console.WriteLine(i);
        }
         Console.WriteLine("maximum elemets is " +a.Max());
        Console.WriteLine("minimum elemets is " +a.Min());
        Console.WriteLine("sum of elemets are " +a.Sum());
        Console.WriteLine("lenghth of array is " +a.Length);
      }
    }
```

Output:

```
10
45
6
2
Sorted array elements are
2
5
6
10
45
maximum elemets is 45
minimum elemets is 2
sum of elemets are 68
lenghth of array is 5
```

<u>Another Example</u>

Let's see a simple example of C# array, where we are going to declare, initialize and traverse array.

```
using System;
class ArrayExample
{
  public static void Main(string[] args)
```

```
        {
            int[] arr = new int[5];//creating array
            arr[0] = 10;//initializing array
            arr[2] = 20;
            arr[4] = 30;
            //traversing array
            for (int i = 0; i < arr.Length; i++)
            {
                Console.WriteLine(arr[i]);
            }
        }
    }
```

Output:

```
10
0
20
0
30
```

Example 2

```
using System;
class MyArray
{
    static void Main(string[] args) {
      int []  n = new int[10];
      int i,j;
      for ( i = 0; i < 10; i++ )
     {
        n[ i ] = i + 100;
        Console.WriteLine("Element[{0}] = {1}", i, n[i]);
      }
    }
  }
```

Output −

```
Element[0] = 100
Element[1] = 101
Element[2] = 102
Element[3] = 103
Element[4] = 104
Element[5] = 105
Element[6] = 106
Element[7] = 107
Element[8] = 108
Element[9] = 109
```

Exercise5: Write a program in C Sharp to store elements in an array and print it.

```
using System;
class Exercise5
{
  public static void Main()
{
  int[] arr = new int[10];
  int i;
  Console.Write("Input 10 elements in the array :\n");
  for(i=0; i<10; i++)
  {
   Console.WriteLine("element - {0} : ",i);
   arr[i] = Convert.ToInt32(Console.ReadLine());
  }
  Console.WriteLine("\nElements in array are: ");
  for(i=0; i<10; i++)
  {
     Console.WriteLine("{0}", arr[i]);
  }
 }
}
```

Exercise6: Write a program in C# to find the sum of all elements of the array.

```
using System;
public class Exercise6
{
    public static void Main()
{
    int[] a= new int[100];
    int i, n, sum=0;
       Console.Write("Input the number of elements to be stored in the array :");
        n = Convert.ToInt32(Console.ReadLine());
       Console.Write("Input {0} elements in the array :\n",n);
       for(i=0;i<n;i++)
        {
          Console.Write("element - {0} : ",i);
                 a[i] = Convert.ToInt32(Console.ReadLine());
        }
    for(i=0; i<n; i++)
    {
        sum += a[i];
    }
    Console.Write("Sum of all elements stored in the array is : {0}\n\n", sum);
  }
}
```

C# Multidimensional Arrays

So far, we have worked with one-dimensional arrays. The number of indexes needed to specify an element is called the *dimension*, or *rank* of the array.

C# allows multidimensional arrays. Multi-dimensional arrays are also called rectangular array. We can declare a 2-dimensional array of strings as –

string [,] names;

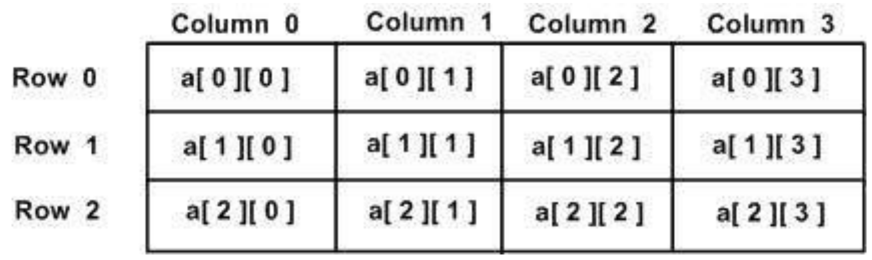
or, a 3-dimensional array of int variables as −

int [, ,] m;

Two-dimensional array

The simplest form of the multidimensional array is the 2-dimensional array. A 2-dimensional array is a list of one-dimensional arrays.

A 2-dimensional array can be thought of as a table, which has x number of rows and y number of columns. Following is a 2-dimensional array, which contains 3 rows and 4 columns −



	Column 0	Column 1	Column 2	Column 3
Row 0	a[0][0]	a[0][1]	a[0][2]	a[0][3]
Row 1	a[1][0]	a[1][1]	a[1][2]	a[1][3]
Row 2	a[2][0]	a[2][1]	a[2][2]	a[2][3]

Thus, every element in the array a is identified by an element name of the form a[i , j], where a is the name of the array, and i and j are the subscripts that uniquely identify each element in array a.

Initializing Two-Dimensional Arrays

Multidimensional arrays may be initialized by specifying bracketed values for each row. The Following array is with 3 rows and each row has 4 columns.

```
  int [,] a = new int [3,4]
{
  {0, 1, 2, 3},   /*  initializers for row indexed by 0 */
  {4, 5, 6, 7},   /*  initializers for row indexed by 1 */
  {8, 9, 10, 11}   /*  initializers for row indexed by 2 */
};
```

Accessing Two-Dimensional Array Elements

An element in 2-dimensional array is accessed by using the subscripts. That is, row index and column index of the array. For example,

```
int val = a[2,3];
```

The above statement takes 4th element from the 3rd row of the array. We can verify it in the above diagram. Let us check the program to handle a two dimensional array −

```
using System;
namespace ArrayApplication {
  class MyArray {
    static void Main(string[] args) {
        /* an array with 5 rows and 2 columns*/
        int[,] a = new int[5, 2] {{0,0}, {1,2}, {2,4}, {3,6}, {4,8} };
        int i, j;
        /* output each array element's value */
        for (i = 0; i < 5; i++) {
           for (j = 0; j < 2; j++) {
              Console.WriteLine("a[{0},{1}] = {2}", i, j, a[i,j]);
           }
        }
        Console.ReadKey();
     }
  }
}
```

Output-

```
a[0,0]: 0
a[0,1]: 0
a[1,0]: 1
a[1,1]: 2
a[2,0]: 2
a[2,1]: 4
a[3,0]: 3
a[3,1]: 6
a[4,0]: 4
a[4,1]: 8
```

We can traverse a two-dimensional array with the foreach loop.

Example

```
using System;
int[,] vals = new int[4, 2]
{
   { 9, 99 },
   { 3, 33 },
   { 4, 44 },
   { 1, 11 }
};
foreach (var val in vals)
{
    Console.WriteLine(val);
}
```

With the foreach loop, we get the elements one-by-one from the beginning to the end.

```
9
99
3
33
4
44
1
11
```

C# Strings

Strings are one of the most important data types in any modern language including C#. In C#, we can use strings as array of characters. Strings are used for storing text.

It can be a character, a word or a long passage surrounded with the double quotes ". The following are string literals.

Example: String Literals

```
"S"
"String"
"This is a string."
```

Example

Create a variable of type string and assign it a value:

```
string greeting = "Hello";
```

There two ways to declare a string variable in C#. Using **System.String** class and using **string** keyword. Both are the same and make no difference. A string is represented by class *System.String*. The *"string" keyword* is an alias for System.String class and instead of writing System.String one can use *String* which is a shorthand for *System.String* class. So we can say string and String both can be used as an alias of System.String class. So string is an *object* of System.String class.

Example: String and string

```
string str1 = "Hello"; // uses string keyword
String str2 = "Hello"; // uses System.String class
```

Example:

```
 class Program
 {
   static void Main(string[] args)
   {
   string val = "Hello World";
    System.String val2 = "Hello World";
    System.Console.WriteLine(val);
    System.Console.WriteLine(val2);
   }
}
```

Special Characters

A text in the real world can include any character. In C#, because a string is surrounded with double quotes, it cannot include " in a string. The following will give a compile-time error.

Example: Invalid String

```
string text = "This is a "string" in C#.";
```

C# includes escaping character \ (backslash) before these special characters to include in a string.

Example: Escape Char \

```
string text = "This is a \"string\" in C#.";
```

Example

```
using System;
  class Program
  {
    static void Main(string[] args)
    {
    string val = "Hello \"Mr. Programmer\" in the world of programming";
    Console.WriteLine(val);
    }
 }
```

Output

```
Hello "Mr. Programmer" in the world of programming
```

<u>String Length</u>

A string in C# is actually an object, which contain properties and methods that can perform certain operations on strings. For example, the length of a string can be found with the Length property as: stringname.Length

Example

```
string alpha = "ABCDEFGHIJKLMNOPQRSTUVWXYZ";
Console.WriteLine("The length of the alpha string is: " + alpha.Length);
```

Output

```
26
```

<u>Other Methods</u>

There are many string methods available, for example ToUpper() and ToLower(), which returns a copy of the string converted to **uppercase** or **lowercase**:

Example

```
string txt = "Hello World";
Console.WriteLine(txt.ToUpper());   // Outputs "HELLO WORLD"
Console.WriteLine(txt.ToLower());   // Outputs "hello world"
```

String Concatenation

The + operator can be used between strings to combine them. This is called **concatenation**:

Example

```
string firstName = "Ashish ";
string lastName = "Sharma";
string name = firstName + lastName;
Console.WriteLine(name); // Outputs "Ashish Sharma"
```

We can also use the string.Concat() method to concatenate two strings:

Example

```
string firstName = "Ashish ";
string lastName = " Sharma ";
string name = string.Concat(firstName, lastName);
Console.WriteLine(name); // Outputs "Ashish Sharma"
```

Access Strings

We can access the characters in a string by referring to its index number inside square brackets [].

This example prints the **first character** in **myString**:

Example

```
string myString = "Hello";
Console.WriteLine(myString[0]);  // Outputs "H"
```

Example

```
string myString = "Hello";
Console.WriteLine(myString[1]);  // Outputs "e"
```

Exercise 6: Write a program in C# Sharp to input a string and print it.

```
using System;
class Exercise6
{
    public static void Main()
   {
        string str;
    Console.Write("Input the string : ");
    str= Console.ReadLine();
    Console.Write("The string you entered is : {0}\n", str);
        }
}
```

Output:

```
Input the string : Hello this is a string.
The string you entered is : Hello this is a string.
```

C# Methods

A **method** is a block of code which only runs when it is called. Or we can say a method is a group of statements that together perform a task. Every C# program has at least one class with a method named **Main**.

To use a method, we need to −

- **Define the method**
- **Call the method**

We can pass data, known as parameters, into a method.

Methods are used to perform certain actions, and they are also known as **functions**.

Use of methods- To reuse code: define the code once, and use it many times.

Defining Methods in C#

When we define a method, we basically declare the elements of its structure. The syntax for defining a method in C# is as follows −

```
<Access Specifier> <Return Type> <Method Name> (Parameter List)
{
  Method Body
}
```

Following are the various elements of a method −

- **Access Specifier** − This determines the visibility of a variable or a method from another class.
- **Return type** − A method may return a value. The return type is the data type of the value the method returns. If the method is not returning any values, then the return type is **void**. The void keyword, used in the examples above, indicates that the method should not return a value. If we want the method to **return** a value, we can use a primitive data type (such as **int or double**) instead of void, and use the **return keyword** inside the method.
- **Method name** − Method name is a unique identifier and it is case sensitive. It cannot be same as any other identifier declared in the class.
- **Parameter list** − Enclosed between parentheses, the parameters are used to pass and receive data from a method. The parameter list refers to the **type**, **order**, and **number of the parameters** of a method. Parameters are **optional**; that is, a method may contain no parameters.
- **Method body** − This contains the set of instructions needed to complete the required activity.

Example

Create a method (without parameters) inside the Program class:

```
class Program
{
  static void MyMethod()
  {
    // code to be executed
  }
}
```

Example Explained

- MyMethod() is the name of the method
- static means that the method belongs to the Program class and not an object of the Program class.
- void means that this method does not have a return value.

Call a Method

To call (execute) a method, write the method's name followed by two parentheses **()** and a semicolon**;**

In the following example, MyMethod() is used to print a text (the action), when it is called:

Example

Inside Main(), call the myMethod() method:

```
using System;
  class Program
  {
                static void MyMethod()
                {
                 Console.WriteLine("Hello");
                 }
              static void Main(string[] args)
            {
             MyMethod();
            }
}
// Outputs "Hello"
```

A method can be called multiple times:

Example

```
using System;
  class Program
  {
static void MyMethod()
{
  Console.WriteLine("Hello");
}
```

```
static void Main(string[] args)
{
  MyMethod();
  MyMethod();
  MyMethod();
}
Output
// Hello
// Hello
// Hello
```

C# Method Parameters

Parameters and Arguments

Information can be passed to methods as parameter. Parameters act as variables inside the method.

They are specified after the method name, inside the parentheses. We can add as many parameters as we want, just separate them with a comma.

The following example has a method that takes a string called **fname** as parameter. When the method is called, we pass along a first name, which is used inside the method to print the full name:

Example

```
using System;
class Program
{
static void MyMethod(string name)
{
  Console.WriteLine(name);
}
static void Main(string[] args)
{
  MyMethod("Mohan");
  MyMethod("Ashish");
  MyMethod("Ankit");
}
}
```

```
Output
// Mohan
// Ashish
// Ankit
```

When a **parameter** is passed to the method, it is called an **argument**. So, from the example above: name is a **parameter**, while Mohan, Ashish and Ankit are **arguments**.

Example 2

```
static void MyMethod(string fname, int age)
{
  Console.WriteLine(fname + " is " + age);
}
static void Main(string[] args)
{
  MyMethod("Liam", 5);
  MyMethod("Jenny", 8);
  MyMethod("Anja", 31);
}
Output
// Liam is 5
// Jenny is 8
// Anja is 31
```

Example 3

```
using System;
namespace MyApplication
{
class Program
  {
    static int add(int x, int y)
    {
      return x + y;
    }
    static void Main(string[] args)
    {
      Console.WriteLine(add(5, 3));
    }
  }
}
```

Output

```
8
```

Note that when we are working with multiple parameters, the method call must have the same number of arguments as there are parameters, and the arguments must be passed in the same order.

Exercise: Write a program in C# Sharp for the sum of two numbers (entered by user) by creating a function.

```
using System;
class funcexer3
{
    static int Sum(int num1, int num2)
    {
        int total;
        total = num1 + num2;
        return total;
    }
    public static void Main()
    {
      Console.WriteLine("Enter a number: ");
      int n1= Convert.ToInt32(Console.ReadLine());
      Console.WriteLine("Enter another number: ");
      int n2= Convert.ToInt32(Console.ReadLine());
      Console.WriteLine("The sum of two numbers is : {0} ", Sum(n1,n2) );
    }
}
```

C# Classes and Objects

C# is an object-oriented programming language.

Procedural programming is about writing procedures or methods that perform operations on the data, while object-oriented programming is about creating objects that contain both data and methods.

Object-oriented programming has several **advantages** over procedural programming:

- OOP is faster and easier to execute
- OOP provides a clear structure for the programs
- OOP helps to keep the C# code DRY **"Don't Repeat Yourself"**, and makes the code easier to maintain, modify and debug
- OOP makes it possible to create full reusable applications with less code and shorter development time

Classes and **objects** are the two main aspects of object-oriented programming.

Look at the following illustration to see the difference between class and objects:

Class	**objects**
Fruit	Apple
	Banana
	Mango
	Orange

So, a class is a **template** or **blueprint** for objects and an object is an **instance** of a class.

When the individual objects are created, they inherit all the variables and methods from the class.

Everything in C# is associated with classes and objects, along with its **attributes** and **methods**. For example: in real life, a car is an object. The car has **attributes**, such as **weight** and **color**, and **methods**, such as **drive** and **brake**.

Create a Class

To create a class, use the `class` keyword:

Create a class named "`Car`" with a variable `color`:

```
class Car
{
  string color = "red";
}
```

Create an Object

An object is created from a class. We have already created the class named `Car`, so now we can use this to create objects.

To create an object of `Car`, specify the class name, followed by the object name, and use the keyword `new`:

Example

Create an object called "`ford`" and use it to print the value of `color`:

```
class Car
{
  string color = "red";
  public static void Main(string[] args)
  {
    Car ford = new Car();
    Console.WriteLine(ford.color);
  }
}
```

Output

red

Example

Create two objects of `Car`:

```
class Car
{
  string color = "red";
  static void Main(string[] args)
  {
    Car myObj1 = new Car();
    Car myObj2 = new Car();
    Console.WriteLine(myObj1.color);
    Console.WriteLine(myObj2.color);
  }
}
```

Output

```
red
red
```

Class Members

Fields and **methods** inside classes are often referred to as "Class Members":

Example

Create a `Car` class with three class members: **two fields** and **one method**.

```
// The class
class MyClass
{
  // Class members
  string color = "red";        // field
  int maxSpeed = 200;          // field
  public void fullThrottle()   // method
  {
    Console.WriteLine("The car is going as fast as it can!");
  }
}
```

Fields

Variables inside a class are called fields, and that we can access them by creating an object of the class, and by using the dot syntax (`.`).

The following example will create an object of the `Car` class, with the name `myObj`. Then we print the value of the fields `color` and `maxSpeed`:

Example

```
class Car
{
  string color = "red";
  int maxSpeed = 200;
  static void Main(string[] args)
  {
    Car myObj = new Car();
    Console.WriteLine(myObj.color);
    Console.WriteLine(myObj.maxSpeed);
  }
}
```

Output

```
red
200
```

We can also leave the fields blank, and modify them when creating the object:

Example

```
class Car
{
  string color;
  int maxSpeed;
  static void Main(string[] args)
  {
    Car myObj = new Car();
    myObj.color = "red";
    myObj.maxSpeed = 200;
    Console.WriteLine(myObj.color);
    Console.WriteLine(myObj.maxSpeed);
  }
}
```

```
Output
red
200
```

This is especially useful when creating multiple objects of one class:

Example

```
class Car
{
  string model;
  string color;
  int year;

  static void Main(string[] args)
  {
    Car Ford = new Car();
    Ford.model = "Mustang";
    Ford.color = "red";
    Ford.year = 1969;

    Car Opel = new Car();
    Opel.model = "Astra";
    Opel.color = "white";
    Opel.year = 2005;

    Console.WriteLine(Ford.model);
    Console.WriteLine(Opel.model);
  }
}
```

```
Output
Mustang
Astra
```

Object Methods

Methods are used to perform certain actions.

Methods normally belong to a class, and they define how an object of a class behaves.

Just like with fields, we can access methods with the dot syntax. However, note that the method must be `public`. The reason is simple: a static method can be accessed without creating an object of the class, while public methods can only be accessed by objects. And remember that we use the name of the method followed by two parantheses `()` and a semicolon `;` to call (execute) the method:

Example

```
class Car
{
  string color;                 // field
  int maxSpeed;                 // field
  public void fullThrottle()    // method
  {
    Console.WriteLine("The car is going as fast as it can!");
  }

  static void Main(string[] args)
  {
    Car myObj = new Car();
    myObj.fullThrottle();  // Call the method
  }
}
```

Output

```
The car is going as fast as it can!
```

Example

```
using System;
class Student
{
 string course;
 int rollno;
 public void Performance()
 {
 Console.WriteLine("Hard work is the key of success");
 }

public static void Main()
{
Student Ram= new Student();
Console.WriteLine(Ram.course= "BCA");
Console.WriteLine(Ram.rollno=123);
Ram.Performance();

Student Shyam= new Student();
Console.WriteLine(Shyam.course= "MCA");
Console.WriteLine(Shyam.rollno=125);
Shyam.Performance();
}
}
```

C# - Inheritance

One of the most important concepts in object-oriented programming is **inheritance**. Inheritance allows us to **define a class in terms of another class**, which makes it easier to create and maintain an application. This also provides an opportunity to **reuse the code** functionality and speeds up implementation time. It is the mechanism in C# by which one class is allowed to inherit the features (fields and methods) of another class.

When creating a class, instead of writing completely new data members and member functions, the programmer can designate that the new class should inherit the members of an existing class. This existing class is called the **base** class, and the new class is referred to as the **derived** class.

That is, In C#, it is possible to inherit **fields** and **methods** from one class to another. We group the "inheritance concept" into two categories:

- **Derived Class (child)** - the class that inherits from another class
- **Base Class (parent)** - the class being inherited from

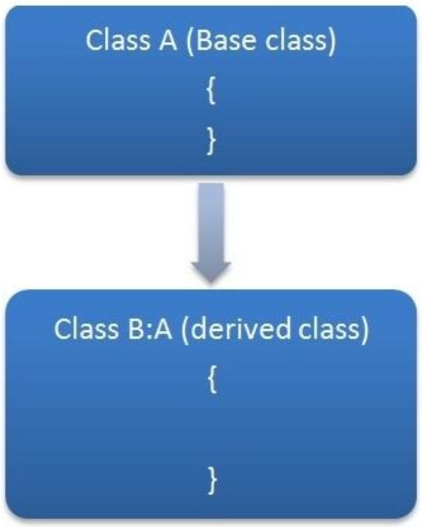
  To inherit from a class, use the **:** symbol.

Base and Derived Classes

A class can be derived from more than one class, which means that it can inherit **fields** and **methods** from multiple base classes.

The syntax used in C# for creating derived classes is as follows −

```
class base_class
{
  ...
}
class derived_class : base_class
 {
  ...
}
```



Let’s understand this by taking simple example of inheritance:

Example 1

```
using System;
 class myclass1
 {
   public int i =10;
 }

 class myclass2 : myclass1
 {
   public int a = 20;
 }

 class program
 {
 public static void Main()
 {
 myclass2 obj = new myclass2();
 Console.WriteLine(obj.i);
 Console.WriteLine(obj.a);
 Console.WriteLine(obj.i + obj.a);
 }
 }
```

Output

```
10
20
30
```

<u>Example2</u>

In the example below, the Car class (child) inherits the fields and methods from the Vehicle class (parent):

```
class Vehicle  // base class (parent)
{
  public string brand = "Ford";  // Vehicle field
  public void honk()             // Vehicle method
  {
    Console.WriteLine("Tuut, tuut!");
  }
}

class Car : Vehicle  // derived class (child)
{
  public string modelName = "Mustang";  // Car field
}

class Program
{
  static void Main(string[] args)
  {
    // Create a myCar object
    Car myCar = new Car();

    // Call the honk() method (From the Vehicle class) on the myCar object
    myCar.honk();

    // Display the value of the brand field (from the Vehicle class) and the value of the modelName from the Car class
    Console.WriteLine(myCar.brand + " " + myCar.modelName);
  }
}
```

Output

```
Tuut, tuut!
Ford Mustang
```

The sealed Keyword

If we don't want other classes to inherit from a class, use the sealed keyword:

If we try to access a sealed class, C# will generate an error:

```
sealed class Vehicle
{
  ...
}

class Car : Vehicle
{
  ...
}
```

The error message will be something like this:

```
'Car': cannot derive from sealed type 'Vehicle'
```

C# Polymorphism

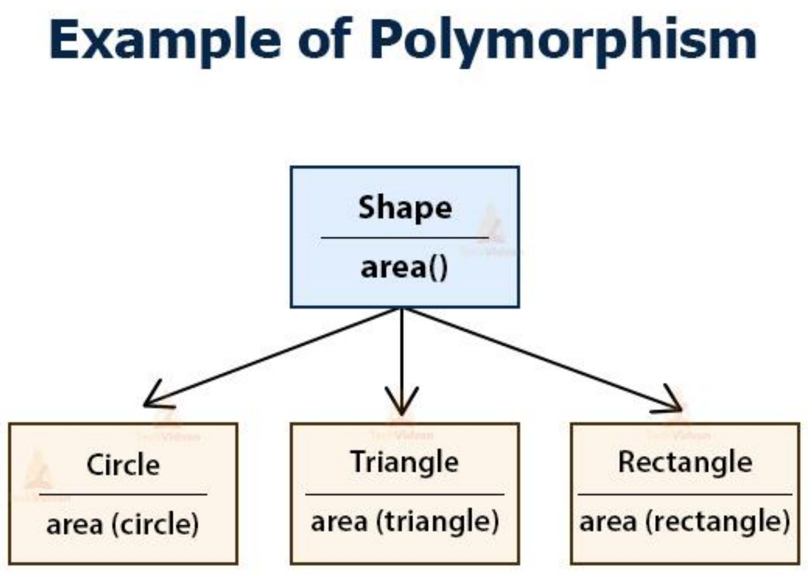
The term "Polymorphism" is the combination of "poly" + "morphs" which means many forms. It is a greek word. In object-oriented programming paradigm, polymorphism is often expressed as 'one interface, multiple functions'. Polymorphism is considered one of the important features of Object-Oriented Programming. Polymorphism allows us to perform a single action in different ways.

It occurs when we have many classes that are related to each other by inheritance.

Inheritance lets us inherit fields and methods from another class. **Polymorphism** uses those methods to perform different tasks. This allows us to perform a single action in different ways.

Real Life Example of Polymorphism: For example, you have a smartphone for communication. The communication mode you choose could be anything. It can be a call, a text message, a picture message, mail, etc. So, the goal is common that is communication, but their approach is different. This is called Polymorphism.

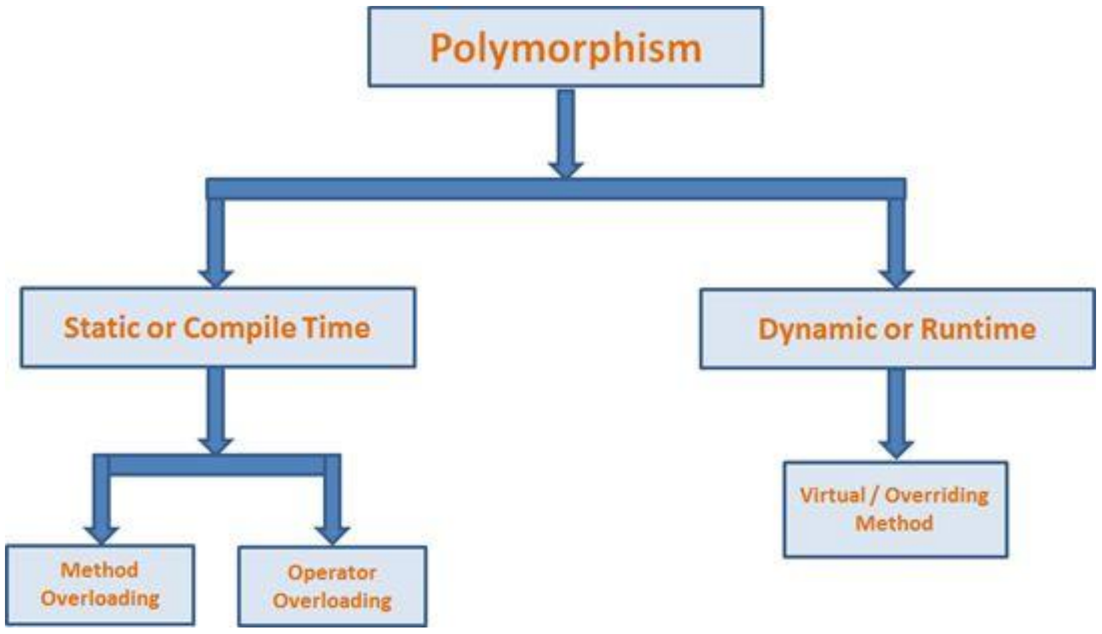
For example, think of a base class called Animal that has a method called animalSound(). Derived classes of Animals could be Pigs, Cats, Dogs, Birds - And they also have their own implementation of an animal sound (the pig oinks, and the cat meows, etc.):



Types of Polymorphism

There are two types of polymorphism in C#:

- **Static / Compile Time Polymorphism.**
- **Dynamic / Runtime Polymorphism.**



<u>Static or Compile Time Polymorphism</u>

<u>Method overloading</u>

It is also known as **Early Binding**. **Method overloading** is an example of Static Polymorphism. In **overloading**, the method / function has a **same name** but **different arguments**. It is also known as Compile Time Polymorphism because the decision of which method is to be called is made at compile time. Overloading is the concept in which method names are the same with a different set of parameters (may be different no of parameters, may be different types of parameters).

Here C# compiler checks the number of parameters passed and the type of parameter and make the decision of which method to call and it throw an error if no matching method is found.

In the following example, a class has two methods with the same name "Add" but with different input parameters (the first method has two parameters and the second method has three parameters).

<u>Example</u>

```
 using System;
 class myclass1
{
  public int ADD(int a, int b)
  {
  int c= a+b;
  return c;
  }
}
```

```
class myclass2 : myclass1
{
  public int ADD(int a, int b, int c)
  {
  int d= a+b+c;
  return d;
  }
}
class program
{
public static void Main()
{
myclass2 obj = new myclass2();
Console.WriteLine(obj.ADD(5,6));
Console.WriteLine(obj.ADD(5,6,6));
}
}
```

Operator Overloading

The concept of overloading a function can also be applied to operators. Operator overloading gives the ability to use the same operator to do various operations. It provides additional capabilities to C# operators when they are applied to user-defined data types. The function of the operator is declared by using the **operator keyword**.

Syntax :

```
access specifier  className  operator Operator_symbol (parameters)
{
   // Code
}
```

For example, we can overload an operator '+' in a class like String so that we can concatenate two strings by just using +.

Dynamic / Runtime Polymorphism

Dynamic / runtime polymorphism is also known as **late binding**. Here, the method name and the method signature (number of parameters and parameter type must be the same and may have a different implementation). Method overriding is an example of dynamic polymorphism.

To perform method overriding in C#, we need to use **virtual** keyword with base class method and **override** keyword with derived class method.

In simple words, Overriding is a feature that allows a subclass or child class to provide a specific implementation of a method that is already provided by one of its super-classes or parent classes. When a method in a subclass has the same name, same parameters or signature and same return type (or sub-type) as a method in its super-class, then the method in the subclass is said to override the method in the super-class. Method overriding is one of the ways by which C# achieve Run Time Polymorphism (Dynamic Polymorphism).

In C# we can use 2 types of keywords for Method Overriding:

- **virtual keyword**: This modifier or keyword use within base class method. It is used to modify a method in base class for overridden that particular method in the derived class.
- **override keyword**: This modifier or keyword use with derived class method. It is used to modify a virtual or abstract method into derived class which presents in base class.

<u>Example </u>–(See the difference between below two programs- working of virtual and override keyword.)

```
using System;
class parent
{
 public void method()
 {
 Console.WriteLine("This is a parent class method.");
 }
}

class child : parent
 {
 public  void method()
 {
 Console.WriteLine("This is child class method.");
 }
}
 class program
 {
 public static void Main()
 {
```

```
 parent obj1= new parent();
 parent obj2 = new child();
 obj1.method();
 obj2.method();
 }
}
```

Output

```
This is a parent class method.
This is a parent class method.
```

(Same program with virtual and override keyword)

```
using System;
 class parent
 {
  public virtual void method()
  {
  Console.WriteLine("This is a parent class method.");
  }
 }
 class child : parent
  {
  public override void method()
  {
  Console.WriteLine("This is child class method.");
  }
 }
  class program
  {
  public static void Main()
  {
  parent obj1= new parent();
  parent obj2 = new child();
  obj1.method();
  obj2.method();
```

```
            }
          }
```

Output

```
This is a parent class method.
This is a child class method.
```

Another Example

```
using System;
  class Animal  // Base class (parent)
  {
    public void animalSound()
    {
      Console.WriteLine("The animal makes a sound");
    }
  }

  class Cat : Animal  // Derived class (child)
  {
    public void animalSound()
    {
      Console.WriteLine("The cat says: meow meow");
    }
  }

  class Dog : Animal  // Derived class (child)
  {
    public void animalSound()
    {
      Console.WriteLine("The dog says: bow wow");
    }
  }

  class Program
  {
    static void Main(string[] args)
    {
      Animal myAnimal = new Animal();  // Create a Animal object
      Animal myCat = new Cat();  // Create a Cat object
      Animal myDog = new Dog();  // Create a Dog object

      myAnimal.animalSound();
      myCat.animalSound();
      myDog.animalSound();
    }
  }
```

Output

```
The animal makes a sound
The animal makes a sound
The animal makes a sound
```

(Same program with virtual and override keyword)

```
using System;
 class Animal  // Base class (parent)
 {
   public virtual void animalSound()
   {
     Console.WriteLine("The animal makes a sound");
   }
 }

 class Cat : Animal  // Derived class (child)
 {
   public override void animalSound()
   {
     Console.WriteLine("The cat says: meow meow");
   }
 }
 class Dog : Animal  // Derived class (child)
 {
   public override void animalSound()
   {
     Console.WriteLine("The dog says: bow wow");
   }
 }

 class Program
 {
   static void Main(string[] args)
   {
     Animal myAnimal = new Animal();  // Create a Animal object
     Animal myCat = new Cat();  // Create a Cat object
     Animal myDog = new Dog();  // Create a Dog object

     myAnimal.animalSound();
     myCat.animalSound();
     myDog.animalSound();
   }
 }
```

Output

```
The animal makes a sound
The cat says: meow meow
The dog says: bow wow
```

Example

```
 using System;
 class Animal
 {
     public virtual void eat()
    {
         Console.WriteLine("Eating...");
     }
 }
```

```
    class Dog: Animal
    {
        public override void eat()
        {
            Console.WriteLine("Eating bones...because dog loves to eat bones.");
        }
    }
    public class Overriding
    {
        public static void Main()
        {
            Animal myanimal = new Animal();
            Animal tommy = new Dog();
            tommy.eat();
        }
    }
```

Output:

```
Eating bones...because dog loves to eat bones.
```

C# Interface

Data **abstraction** is the process of hiding certain details and showing only essential information to the user. Abstraction can be achieved with either **abstract classes** or **interfaces**.

Interface in C# is a **blueprint** of a **class**. Like a class, Interface can have **methods**, **properties** and **events** as its members. But interfaces will contain **only the declaration** of the members. The implementation of the interface's members will be given by class who implements the interface implicitly or explicitly.

- Interfaces specify **what a class must do, not how**.
- Interfaces **can't have private members**.
- By default all the members of Interface are **public**.
- The interface will always defined with the help of **keyword 'interface'.**
- Interface **cannot contain fields**.
- **Multiple inheritances** are possible with the help of Interfaces, which can't be achieved by class.
- Its implementation must be provided by class.

Syntax for Interface Declaration:

```
interface  <interface_name >
```

```
{
   // declare Events
   // declare indexers
   // declare methods
   // declare properties
}
```

Syntax for Implementing Interface:

```
class class_name : interface_name
```

To declare an interface, use ***interface*** keyword. It is used to provide total abstraction. That means all the members in the interface are declared with the empty body and are public and abstract by default. A class that implements interface must implement all the methods declared in the interface.

Example 1:

```
using System;
interface interface_exmp
{
   void display();
}
class program : interface_exmp
{
   public void display()
   {
      Console.WriteLine("Implementing Interface");
   }
   public static void Main (String []args)
   {
      program obj = new program();
      obj.display();
   }
}
```

Output:

```
Implementing Interface
```

Example 2

```
using System;
  interface IAnimal
  {
    void animalSound(); // interface method (does not have a body)
  }
  // Pig "implements" the IAnimal interface
  class Pig : IAnimal
  {
    public void animalSound()
    {
      // The body of animalSound() is provided here
      Console.WriteLine("The pig says: wee wee");
    }
  }
  class Program
  {
    static void Main(string[] args)
    {
      Pig myPig = new Pig();  // Create a Pig object
      myPig.animalSound();
    }
}
```

Output:

```
The pig says: wee wee
```

using System;

Example 3

```
interface Drawable
{
    void draw();
}

class Rectangle : Drawable
```

```
{
    public void draw()
    {
        Console.WriteLine("drawing rectangle...");
    }
}

class Circle : Drawable
{
    public void draw()
    {
        Console.WriteLine("drawing circle...");
    }
}

public class TestInterface
{
    public static void Main()
    {
        Drawable d1 = new Rectangle();
        Drawable d2 = new Circle();
        d1.draw();
        d2.draw();
    }
}
```

Note:

- Interfaces **cannot** be used to **create objects** (in the example above, it is not possible to create an "IAnimal" object in the Program class)
- Interface methods **do not have a body** - the body is provided by the "implement" class
- On implementation of an interface, we must override all of its methods
- Interfaces can contain properties and methods, but not fields/variables
- Interface members are by default abstract and public
- An interface cannot contain a constructor (as it cannot be used to create objects)

Why And When To Use Interfaces?

1) **To achieve security** - hide certain details and only show the important details of an object (interface).

2) C# does not support "**multiple inheritance**" (a class can only inherit from one base class). However, it can be achieved with interfaces, because the class can **implement** multiple interfaces. **Note:** To implement multiple interfaces, separate them with a comma (see example below).

Multiple Interfaces

To implement multiple interfaces, separate them with a comma:

Example

```
interface IFirstInterface
{
  void myMethod(); // interface method
}
interface ISecondInterface
{
  void myOtherMethod(); // interface method
}
// Implement multiple interfaces
class DemoClass : IFirstInterface, ISecondInterface
{
  public void myMethod()
  {
    Console.WriteLine("Some text..");
  }
  public void myOtherMethod()
  {
    Console.WriteLine("Some other text...");
  }
}
class Program
{
  static void Main(string[] args)
  {
    DemoClass myObj = new DemoClass();
    myObj.myMethod();
    myObj.myOtherMethod();
  }
}
```

Output:

```
Some text..
Some other text...
```

C# Delegates and Events

C# delegates are similar to pointers to functions, in C or C++. A **delegate** is a **reference type** variable that holds the reference to a method. The reference can be changed at runtime.

It provides a way which tells which method is to be called when an event is triggered. **For example**, if we click a *Button* on a form (Windows Form application), the program would call a specific method. In simple words, it is a type that represents references to methods with a particular parameter list and return type and then calls the method in a program for execution when it is needed.

Delegates are especially used for implementing events and the call-back methods. All delegates are implicitly derived from the **System.Delegate** class.

So, we can use- delegate to pass a function as a parameter and to handles the callback functions or event handler. The best use of delegate is to use as event.

The delegate defines the method signature. We can define variables of delegate, just like other data type, that can refer to any method with the same signature as the delegate.

Note: A delegate will call only a method which agrees with its signature and return type. A method can be a static method associated with a class or can be an instance method associated with an object, it doesn't matter.

Important Points about Delegates:

- Provides a good way to encapsulate the methods.
- Delegates are the library class in System namespace.
- These are the type-safe pointer of any method.
- Delegates are mainly used in implementing the call-back methods and events.
- It doesn't care about the class of the object that it references.
- They have a signature and a return type. A function that is added to delegates must be compatible with this signature.
- Delegates can point to either static or instance methods.
- Once a delegate object has been created, it may dynamically invoke the methods it points to at runtime.

There are three steps involved while working with delegates:

1. **Declare a delegate**
2. **Set a target method**
3. **Invoke a delegate**

A delegate can be declared using the delegate keyword followed by a function signature, as shown below.

Delegate Syntax

```
[access modifier] delegate [return type] [delegate name]([parameters])
```

The following declares a delegate named MyDelegate.

Example: Declare a Delegate

```
public delegate void MyDelegate(string msg);
```

Above, we have declared a delegate MyDelegate with a void return type and a string parameter. A delegate can be declared outside of the class or inside the class. Practically, it should be declared out of the class.

After declaring a delegate, we need to set the target method or a lambda expression. We can do it by creating an object of the delegate using the new keyword and passing a method whose signature matches the delegate signature.

Example: Set Delegate Target

```
public delegate void MyDelegate(string msg); // declare a delegate
// set target method
MyDelegate del = new MyDelegate(MethodA);
// or
MyDelegate del = MethodA;
// or set lambda expression
MyDelegate del = (string msg) =>  Console.WriteLine(msg);

// target method
static void MethodA(string message)
{
   Console.WriteLine(message);
}
```

We can set the target method by assigning a method directly without creating an object of delegate e.g., MyDelegate del = MethodA.

After setting a target method, a delegate can be invoked using the Invoke() method or using the () operator.

Example: Invoke a Delegate

```
del.Invoke("Hello World!");
```

```
// or
del("Hello World!");
```

The following is a simple example of a delegate.

Example 1: Delegate

```
using System;
public delegate void del_display();
class program
{
   public void display()
   {
   Console.WriteLine("Invoking Delegates");
   }
   public static void Main(String []args)
   {
            program obj = new program();
            del_display delobj = new del_display(obj.display);
            delobj();
            // we can write it like
            // delobj.Invoke();
   }
}
```

Output

```
 Invoking Delegates
```

Example 2: Delegate

```
using System;
public delegate void addnum(int a, int b);
public delegate void subnum(int a, int b);
class program
{
   public void sum(int a, int b)
   {
       Console.WriteLine("({0} + {1}) = {2}", a,b,a+b);
   }

   public void subtract(int a, int b)
   {
       Console.WriteLine("({0} - {1}) = {2}", a,b,a-b);
   }

  public static void Main(String []args)
  {

           program obj = new program();
           addnum del_obj1 = new addnum(obj.sum);
           subnum del_obj2 = new subnum(obj.subtract);
   del_obj1(100, 10);
   del_obj2(100, 10);
   // These can be written as using
   // "Invoke" method
   // del_obj1.Invoke(100, 40);
   // del_obj2.Invoke(100, 60);
}
}
```

Output

```
 (100 + 10) = 110
 (100 - 10) = 90
```

Example 3: Delegate

```
using System;
delegate int Calculator(int n);//declaring delegate

public class DelegateExample
{
  static int number = 100;
```

```
    public static int add(int n)
    {
        number = number + n;
        return number;
    }
    public static int mul(int n)
    {
        number = number * n;
        return number;
    }
    public static int getNumber()
    {
        return number;
    }
    public static void Main(string[] args)
    {
        Calculator c1 = new Calculator(add);//instantiating delegate
        Calculator c2 = new Calculator(mul);
        c1(20);//calling method using delegate
        Console.WriteLine("After c1 delegate, Number is: " + getNumber());
        c2(3);
        Console.WriteLine("After c2 delegate, Number is: " + getNumber());
    }
}
```

Output:

```
After c1 delegate, Number is: 120
After c2 delegate, Number is: 360
```

Unit-3

.Net Framework using C#

C# Namespaces

Namespaces are used to organize the classes. It helps to control the scope of methods and classes in larger .Net programming projects. In simpler words we can say that it provides a way to keep one set of names (like class names) different from other sets of names.

The biggest advantage of using namespace is that the class names which are declared in one namespace will not clash with the same class names declared in another namespace. It is also referred as **named group of classes** having common features. The members of a namespace can be **namespaces, interfaces, structures, and delegates.**

Defining a Namespace

To define a namespace in C#, we will use the **namespace** keyword followed by the name of the namespace and curly braces containing the body of the namespace as follows:

Syntax:

```
namespace name_of_namespace
{
// Namespace (Nested Namespaces)
// Classes
// Interfaces
// Structures
// Delegates
}
```

Example:

```
namespace name1
{
   // C1 is the class in the namespace name1
   class C1
   {
   }
}
```

Accessing the Members of Namespace

The members of a namespace are accessed by using dot (.) operator. A class in C# is fully known by its respective namespace.

Syntax:

[namespace_name].[member_name]

Note:

- Two classes with the same name can be created inside 2 different namespaces in a single program.
- Inside a namespace, no two classes can have the same name.
- In C#, the full name of the class starts from its namespace name followed by dot (.) *operator* and the class name, which is termed as the fully qualified name of the class.

Example:

```
using System;
namespace myspace
{
    class firstclass
    {
        public static void display()
        {
            System.Console.WriteLine("Hello World!");
        }
    }
        /* Removing comment will give the error
       because no two classes can have the
       same name under a single namespace
    class firstclass
    {
        public static void display()
        {
            System.Console.WriteLine("Hello World!");
        }
    }
    */
}
```

```
    class firstclass
    {
       public static void Main(String []args)
       {
          myspace.firstclass.display();
       }
  }
```

Output:

```
Hello World!
```

In the above example:

- In **System.Console.WriteLine()**" "*System*" is a namespace in which we have a class named "*Console*" whose method is "*WriteLine()*".
- It is not necessary to keep each class in C# within Namespace but we do it to organize our code well.
- Here "**.**" is the delimiter used to separate the class name from the namespace and function name from the class name.

<u>Example 2</u>

```
using System;
namespace First
{
    public class Hello
    {
    public void sayHello()
    {
    Console.WriteLine("Hello First Namespace");
    }
    }
}
namespace Second
{
   public class Hello
   {
          public void sayHello()
          {
          Console.WriteLine("Hello Second Namespace");
          }
      }
  }
  public class TestNamespace
  {
      public static void Main()
      {
          First.Hello h1 = new First.Hello();
          Second.Hello h2 = new Second.Hello();
          h1.sayHello();
          h2.sayHello();

      }
  }
```

Output

```
Hello First Namespace
Hello Second Namespace
```

Namespace: System

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.

Classes

AccessViolationException	The exception that is thrown when there is an attempt to read or write protected memory.
Activator	Contains methods to create types of objects locally or remotely, or obtain references to existing remote objects. This class cannot be inherited.
AggregateException	Represents one or more errors that occur during application execution.
AppContext	Provides members for setting and retrieving data about an

	application's context.
ArgumentOutOfRangeException	The exception that is thrown when the value of an argument is outside the allowable range of values as defined by the invoked method.
ArithmeticException	The exception that is thrown for errors in an arithmetic, casting, or conversion operation.
Array	Provides methods for creating, manipulating, searching, and sorting arrays, thereby serving as the base class for all arrays in the common language runtime.
ArrayTypeMismatchException	The exception that is thrown when an attempt is made to store an element of the wrong type within an array.
AssemblyLoadEventArgs	Provides data for the AssemblyLoad event.
AttributeUsageAttribute	Specifies the usage of another attribute class. This class cannot be inherited.
BadImageFormatException	The exception that is thrown when the file image of a dynamic link library (DLL) or an executable program is invalid.
BitConverter	Converts base data types to an array of bytes, and an array of bytes to base data types.
Buffer	Manipulates arrays of primitive types.
CannotUnloadAppDomainException	The exception that is thrown when an attempt to unload an application domain fails.
CharEnumerator	Supports iterating over a String object and reading its individual characters. This class cannot be inherited.
CLSCompliantAttribute	Indicates whether a program element is compliant with the Common Language Specification (CLS). This class cannot be inherited.
Console	Represents the standard input, output, and error streams for console applications. This class cannot be inherited.

C# Input output Classes

The **System.IO** namespace has various classes that are used for performing numerous operations with files, such as creating and deleting files, reading from or writing to a file, closing a file etc.

The following table shows some commonly used non-abstract classes in the System.IO namespace −

Sr. No.	I/O Class & Description
1	**BinaryReader** Reads primitive data from a binary stream.
2	**BinaryWriter** Writes primitive data in binary format.
3	**BufferedStream** A temporary storage for a stream of bytes.
4	**Directory** Helps in manipulating a directory structure.
5	**DirectoryInfo** Used for performing operations on directories.
6	**DriveInfo** Provides information for the drives.
7	**File** Helps in manipulating files.
8	**FileInfo** Used for performing operations on files.
9	**FileStream** Used to read from and write to any location in a file.
10	**MemoryStream** Used for random access to streamed data stored in memory.
11	**Path** Performs operations on path information.
12	**StreamReader**

	Used for reading characters from a byte stream.
13	**StreamWriter** Is used for writing characters to a stream.
14	**StringReader** Is used for reading from a string buffer.
15	**StringWriter** Is used for writing into a string buffer.

A **file** is a collection of data stored in a disk with a specific name and a directory path. When a file is opened for reading or writing, it becomes a **stream**.

The stream is basically the sequence of bytes passing through the communication path. There are two main streams: the **input stream** and the **output stream**. The **input stream** is used for reading data from file (read operation) and the **output stream** is used for writing into the file (write operation).

PROGRAMMING EXAMPLE

In this programming Example we will create a new file **"CsharpFile.txt"** and saves it on disk. And then we will open this file, saves some text in it and then close this file.

CREATE A BLANK .TXT FILE USING FILESTREAM

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;
using System.IO;

namespace FileStream_CreateFile
{
    class Program
    {
        static void Main(string[] args)
        {
                FileStream fs = new FileStream("D:\\csharpfile.txt", FileMode.Create);
                fs.Close();
                Console.Write("File has been created and the Path is D:\\csharpfile.txt");
                Console.ReadKey();
            }
        }
    }
```

Output

```
File has been created and the Path is D:\\csharpfile.txt


_
```

Explanation:

In the above program I added **System.IO** namespace so that I could use FileStream class in my program. Then I created an object of **FileStream** class **fs** to create a new **csharpfile.txt** in D drive.

OPEN CSHARPFILE.TXT AND WRITE SOME TEXT IN IT

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;
using System.IO;

namespace AccessFile
{
    class Program
    {
        static void Main(string[] args)
        {
            FileStream fs = new FileStream("D:\\csharpfile.txt", FileMode.Append);
            byte[] bdata=Encoding.Default.GetBytes("Hello File Handling!");
            fs.Write(bdata, 0, bdata.Length);
            fs.Close();
            Console.WriteLine("Successfully saved file with data : Hello File Handling!");
            Console.ReadKey();
        }
    }
}
```

Output

```
Successfully saved file with data : Hello File Handling!
_
```

Explanation

In the above program again I created object as **fs** of **FileStrem** class. Then Encoded a string into bytes and kept into **byte[]** variable bdata and finally using **Write()** method of FileStream stored string into file.

READ DATA FROM CSHARPFILE.TXT FILE

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;
using System.IO;

namespace FileStream_ReadFile
{
    class Program
    {
        static void Main(string[] args)
        {
            string data;
            FileStream fsSource = new FileStream("D:\\csharpfile.txt", FileMode.Open, FileAccess.Read);
            using (StreamReader sr = new StreamReader(fsSource))
            {
                data = sr.ReadToEnd();
            }
            Console.WriteLine(data);
            Console.ReadLine();
        }
    }
}
```

Output

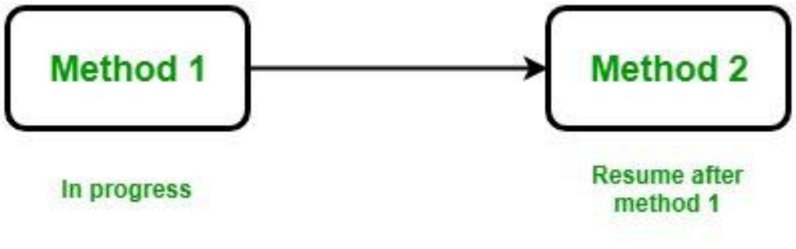
```
Hello File Handling!
```

C# Multithreading

Multitasking is the **simultaneous execution** of **multiple tasks** or processes over a certain time interval. Windows operating system is an example of multitasking because it is capable of running more than one process at a time like running Google Chrome, Notepad, VLC player, etc. at the same time. The operating system uses a term known as a ***process*** to execute all these applications at the same time. A process is a part of an operating system that is responsible for executing an application. Every program that executes on your system is a process and to run the code inside the application a process uses a term known as a ***thread***.

A thread is a lightweight process, or in other words, a thread is a unit which executes the code under the program. So every program has logic and a thread is responsible for executing this logic. Every program by default carries one thread to execute the logic of the program and the thread is known as the ***Main Thread***, so every program or application is by default **single-threaded model**. This single-threaded model has a drawback. The single thread runs all the process present in the program in synchronizing manner, means one after another. So, the second process waits until the first process completes its execution, it consumes more time in processing.

For example, we have a class named as *program* and this class contains two different methods, i.e *method1*, *method2*. Now the main thread is responsible for executing all these methods, so the main thread executes all these methods one by one.



So, **“Multithreading in C# is a process in which multiple threads work simultaneously. It is a process to achieve multitasking. It saves time because multiple tasks are being executed at a time. To create multithreaded application in C#, we need to use System.Threding namespace.”**

System.Threading Namespace

The System.Threading namespace contains classes and interfaces to provide the facility of multithreaded programming. It also provides classes to synchronize the thread resource. A list of commonly used **classes** are given below:

- Thread
- Mutex
- Timer
- Monitor
- Semaphore
- ThreadLocal
- ThreadPool
- Volatile etc.

Process and Thread

A process represents an application whereas a thread represents a module of the application. Process is heavyweight component whereas thread is lightweight. A thread can be termed as lightweight subprocess because it is executed inside a process.

Whenever you create a process, a separate memory area is occupied. But threads share a common memory area.

C# Thread Life Cycle

In C#, each thread has a life cycle. The life cycle of a thread is started when instance of *System.Threading.Thread class* is created. When the task execution of the thread is completed, its life cycle is ended.

There are following states in the life cycle of a Thread in C#.

- Unstarted
- Runnable (Ready to run)
- Running
- Not Runnable
- Dead (Terminated)

Unstarted State

When the instance of Thread class is created, it is in unstarted state by default.

Runnable State

When start() method on the thread is called, it is in runnable or ready to run state.

Running State

Only one thread within a process can be executed at a time. At the time of execution, thread is in running state.

Not Runnable State

The thread is in not runnable state, if sleep() or wait() method is called on the thread, or input/output operation is blocked.

Dead State

After completing the task, thread enters into dead or terminated state.

C# Thread class

C# Thread class provides properties and methods to create and control threads. It is found in **System.Threading** namespace.

C# Thread Properties

A list of important properties of Thread class are given below:

Property	**Description**
CurrentThread	returns the instance of currently running thread.
IsAlive	checks whether the current thread is alive or not. It is used to find the execution status of the thread.
IsBackground	is used to get or set value whether current thread is in background or not.
ManagedThreadId	is used to get unique id for the current managed thread.
Name	is used to get or set the name of the current thread.
Priority	is used to get or set the priority of the current thread.
ThreadState	is used to return a value representing the thread state.

C# Thread Methods

A list of important methods of Thread class are given below:

Method	**Description**
Abort()	is used to terminate the thread. It raises ThreadAbortException.
Interrupt()	is used to interrupt a thread which is in *WaitSleepJoin* state.

Join()	is used to block all the calling threads until this thread terminates.
ResetAbort()	is used to cancel the Abort request for the current thread.
Resume()	is used to resume the suspended thread. It is obselete.
Sleep(Int32)	is used to suspend the current thread for the specified milliseconds.
Start()	changes the current state of the thread to Runnable.
Suspend()	suspends the current thread if it is not suspended. It is obselete.
Yield()	is used to yield the execution of current thread to another thread.

C# Main Thread Example

The first thread which is created inside a process is called **Main thread**. It starts first and ends at last.

Let's see an example of Main thread in C#.

```
using System;
using System.Threading;
public class ThreadExample
{
    public static void Main(string[] args)
    {
        Thread t = Thread.CurrentThread;
        t.Name = "MainThread";
        Console.WriteLine(t.Name);
    }
 }
```

Output:

```
MainThread
```

C# Threading Example: static method

We can call static and non-static methods on the execution of the thread. To call the static and non-static methods, we need to pass method name in the constructor of ThreadStart class. For static method, we don't need to create the instance of the class. You can refer it by the name of class.

Example:

```
using System;
using System.Threading;
public class MyThread
{
   public static void Thread1()
   {
      for (int i = 0; i < 10; i++)
      {
         Console.WriteLine(i);
      }
   }
}
public class ThreadExample
{
   public static void Main()
   {
      Thread t1 = new Thread(new ThreadStart(MyThread.Thread1));
      Thread t2 = new Thread(new ThreadStart(MyThread.Thread1));
      t1.Start();
      t2.Start();
   }
}
```

Output:

The output of the above program can be anything because there is context switching between the threads.

```
0
1
2
3
4
0
1
2
3
4
```

C# Threading Example: non-static method

For non-static method, you need to create instance of the class so that you can refer it in the constructor of ThreadStart class.

ADVERTISEMENT

```
using System;
using System.Threading;
public class MyThread
{
    public void Thread1()
    {
        for (int i = 0; i < 5; i++)
        {
            Console.WriteLine(i);
        }
    }
}
public class ThreadExample
{
    public static void Main()
    {
        MyThread mt = new MyThread();
        Thread t1 = new Thread(new ThreadStart(mt.Thread1));
        Thread t2 = new Thread(new ThreadStart(mt.Thread1));
        t1.Start();
        t2.Start();
    }
}
```

Output:

```
0
1
2
3
4
```

```
0
1
2
3
4
```

C# Threading Example: performing different tasks on each thread

Let's see an example where we are executing different methods on each thread.

```
using System;
using System.Threading;
public class MyThread
{
    public static void Thread1()
    {
        Console.WriteLine("task one");
    }
    public static void Thread2()
    {
        Console.WriteLine("task two");
    }
}
public class ThreadExample
{
    public static void Main()
    {
        Thread t1 = new Thread(new ThreadStart(MyThread.Thread1));
        Thread t2 = new Thread(new ThreadStart(MyThread.Thread2));
        t1.Start();
        t2.Start();
    }
}
```

Output:

```
task one
task two
```

C# Threading Example: Sleep() method

The Sleep() method suspends the current thread for the specified milliseconds. So, other threads get the chance to start execution.

Example:

```
using System;
using System.Threading;
public class mythread
{
    // static method one
    public static void method1()
    {
        // It prints numbers from 0 to 10
        for (int I = 0; I <= 10; I++)
            {
            Console.WriteLine("Method1 is : {0}", I);
            // When the value of I is equal to 5 then
            // this method sleeps for 6 seconds
             if (I == 5)
            {
                Thread.Sleep(6000);
             }
        }
    }
    // static method two
    public static void method2()
    {
        // It prints numbers from 0 to 10
        for (int J = 0; J <= 5; J++)
            {
            Console.WriteLine("Method2 is : {0}", J);
        }
    }
    // Main Method
    static public void Main()
    {
        // Creating and initializing threads
        Thread thr1 = new Thread(method1);
        Thread thr2 = new Thread(method2);
        thr1.Start();
        thr2.Start();
    }}
```

Output :

```
Method1 is : 0
Method1 is : 1
Method1 is : 2
Method1 is : 3
Method1 is : 4
Method1 is : 5
Method2 is : 0
Method2 is : 1
Method2 is : 2
Method2 is : 3
Method2 is : 4
Method2 is : 5
Method1 is : 6
Method1 is : 7
Method1 is : 8
Method1 is : 9
Method1 is : 10
```

Explanation: Here, we create and initialize two threads, i.e *thr1* and *thr2* using Thread class. Now using thr1.Start(); and thr2.Start(); we start the execution of both the threads. Now both thread runs simultaneously and the processing of *thr2* does not depend upon the processing of *thr1* like in the single threaded model.

Note: Output may vary due to context switching.

Advantages of Multithreading:

- It executes multiple processes simultaneously.
- Maximize the utilization of CPU resources.
- Time sharing between multiple processes.

Networking and Socket Programming in C#

Network Programming involves writing programs that communicate with other programs across a computer **network**. The .NET framework provides two namespaces, **System.Net** and **System.Net.Sockets** for network programming. The classes and methods of these namespaces help us to write programs, which can communicate across the network.

The .NET Framework has a layered, extensible, and managed implementation of networking services. We can easily integrate them into our applications. The **System.Net** namespace provides a simple programming interface for many of the protocols used on networks today.

C# Uri

Uri provides an object representation of a uniform resource identifier (URI) and easy access to the parts of the URI.

C# UriBuilder

UriBuilder provides a convenient way to modify the contents of a Uri instance without creating a new Uri instance for each modification.

C# HttpWebRequest

HttpWebRequest is used to create an HTTP request. The resource is specified with the Uri.

C# hostname

The Dns.GetHostName method gets the host name of the local computer.

C# GetHostEntry

With the Dns.GetHostEntry method, we can determine an IP address from the hostname.

C# Ping

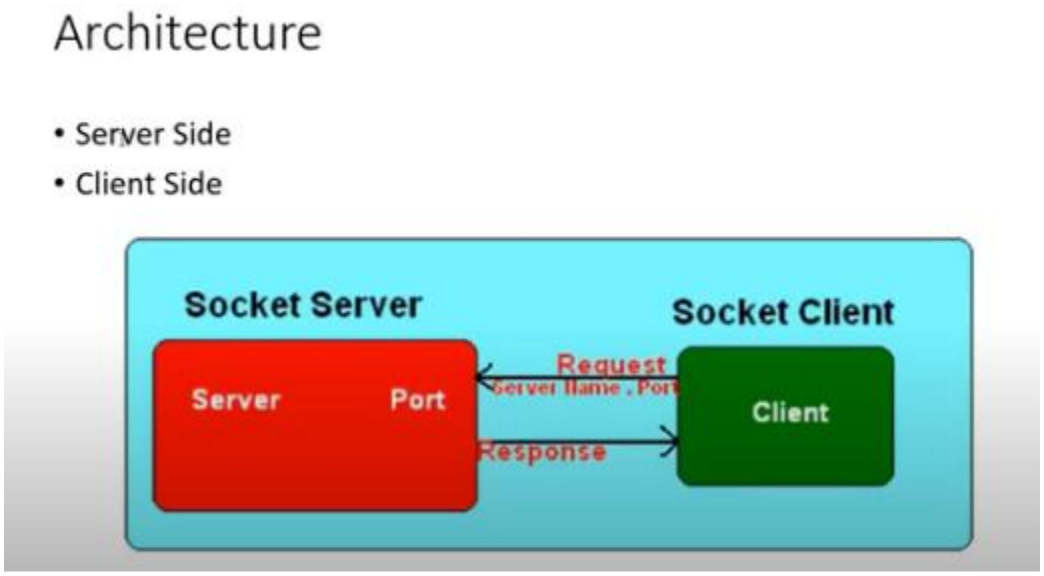
Ping is a network administration utility used to test the availability of a host on an Internet Protocol (IP) network. Ping works by sending Internet Control Message Protocol (ICMP) echo request packets to the target host and waits for an ICMP echo reply. The program reports errors, packet loss, and a statistical summary of the results.

.NET contains the Ping class for sending ping requests. The Ping class uses instances of the PingReply class to return information about the operation and receive the reply.

C# Socket

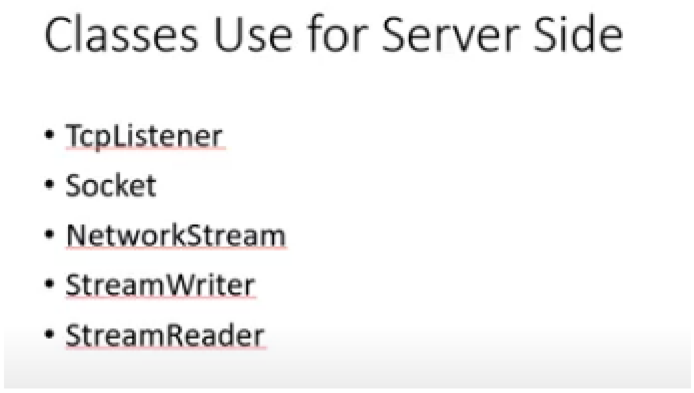
In programming, a *socket* is an endpoint of a communication between two programs running on a network. Sockets are used to create a connection between a client program and a server program.

Sockets in computer networks are used to establish a connection between two or more computers and used to send data from one computer to another. Each computer in the network is called a node. Sockets use nodes' IP addresses and a network protocol to create a secure channel of communication and use this channel to transfer data.



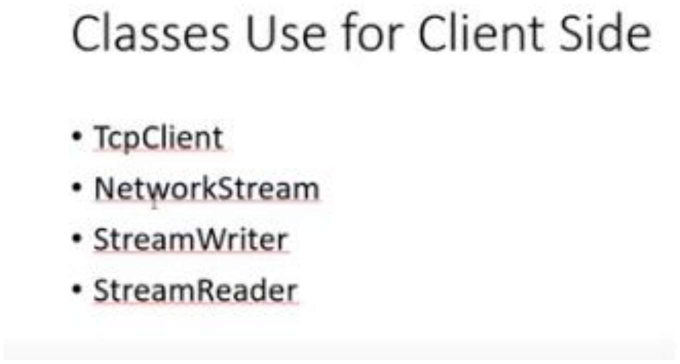
Architecture

- Server Side
- Client Side



Classes Use for Server Side

- TcpListener
- Socket
- NetworkStream
- StreamWriter
- StreamReader



Classes Use for Client Side

- TcpClient
- NetworkStream
- StreamWriter
- StreamReader

Managing Console I/O Operations

To allow console input/output operations, C# provides, C# provides a console class. The Console class provides basic input and output support for applications that read from and write characters to the console. The standard input, output, and error streams are represented by properties, and are automatically associated with the console when the application starts. Application can redirected these properties to other streams; for example, streams associated with fies instead of the console.

By default, the read methods in console class use the standard input stream (keyboard) and the write methods use the standard output (monitor) stream.

The write methods support writing data with or without automatically appending carriage return and linefeed characters. This enables the writing of string, formatted strings, arrays of characters, instances of primitive types, and arbitrary objects without first having to convert them to strings.

The following example demonstrates the use of basic Console input and output functions.

```
/* Console Input/Ouput */
using System;
class ConsoleTest
{
  public static void Main()
  {
    Console.Write("Hello");
    Console.WriteLine("World");
    Console.Write("What is Your Name");
    String name = Console.ReadLine();
    Console.Write("Hello, ");
    Console.Write(name);
    Console.WriteLine(" ! ");
  }
}
```

Console Class:

The methods for reading from and writing to the console are provided by the System. Console class. This class gives us access to the standard input, standard output.

Console Input:

The console input stream object supports two methods for obtaining input from the keyboard.

1. **Read():** It returns a single character as **int**.

Example:

- **int** p=Console.Read();
- Console.WriteLine((**char**)p);

2. **ReadLine():** It returns a string containing a line of text.

Example:

- **string** sk=Console.ReadLine();
- Console.WriteLine(sk);

Console Output:

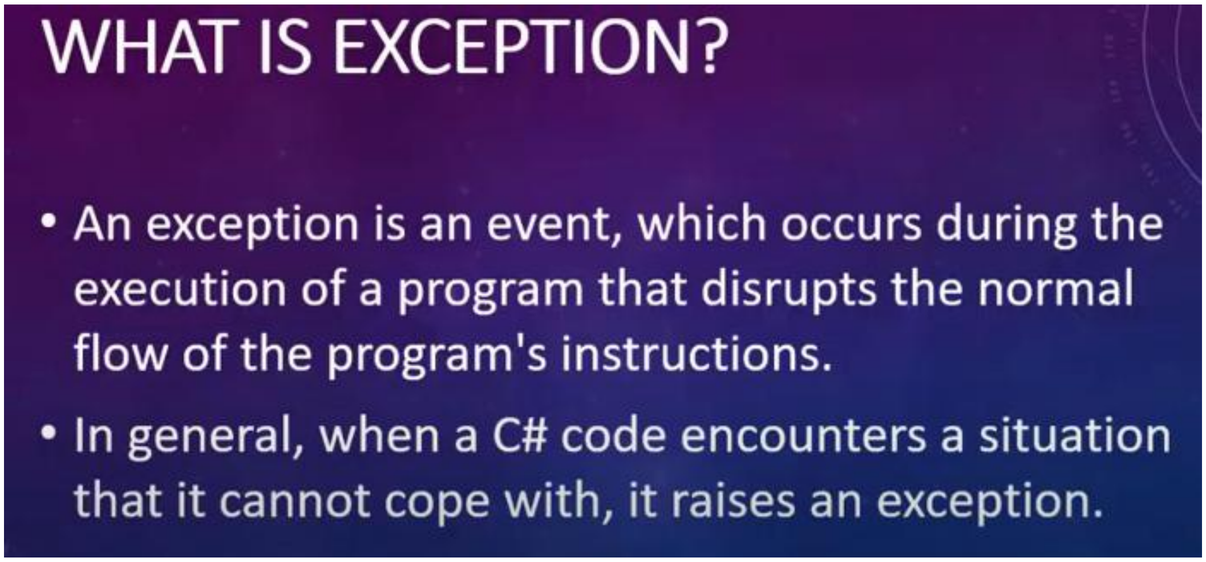
The console output stream supports two methods for writing to the console:

1. **Write():** Outputs one or more values to the screen without a newline character.
2. **WriteLine():** Outputs one or more values to the screen but adds a newline character at the end of the output.

C# - Error Handling

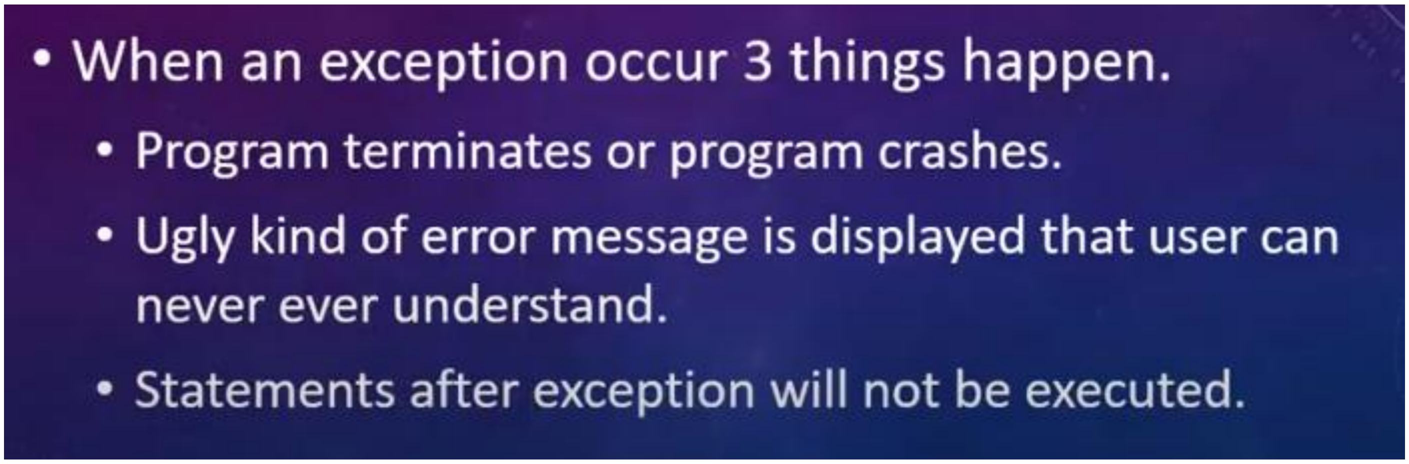
When executing C# code, different errors can occur: coding errors made by the programmer, errors due to wrong input, or other unforeseeable things.

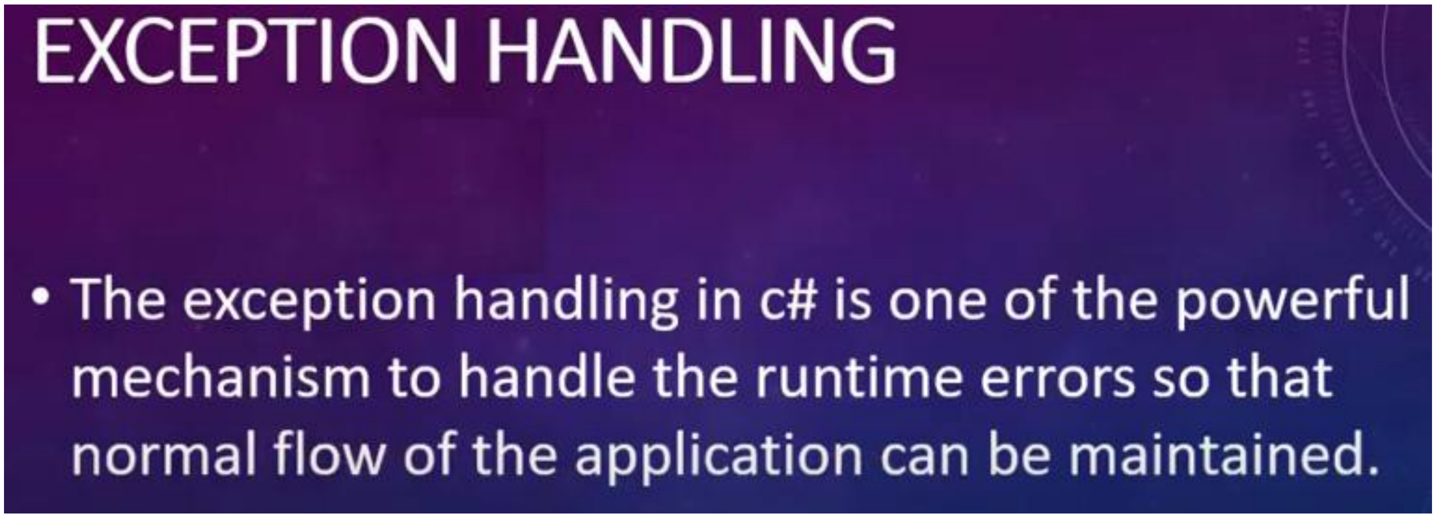
When an error occurs, C# will normally stop and generate an error message. The technical term for this is: C# will throw an **exception** (throw an error).



WHAT IS EXCEPTION?

- An exception is an event, which occurs during the execution of a program that disrupts the normal flow of the program's instructions.
- In general, when a C# code encounters a situation that it cannot cope with, it raises an exception.


- When an exception occur 3 things happen.
  - Program terminates or program crashes.
  - Ugly kind of error message is displayed that user can never ever understand.
  - Statements after exception will not be executed.

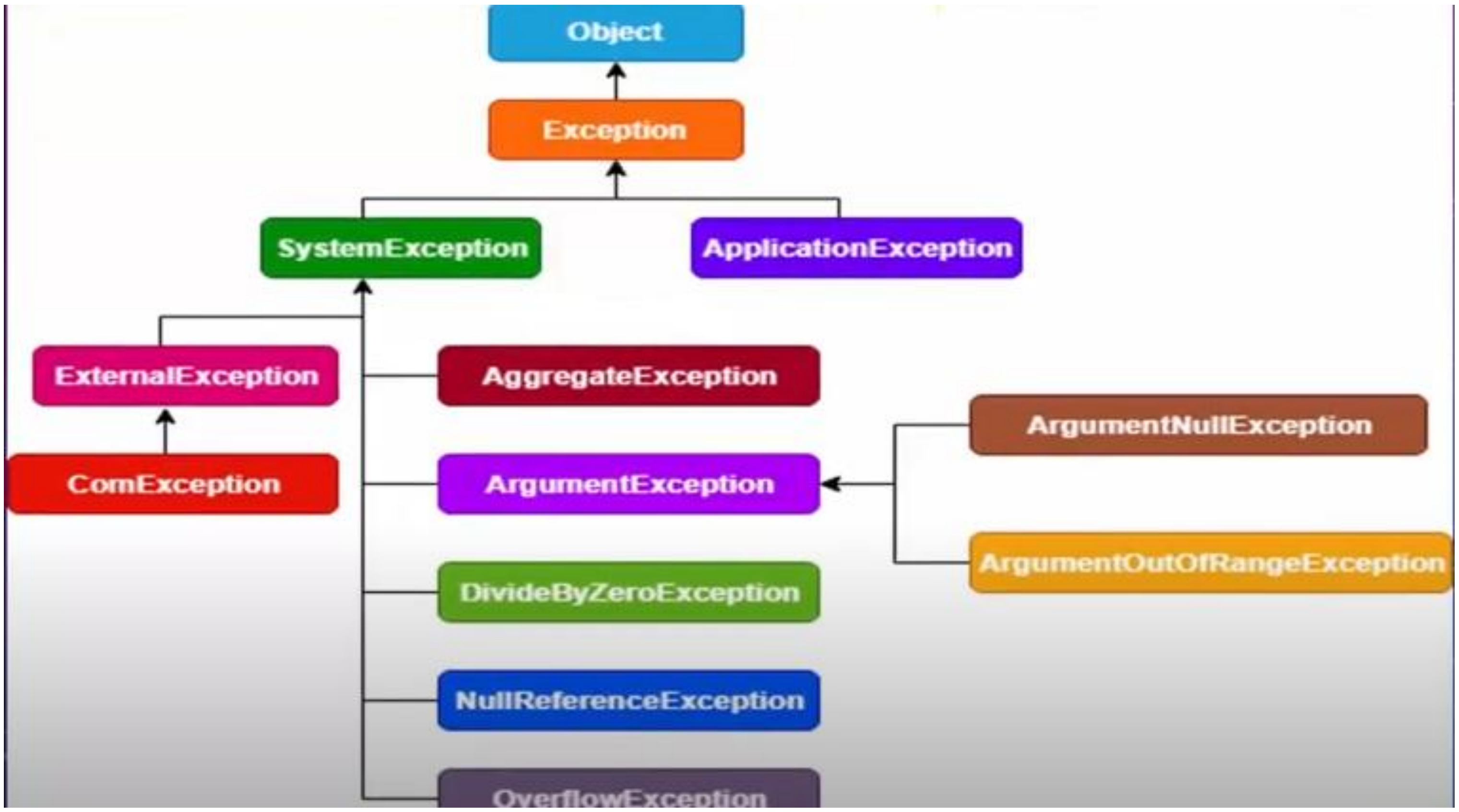

EXCEPTION HANDLING

- The exception handling in c# is one of the powerful mechanism to handle the runtime errors so that normal flow of the application can be maintained.

C# Exception Classes

All the exception classes in C# are derived from **System.Exception** class. Let's see the list of C# common exception classes.

Exception	Description
System.DivideByZeroException	handles the error generated by dividing a number with zero.
System.NullReferenceException	handles the error generated by referencing the null object.
System.InvalidCastException	handles the error generated by invalid typecasting.
System.IO.IOException	handles the Input Output errors.
System.FieldAccessException	handles the error generated by invalid private or protected field access.



Exceptions provide a way to transfer control from one part of a program to another. C# exception handling is built upon four keywords: **try**, **catch**, **finally**, and **throw**.

- **try** − A try block allows us to define a block of code to be tested for errors while it is being executed. It is followed by one or more catch blocks.
- **catch** − The catch keyword indicates the catching of an exception. When an exception occurs, the Catch block of code is executed. This is where we are able to handle the exception, log it, or ignore it.
- **finally** − The finally block is used to execute a given set of statements, whether an exception is thrown or not thrown. For example, if you open a file, it must be closed whether an exception is raised or not.
- **throw** − A programmer throws an exception when a problem shows up. This is done using a throw keyword.

C# try and catch

The try statement allows us to define a block of code to be tested for errors while it is being executed.

The catch statement allows us to define a block of code to be executed, if an error occurs in the try block.

The try and catch keywords come in pairs:

Syntax

```
try
{
  //  Block of code to try
}
catch (Exception e)
{
  //  Block of code to handle errors
}
```

Consider the following example, where we create an array of three integers:

This will generate an error, because **myNumbers[10]** does not exist.

```
int[] myNumbers = {1, 2, 3};
Console.WriteLine(myNumbers[10]); // error!
```

The error message will be something like this:

```
System.IndexOutOfRangeException: 'Index was outside the bounds of the array.'
```

If an error occurs, we can use try...catch to catch the error and execute some code to handle it.

In the following example, we use the variable inside the catch block (e) together with the built-in Message property, which outputs a message that describes the exception:

Example

```
using System;
namespace MyApplication
{
  class Program
  {
    static void Main(string[] args)
    {
      try
      {
        int[] myNumbers = {1, 2, 3};
        Console.WriteLine(myNumbers[10]);
      }
      catch (Exception e)
      {
        Console.WriteLine(e.Message);
      }
    }
  }
}
```

Output:

```
Index was outside the bounds of the array.
```

We can also output our own error message:

Example

```
try
{
  int[] myNumbers = {1, 2, 3};
  Console.WriteLine(myNumbers[10]);
}
catch (Exception e)
{
  Console.WriteLine("Something went wrong.");
}
```

Output:

```
Something went wrong.
```

Example 2

Without exception handling

```
using System;
namespace HelloWorld
{
  class Program
  {
    static void Main(string[] args)
    {
   Console.WriteLine("Enter First number");
   int num1 = Convert.ToInt32(Console.ReadLine());

Console.WriteLine("Enter Second number");
int num2 = Convert.ToInt32(Console.ReadLine());

     int result= num1/num2;
     Console.WriteLine(result);
       int sum = num1+num2;
       Console.WriteLine(sum);

    }
  }
}
```

Output:

```
Unhandled Exception:
System.DivideByZeroException: Attempted to divide by zero.
  at HelloWorld.Program.Main (System.String[] args) [0x0002a] in
<b179e51f5ea24fe5bec684f8b3e11004>:0
[ERROR] FATAL UNHANDLED EXCEPTION: System.DivideByZeroException: Attempted to divide
by zero.
  at HelloWorld.Program.Main (System.String[] args) [0x0002a] in
<b179e51f5ea24fe5bec684f8b3e11004>:0
```

With exception handling

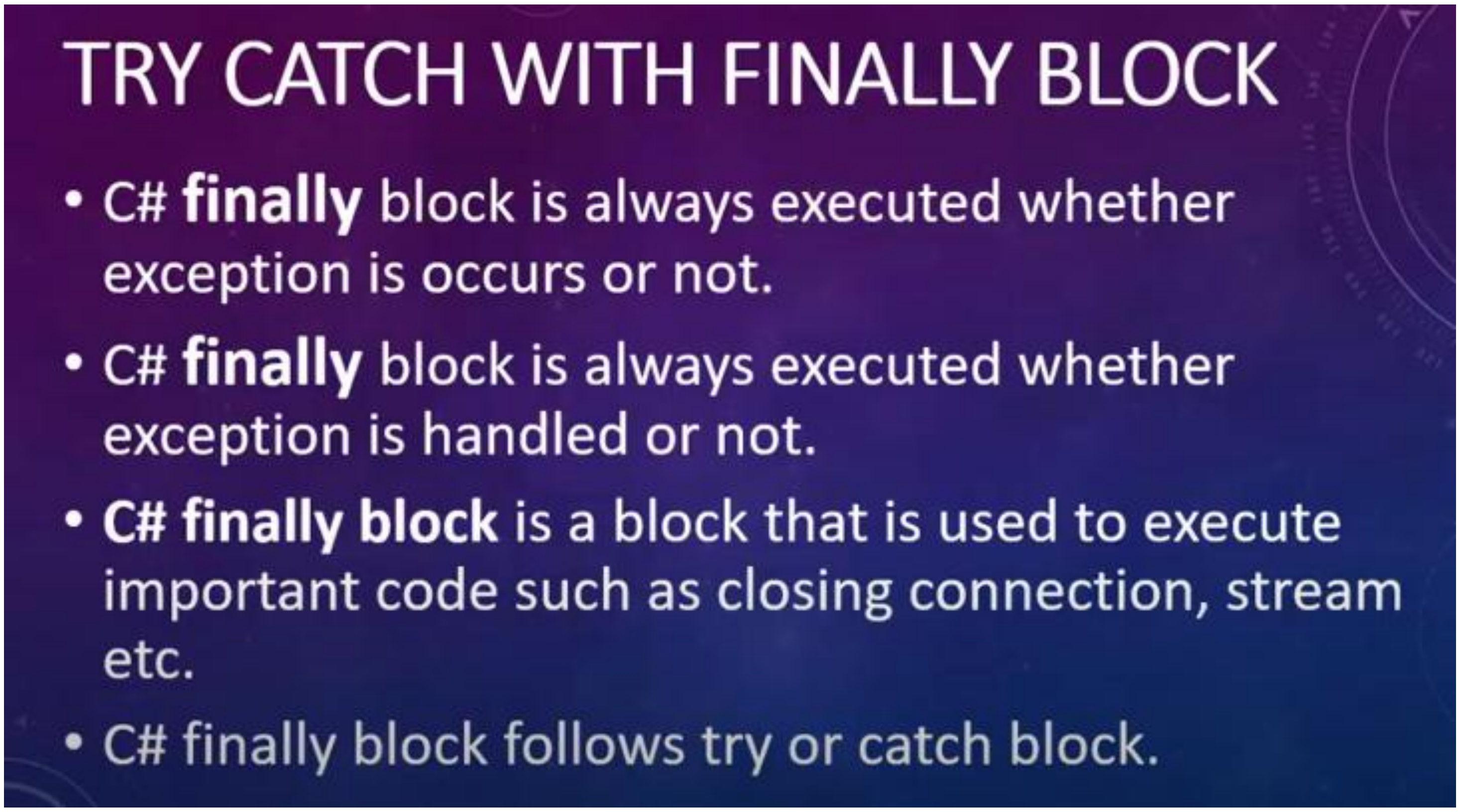
```
using System;
namespace HelloWorld
{
  class Program
  {
    static void Main(string[] args)
    {
   Console.WriteLine("Enter First number ");
   int num1 = Convert.ToInt32(Console.ReadLine());

  Console.WriteLine("Enter Second number ");
   int num2 = Convert.ToInt32(Console.ReadLine());
    try
    {
     int result= num1/num2;
     Console.WriteLine(result);
    }
   catch (Exception e)
       {
        Console.WriteLine("You can not divide a number by zero");
       Console.WriteLine(e.Message);
       }
       int sum = num1+num2;
       Console.WriteLine(sum);

    }
  }
}
```

Output:

```
Enter First number 10
Enter Second number 0
You can not divide a number by zero
Attempted to divide by zero.
10
```

Finally

The finally statement lets you execute code, after try...catch, regardless of the result:



TRY CATCH WITH FINALLY BLOCK

- C# **finally** block is always executed whether exception is occurs or not.
- C# **finally** block is always executed whether exception is handled or not.
- **C# finally block** is a block that is used to execute important code such as closing connection, stream etc.
- C# finally block follows try or catch block.

Example

```
using System;
namespace MyApplication
{
  class Program
  {
    static void Main(string[] args)
    {
      try
      {
        int[] myNumbers = {1, 2, 3};
        Console.WriteLine(myNumbers[10]);
      }
      catch (Exception e)
      {
        Console.WriteLine("Something went wrong.");
      }
      finally
      {
        Console.WriteLine("The 'try catch' is finished.");
      }
    }
  }
}
```

Output:

```
Something went wrong.
The 'try catch' is finished.
```

The throw keyword

C# allows us to create user-defined or custom exception. It is used to make the meaningful exception. To do this, we need to inherit Exception class. The **throw** statement allows us to create a custom error.

The throw statement is used together with an **exception class**.

Example

```
using System;

class program

{

public static void Main()

{
```

```
Console.WriteLine(Enter your Age :);

int age=Convert.ToInt32(Console.ReadLine());

 try

 {

             if(age>=18)

             {

             Console.WriteLine("You can vote");

             }



            else

             {

            throw new Exception("You can not vote.");

            }

 }

            catch (Exception e)

            {

            Console.WriteLine(e.Message);

            }

}

}
```

The error message displayed in the program will be:

Output:

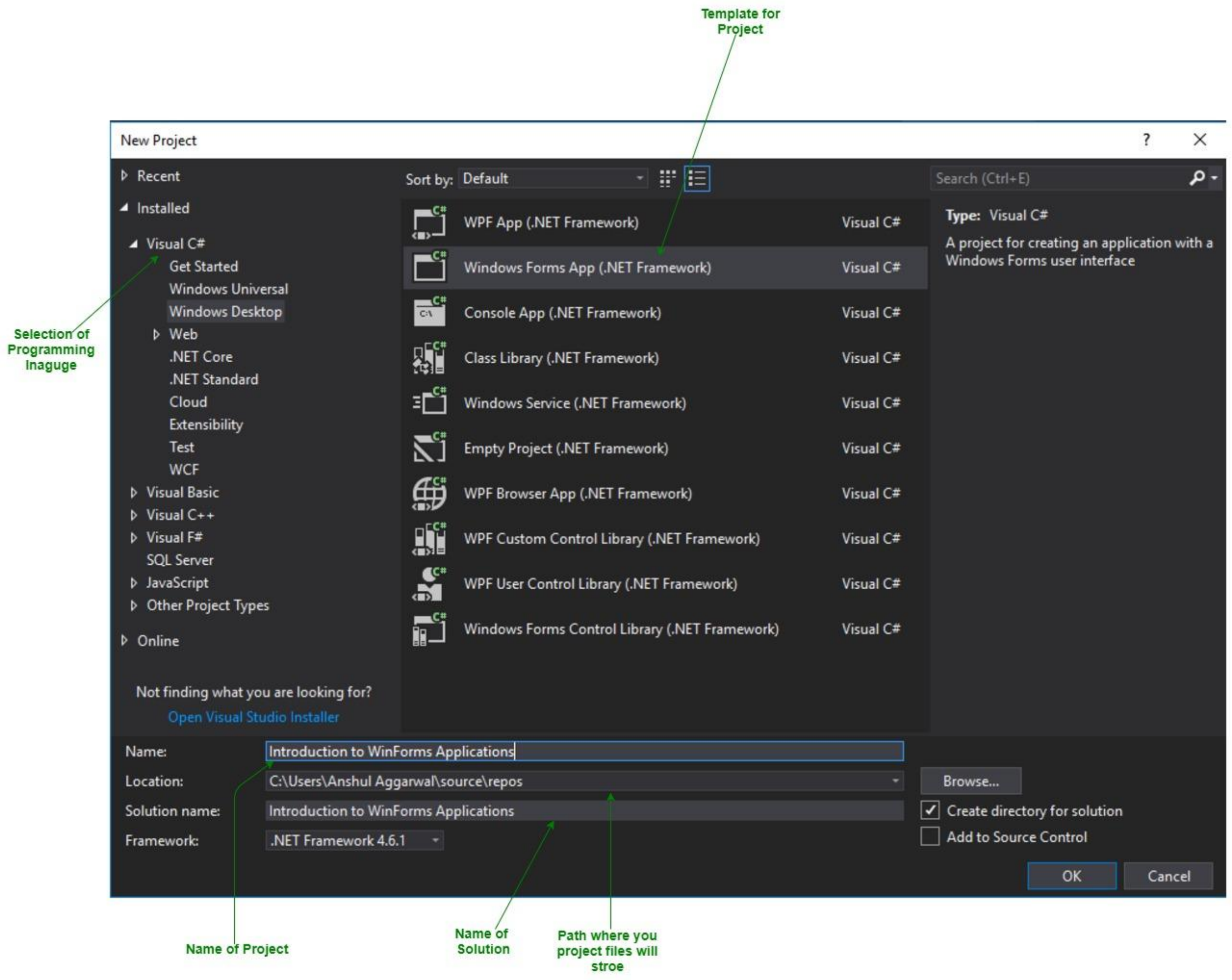
```
Enter your Age :

15

You can not vote.
```

Windows Forms

Windows Forms is a Graphical User Interface (GUI) class library which is bundled in *.Net Framework*. Its main purpose is to provide an easier interface to develop the applications for desktop, tablet, PCs. It is also termed as the **WinForms**. The applications which are developed by using **Windows Forms** or **WinForms** are known as the **Windows Forms Applications** that runs on the desktop computer. WinForms can be used only to develop the Windows Forms Applications not web applications. WinForms applications can contain the different type of controls like labels, list boxes, tooltip etc.

Creating a Windows Forms Application Using Visual Studio 2017

- First, open the Visual Studio then Go to **File -> New -> Project** to create a new project and then select the language as *Visual C#* from the left menu. Click on *Windows Forms App(.NET Framework)* in the middle of current window. After that give the project name and Click **OK**.


Template for Project
New Project
Recent
Installed
Visual C#
Get Started
Windows Universal
Windows Desktop
Web
.NET Core
.NET Standard
Cloud
Extensibility
Test
WCF
Visual Basic
Visual C++
Visual F#
SQL Server
JavaScript
Other Project Types
Online
Not finding what you are looking for?
Open Visual Studio Installer
Sort by: Default
Search (Ctrl+E)
WPF App (.NET Framework) Visual C#
Windows Forms App (.NET Framework) Visual C#
Console App (.NET Framework) Visual C#
Class Library (.NET Framework) Visual C#
Windows Service (.NET Framework) Visual C#
Empty Project (.NET Framework) Visual C#
WPF Browser App (.NET Framework) Visual C#
WPF Custom Control Library (.NET Framework) Visual C#
WPF User Control Library (.NET Framework) Visual C#
Windows Forms Control Library (.NET Framework) Visual C#
Browse...
OK
Cancel
Selection of Programming lnagude
Name of Project
Name of Solution
Path where you project files will stroe

Type: Visual C#

A project for creating an application with a Windows Forms user interface

Name: Introduction to WinForms Applications

Location: C:\Users\Anshul Aggarwal\source\repos

Solution name: Introduction to WinForms Applications

Framework: .NET Framework 4.6.1

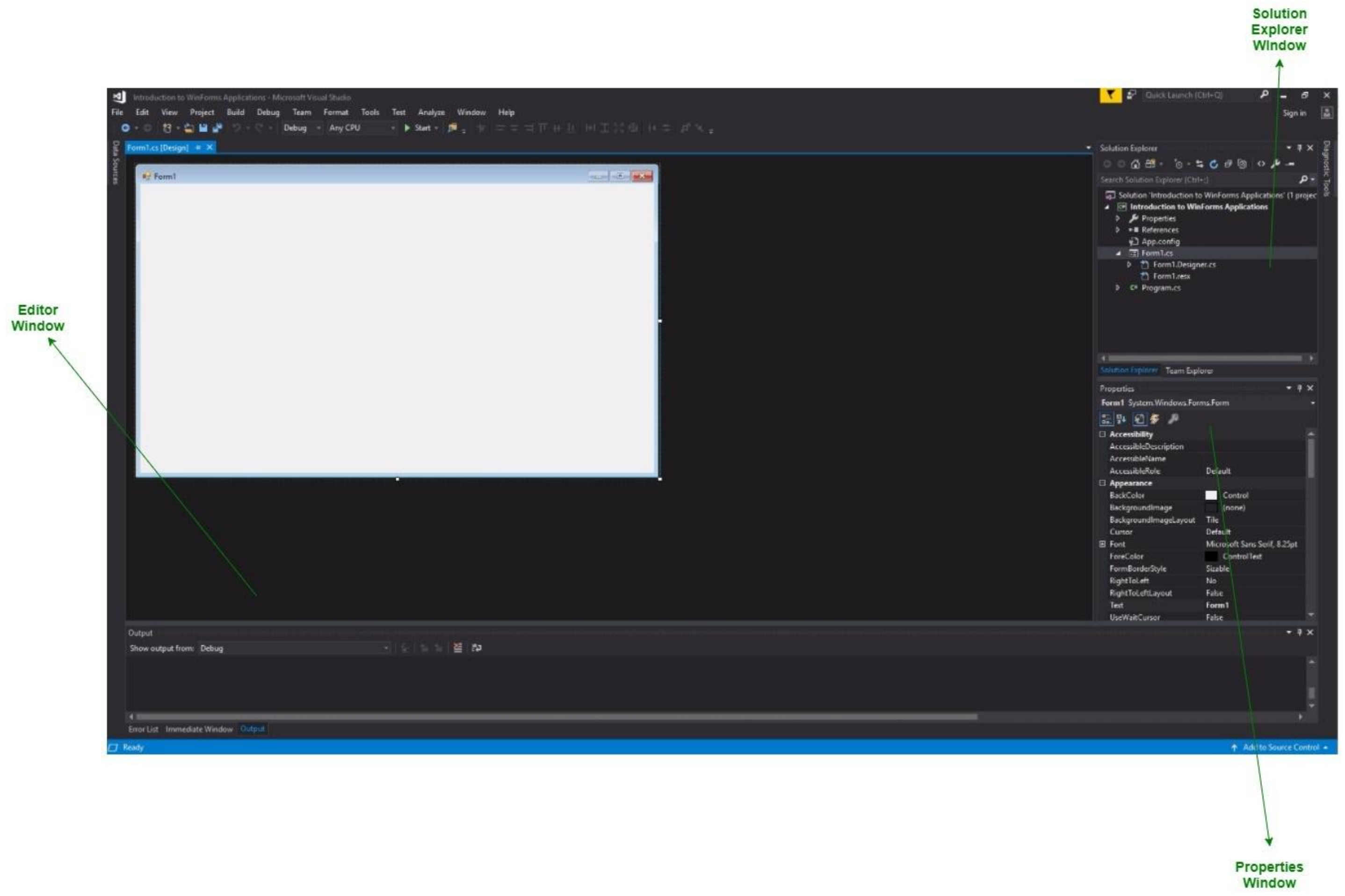
[✓] Create directory for solution

[ ] Add to Source Control

Here the solution is like a container which contains the projects and files that may be required by the program.

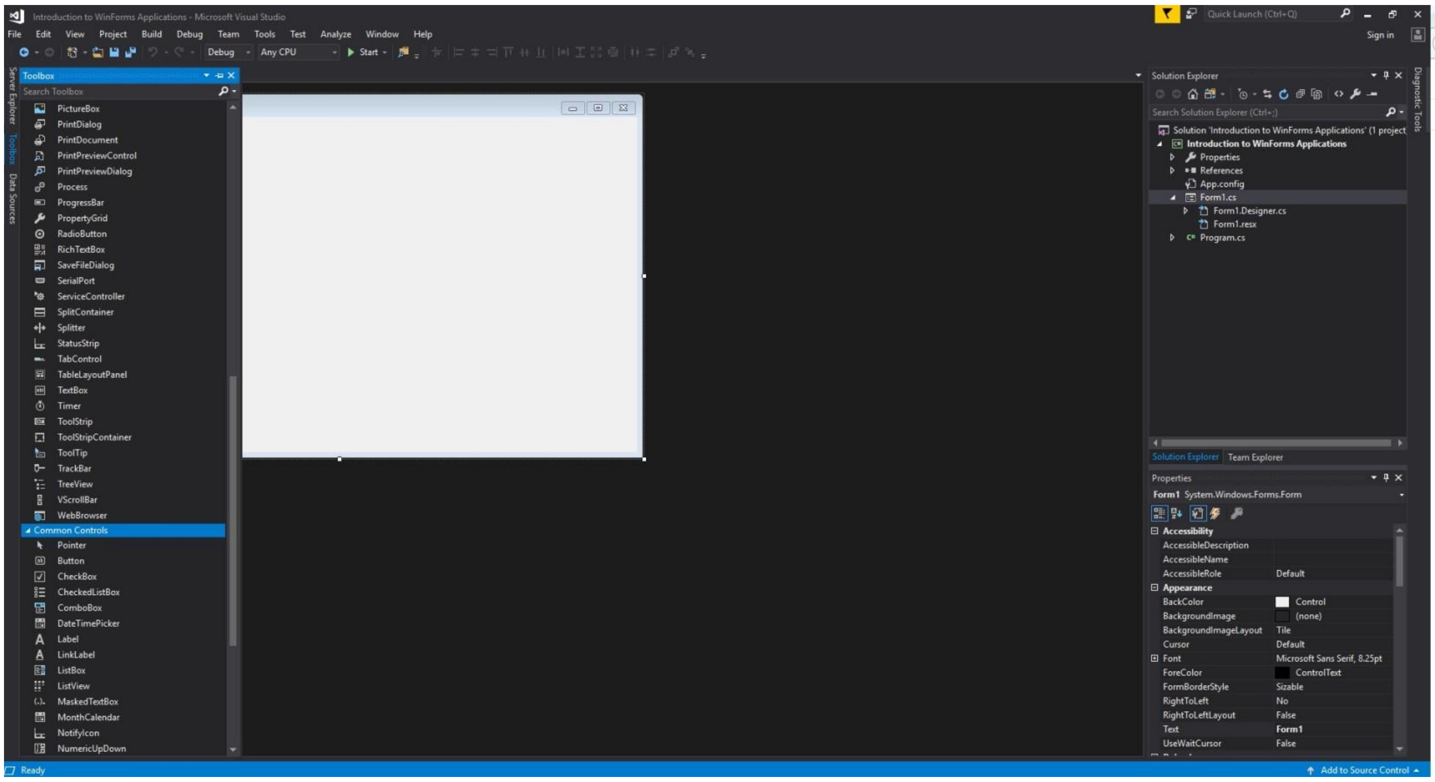
- After that following window will display which will be divided into three parts as follows:

1. **Editor Window or Main Window:** Here, you will work with forms and code editing. You can notice the layout of form which is now blank. You will double click the form then it will open the code for that.
2. **Solution Explorer Window:** It is used to navigate between all items in solution. For example, if you will select a file form this window then particular information will be display in the property window.
3. **Properties Window:** This window is used to change the different properties of the selected item in the Solution Explorer. Also, you can change the properties of components or controls that you will add to the forms.



You can also reset the window layout by setting it to default. To set the default layout, go to **Window -> Reset Window Layout** in Visual Studio Menu.

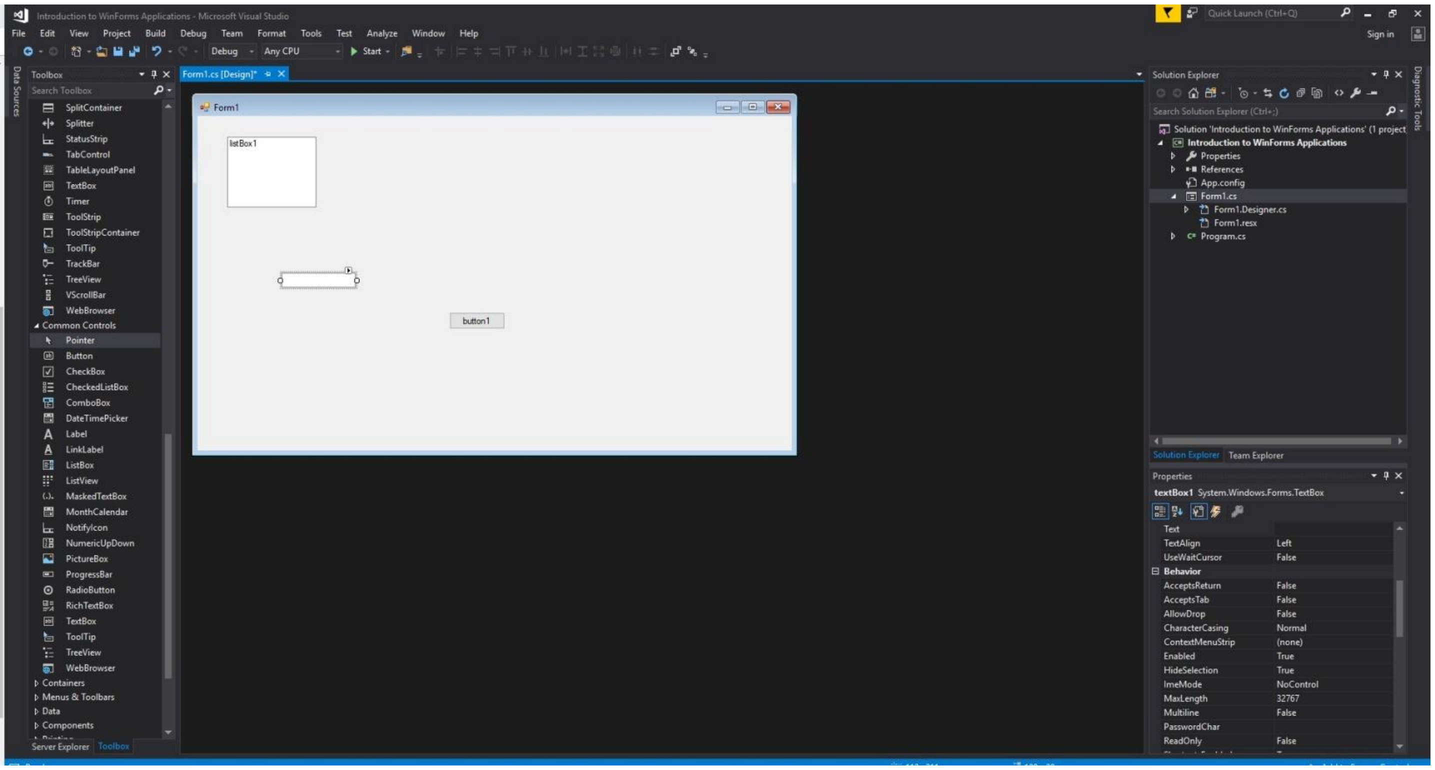
- Now **to add the controls to your WinForms application** go to **Toolbox** tab present in the extreme left side of Visual Studio. Here, you can see a list of controls. To access the most commonly used controls go to **Common Controls** present in Toolbox tab.


Introduction to WinForms Applications - Microsoft Visual Studio
File Edit View Project Build Debug Team Tools Test Analyze Window Help
Toolbox
Common Controls
Solution Explorer
Solution 'Introduction to WinForms Applications' (1 project)
Introduction to WinForms Applications
Properties
References
App.config
Form1.cs
Form1.Designer.cs
Form1.resx
Program.cs
Ready
Add to Source Control

Properties
Form1 System.Windows.Forms.Form

Accessibility	
AccessibleDescription	
AccessibleName	
AccessibleRole	Default
Appearance	
BackColor	Control
BackgroundImage	(none)
BackgroundImageLayout	Tile
Cursor	Default
Font	Microsoft Sans Serif, 8.25pt
ForeColor	ControlText
FormBorderStyle	Sizable
RightToLeft	No
RightToLeftLayout	False
Text	**Form1**
UseWaitCursor	False

- Now drag and drop the controls that you needed on created Form. For example, if you can add TextBox, ListBox, Button etc. as shown below. By clicking on the particular dropped control you can see and change its properties present in the right most corner of Visual Studio.


Introduction to WinForms Applications - Microsoft Visual Studio
File Edit View Project Build Debug Team Format Tools Test Analyze Window Help
Toolbox
Form1.cs [Design]*
Form1
listBox1
button1
Common Controls
Containers
Menus & Toolbars
Data
Components
Solution Explorer
Team Explorer

Properties
textBox1 System.Windows.Forms.TextBox

Text	
TextAlign	Left
UseWaitCursor	False
Behavior	
AcceptsReturn	False
AcceptsTab	False
AllowDrop	False
CharacterCasing	Normal
ContextMenuStrip	(none)
Enabled	True
HideSelection	True
ImeMode	NoControl
MaxLength	32767
Multiline	False
PasswordChar	
ReadOnly	False

In the above image, you can see the TextBox is selected and its properties like TextAlign, MaxLength etc. are opened in right most corner. You can change its properties' values as per the application need. The code of controls will be automatically added in the background. You can check the *Form1.Designer.cs* file present in the Solution Explorer Window.

- To run the program you can use an **F5 key** or **Play button** present in the toolbar of Visual Studio. To stop the program you can use pause button present in the ToolBar. You can also run the program by going to **Debug->Start Debugging** menu in the menubar.

Unit-4

.Net Framework using C#

Web Services in C#

Introduction

Web Service is known as the software program. These services use the XML to exchange the information with the other software with the help of the common internet Protocols. In the simple term, we use the Web Service to interact with the objects over the internet. Web services are web application components. Web services can be published, found, and used on the Web.

Here are some points about the Web Service.

1. Web Service is not dependent on any particular language.
2. Web Service is a protocol Independent.
3. Web Service is platform-independent.
4. Web Service is known as the Stateless Architecture. These services are dependent only on the given input.
5. Web Service is also Scalable.
6. Web Service is based on the XML (Open, Text-Based Standard).

Technology Used in Web Service

XML (Extensible Markup Language): Web Service specifies only the data. So, the application which understands the XML regarding the programming language or the platform can format the XML in different ways.

SOAP (Simple Object Access Protocol): The communication between the Services and the application is set up by the SOAP. It is a **protocol** which is **used to** interchange data between applications which are built on different programming languages. **SOAP** is built upon the **XML specification** and works with the **HTTP protocol**. This makes it a perfect for usage within web applications.

WSDL (Web Services Description Language.): The Web Services Description Language is an XML-based interface description language that is used for describing the functionality offered by a web service. WSDL gives us a uniform method that is helpful to specify the Web Services to the other programs. The WSDL document formally defines a web service. It contains

1. All the methods that are exposed by the web service
2. The parameters and their types
3. The return types of the methods

UDDI (Universal Description, Discovery, and Integration): Universal Description, Discovery and Integration (**UDDI**) is a platform-independent, Extensible Markup Language protocol that includes a (XML-based) registry by

which businesses worldwide can list themselves on the Internet, and a mechanism to register and locate web service applications. With the help of UDDI, we can search the Web Service registries.
At the time of the deployment of these technologies, this allows the developers to do the packaging of the applications in the form of the Service and publishing of the Service on the network.

Advantages of Web Services

1. Web Services always use the open, text-based standard. Web service uses all those components even they are written in various languages for different platforms.
2. Web Service promotes the modular approach of the programming so that the multiple organizations can communicate with the same web service.
3. Web Services are easy to implement but expensive because they use the existing infrastructure, and we can repackage most of the applications as the Web Service.
4. Web Service reduces the cost of enterprise application integration and B2B communications.
5. Web Services are the interoperable Organization that contains the 100 vendors and promote them interoperability.

Limitations of Web Services

Limitations of Web Services are:

1. One of the limitations of the Web Services is that the SOAP, WSDL, UDDI requires the development.
2. Supports to the interoperability are also the limitation of the Web Service.
3. The limitation of web service is also royalty fees.
4. If we want to use web services for the high-performance situation, in that case, web service will be slow.
5. The use of web service increases the traffic on the network.
6. The security level for Web Services is low.
7. We use the standard procedure to describe the quality of particular web services.
8. The standard for the Web Service is in the draft form.
9. To retain the intellectual property of the specific standard by the vendor is also the limitation of the web service.

Here we have some points about the Web Services.
Note1: Web Services are not limited only to the .Net Framework. The standards of Web Services were already defined before the release of the .NET. Web Services and supported by the vendors other than Microsoft.
Note2: Web Services can also work on the cross-platform. If the services were written in one language, these could be used by the other application despite, and the application used the other language. If we want to use the web services, the only way for that is we only need the internet to connect where we will make the HTTP request.

As we know that the Web Service is cross-platform, but despite this, there should be an understandable language so that we can make a request for the services and can get the Service in their response. Web Services use the XML, which can be understood easily.

This is the only reason why the web services were built with the XML based standards of exchanging the data.

Web Services uses the Set of Data type. The XML Schema easily recognizes these data types.

Web Services uses a simple data type like strings and numbers. These data types are helpful for communication with Web Services. And we cannot send the proprietary .NET objects like image, FileStream, or the EventLogs. The other programming language does not have any way to contact these .NET objects. If we use some devices to send them to the client, still the different programming languages will not be able to interpret.

Note:3 If we want to work with the .NET objects, we can use the .NET remoting. .NET remoting is known as distributed technology through which we can use the .NET objects. But the non-.NET client can't use it.

Create Web Service

Create a web service

A web service is a simple **asmx** page.

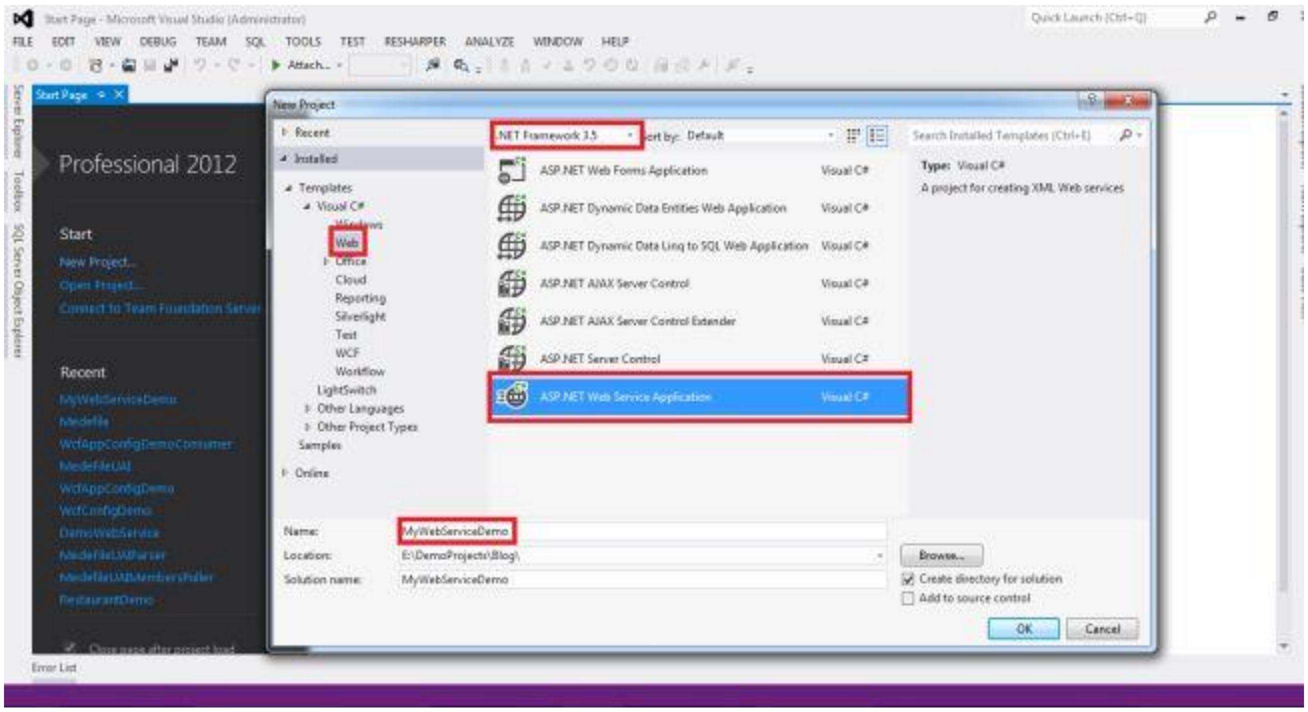
Here I will use Visual Studio 2012 (though you can use any editor), with the .Net Framework 3.5 to create a web service.

Up to framework 3.5, Visual Studio provides a direct template for Web Services. Starting from 4, it doesn't provide any direct template, so you need to create a web application and add a web service to your application (right-click on your web application → Add → New Item → WebService).

Let's create a simple service that will return a sum of 2 integers.

Open Visual Studio in Administrator mode. File → New → Project then select .Net Framework 3.5 (on the top) then select ASP.NET web service application then name your project (I named it MyWebServiceDemo) then click OK.

Here ASP stands for **Active Server Page.**



.NET Framework 3.5 Sort by: Default

ASP.NET Web Forms Application Visual C#
ASP.NET Dynamic Data Entities Web Application Visual C#
ASP.NET Dynamic Data Linq to SQL Web Application Visual C#
ASP.NET AJAX Server Control Visual C#
ASP.NET AJAX Server Control Extender Visual C#
ASP.NET Server Control Visual C#
ASP.NET Web Service Application Visual C#

Type: Visual C#
A project for creating XML Web services

Name: MyWebServiceDemo
Location: E:\DemoProjects\Blog\
Solution name: MyWebServiceDemo
Create directory for solution [✓]
Add to source control [ ]

Visual Studio will create a web service boilerplate (**Service1.asmx** and **Service1.asmx.cs**) for you. Now, let's analyze this template created by Visual Studio.

Note the following In Service1.asmx.cs:

1. An additional namespace "**System.Web.Services**" is included along with the 4 default namespaces that Visual Studio includes for web application.

2. The "Service1" class is inherited from "**System.Web.Services.WebService**". By inheriting a class from "**System.Web.Services.WebService**", you can access built-in ASP.NET objects such as (Application, Session, User, Context, server). If you don't need built-in objects of .Net, you don't need to inherit your service class from "**WebService**".

3. "Service1" is decorated with a "**WebService**(Namespace = "http://tempuri.org/")" attribute. If you want to expose a class as a service, you need to decorate it with the "WebService" attribute.
   This **WebService** attribute has several properties like:
   - **Namespace:** This property makes a service uniquely identifiable. This is an XML property. A client application may be consuming several services, so there is the probability of a naming collision. To avoid this, it's service providers responsibility to use a unique namespace.
   - **Name**: Use this property to provide a descriptive name to your service.
   - **Description:** To provide a brief description on the service.

4. “Service1” has another attribute “**WebServiceBinding**(ConformsTo = **WsiProfiles**.BasicProfile1_1)”. This is to indicate the standard which service is following. If the service does not confirm to this standard, you will get an exception.
5. One more attribute service is decorated with is “[System.Web.Script.Services.**ScriptService**]”, to make a service accessible from the client script, it should be decorated with this attribute.
6. The Service1 class has a method HelloWorld that is decorated with a “[WebMethod]” attribute. The methods of the service that are to be accessed by the client application should be decorated with this attribute. There may be some method that the service is using for some internal functionality, client applications don't need to access them. Don't decorate such methods with a WebMethod attribute. The WebMethod attribute also has Name and Description properties that you can use to provide a self describing name or description respectively.

Now let's see the mark up. Right-click on Service1.asmx in Solution Explorer then select view mark up. In Service1.asmx, you will see that there is only a WebService directive with some attributes, since a service will be invoked by some application not by any end user. So the asmx page has no mark up.

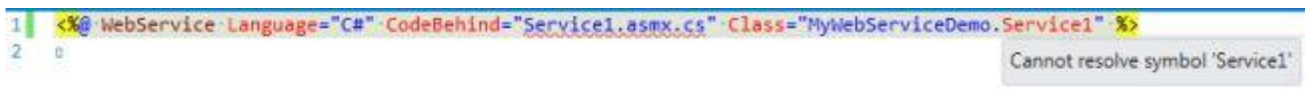
```
<%@ WebService Language="C#" CodeBehind="Service1.asmx.cs"
Class="MyWebServiceDemo.Service1"%>
```

- The “**WebService**” directive, indicates this asmx page is a web service.
- The “**Language="C#"**”, is to indicate language used for this service.
- The “**CodeBehind**” property is nothing to do with ASP.NET or web service, this is completely a Visual Studio property, that is used to map the asmx page with it's code behind page.
- The “**Class**” property holds fully qualified name of service class. This marks the entry point of service like main() in C programming language.

Now, run your application by hitting F5, http://localhost:56655/Service1.asmx will open in your browser (the port number may vary). You will find a link for Service Description, that will redirect to the WSDL document of the service, another link for HelloWorld (list for methods exposed by service) that will redirect to a page for testing this method.

Implementing a web service

Now let's implement the service. Rename the “Service1” file in Solution Explorer to something convenient like “MyService”. Change the class name from Service1 to MyService. Open the mark up (asmx) page.



```
<%@ WebService Language="C#" CodeBehind="Service1.asmx.cs" Class="MyWebServiceDemo.Service1" %>
```

Cannot resolve symbol 'Service1'

As you can see here Visual Studio is unable to resolve “Service1” in the class property, since the class indicates a fully qualified name of the service and we renamed our Service1 class to MyService. So Visual Studio is unable to resolve it. So, now change the class property to “**MyWebServiceDemo.MyService**”. Change the “CodeBehind” property from “Service1.asmx.cs” to “MyService.asmx.cs” as we renamed the file also.

MyService.asmx

```
<%@ WebService Language="C#" CodeBehind="MyService.asmx.cs"
Class="MyWebServiceDemo.MyService"%>
```

MyService.asmx.cs

```
using System.Web.Script.Serialization;
using System.Web.Services;

namespace MyWebServiceDemo
{
    // Use "Namespace" attribute with an unique name,to make service uniquely        discoverable
    [WebService(Namespace = "http://tempuri.org/")]
    // To indicate service confirms to "WsiProfiles.BasicProfile1_1" standard,
    // if not, it will throw compile time error.
    [WebServiceBinding(ConformsTo = WsiProfiles.BasicProfile1_1)]
    // To restrict this service from getting added as a custom tool to toolbox
    [System.ComponentModel.ToolboxItem(false)]
    // To allow this Web Service to be called from script, using ASP.NET AJAX
    [System.Web.Script.Services.ScriptService]
    public class MyService : WebService
    {
        [WebMethod]
        public int SumOfNums(int First, int Second)
        {
            return First + Second;
        }
    }
}
```

Now, the service is ready to be used, let's compile and test it.

Test a web service

Let's run the project by hitting F5. The "**http://localhost:56655/MyService.asmx**" page will open that has a link for the Service description (the WSDL document, documentation for web service) another link for SumOfNums, which is for test page of SumOfNums method.

Let's use method overloading of the OOP concept. Add the following WebMethod in MyService class.

```
[WebMethod]
public float SumOfNums(float First, float Second)
{
    return First + Second;
}
```

Hit F5 to run the application, you will get "Both Single SumOfNums(Single, Single) and Int32 SumOfNums(Int32, Int32) use the message name 'SumOfNums'. Use the MessageName property of the WebMethod custom attribute to specify unique message names for the methods." error message. We just used method overloading concept, so why this error message? This is because these methods are not unique for a client application. As the error message suggests let's use the MessageName property of the WebMethod attribute as shown below:

```
[WebMethod (MessageName = "SumOfFloats")]
public float SumOfNums(float First, float Second)
{
    return First + Second;
}
```

Now, compile and run the application. Again it's showing some different error message "Service 'MyWebServiceDemo.MyService' does not conform to WS-I Basic Profile v1.1". As, **WsiProfiles**.BasicProfile1_1 doesn't support method overloading we are getting this exception. Now, either remove this "[**WebServiceBinding**(ConformsTo = **WsiProfiles**.BasicProfile1_1)]" attribute or make it "[WebServiceBinding(ConformsTo = WsiProfiles.None)]".

```
using System.Web.Script.Serialization;
using System.Web.Services;

namespace MyWebServiceDemo
{
   [WebService(Namespace = "http://tempuri.org/")]
   [WebServiceBinding(ConformsTo = WsiProfiles.None)]
   [System.ComponentModel.ToolboxItem(false)]
```

```
    [System.Web.Script.Services.ScriptService]
    public class MyService : WebService
    {
        [WebMethod]
        public int SumOfNums(int First, int Second)
        {
            return First + Second;
        }
        [WebMethod(MessageName = "SumOfFloats")]
        public float SumOfNums(float First, float Second)
        {
            return First + Second;
        }
    }
}
```
Now you can use method overloading in the service.

The Test page

Click on SumOfNums, you will be redirected to "**http://localhost:56655/MyService.asmx?op=SumOfNums**". You will see here that just "?op=SumOfNums" is appended to the service URL. This page has 2 text boxes for 2 input values (First, Second) that the SumOfNums method takes as an input parameter and a button "Invoke", clicking on which you'll be redirected to: "**http://localhost/WebServiceForBlog/MyService.asmx/SumOfNums**" that is has the value that the SumOfNums method returned in XML format. Similarly, by clicking on "SumOfNums MessageName="SumOfFloats"", you will be redirected to "http://localhost:56655/MyService.asmx?op=**SumOfFloats**". So the "SumOfNums MessageName="SumOfFloats"" method will be known as "SumOfFloats" for client applications.

Now, the question is from where does this test page come? We never added any mark up but still a page was rendered!

The test pages aren't part of the Web Services; they're just a frill provided by ASP.NET. The test page is rendered by ASP.NET using the web page c:\[WinDir]\Microsoft. NET\Framework\[Version] \Config\DefaultWsdlHelpGenerator.aspx. "Reflection" concept to render the test page.

You can also modify this test page for which you simply need to copy the DefaultWsdlHelpGenerator.aspx file to your web application directory, modify it and give it a name. I named it "**MyWsdlHelpGenerator.aspx**" and then changed the web.config file for the application by adding the <wsdlHelpGenerator> element, as shown here:

```
<configuration>
  <system.web>
    <webServices>
      <wsdlHelpGenerator href="MyWsdlHelpGenerator.aspx"/>
    </webServices>
  </system.web>
</configuration>
```

The WSDL document

Web Services are self-describing, that means ASP.NET automatically provides all the information the client needs to consume a service as a WSDL document. The WSDL document tells a client what methods are present in a web service, what parameters and return values each method uses and how to communicate with them. WSDL is a XML standard.

Host web service

Since we will add a reference to this service and consume it from various applications and the port number is supposed to change, let's host this service on IIS to have a specific address of a service. Open IIS (To **open IIS Manager** from a **command** window , type **start** inetmgr and press ENTER.), go to the default web sites node then select Add Application then provide an alias name (I gave it WebServiceForBlog) then browse to the physical location of your service for the physical path field then click “OK”. You can now browse with an alias name (like **http://localhost/WebServiceForBlog/**) to test if the application was hosted properly. You'll get a “**HTTP Error 403.14 – Forbidden**” error since there is no default document set for this application. Add a “MyService.asmx” page as a default document. Now you can browse your service.

How to access a web service from a client application

Here our primary focus will be on consuming a web service. Let's quickly create a web service (or modify, if you have already created one).

MyService.asmx

```
<%@ WebService Language="C#" CodeBehind="MyService.asmx.cs"
Class="MyWebServiceDemo.MyService" %>
```

MyService.asmx.cs

```
using System.Web.Script.Serialization;
using System.Web.Services;

```

```
namespace MyWebServiceDemo
{
    [WebService(Namespace = "http://tempuri.org/")]
    [WebServiceBinding(ConformsTo = WsiProfiles.None)]
    [System.ComponentModel.ToolboxItem(false)]
    [System.Web.Script.Services.ScriptService]
    public class MyService
    {
        // Takes 2 int values & returns their summation
        [WebMethod]
        public int SumOfNums(int First, int Second)
        {
            return First + Second;
        }

        // Takes a stringified JSON object & returns an object of SumClass
        [WebMethod(MessageName = "GetSumThroughObject")]
        public SumClass SumOfNums(string JsonStr)
        {
            var ObjSerializer = new JavaScriptSerializer();
            var ObjSumClass = ObjSerializer.Deserialize<SumClass>(JsonStr);
            return new SumClass().GetSumClass(ObjSumClass.First, ObjSumClass.Second);
        }
    }

     // Normal class, an instance of which will be returned by service
    public class SumClass
    {
        public int First, Second, Sum;

        public SumClass GetSumClass(int Num1, int Num2)
        {
            var ObjSum = new SumClass
            {
                Sum = Num1 + Num2,
            };
            return ObjSum;
        }
    }
}
```

Compile this application. Since we will consume this service from other applications we need a fixed address for this service, the port numbers are supposed to vary. So, host the web service on IIS (refer to the previous article for hosting the web service). I hosted it with the virtual directory “**WebServiceForBlog**”.

How to consume a web service

Let's add another project to our solution. You can also create a new project. I am adding to the same solution, so that it will not need me to open several instances of Visual Studio. When developing and testing a web service in Visual Studio, it's often preferred to add both the web service and the client application to the same solution. This allows you to test and change both pieces at the same time. You can even use the integrated debugger to set breakpoints and step through the code in both the client and the server.

Right-click on solution, select Add → New project then seelct ASP.NET Empty Application then name your application, I am naming it “ServiceConsumer”.
Add a Web Form, I name it “Home.aspx”.
On the click of a button in this page, we will call our service.
For calling a Web Service you need a proxy object that will handle the complexities of sending a SOAP request and response messages.
To create this proxy class, you need a reference to the service class. Right-click on this project then select Add service reference, a window will open, type the URL of your service, click on discover, you will see all the webmethods exposed by your service listed. At the bottom of the window, there'll be a field for Namespace. Provide a name for the namespace in which the proxy class of the referenced service will be generated, I am giving it “MyServiceReference”. Now, click on “Go”.
By adding a service reference, we created a proxy class of the referenced service to the current project (client app). The proxy class wraps the calls to the web service's methods. It takes care of generating the correct SOAP message format and managing the transmission of the messages over the network using HTTP and converting the results received back to the corresponding .NET data types.
Now, add mark up in Home.aspx to receive inputs. Here's my mark up.

```
<table>
```

```
    <tr>
        <td>First Number:</td>
        <td><input type="text" id="Text1" runat="server" /></td>
    </tr>
    <tr>
        <td>Second Number:</td>
        <td><input type="text" id="Text2" runat="server" /></td>
    </tr>
    <tr>
        <td><input type="button" onserverclick="AddNumber" value="Add" runat="server" /></td>
        <td><div id="divSum" runat="server"></div>
        <div id="divSumThroughJson" runat="server"></div></td>
    </tr>
</table>
```

Home.aspx.cs

```
using System;
using ServiceConsumer.MyServiceReference;

namespace ServiceConsumer
{
    public partial class Home : System.Web.UI.Page
    {
        protected void Page_Load(object sender, EventArgs e)
        {}
        protected void AddNumber(object Sender, EventArgs E)
        {
            int Num1, Num2;
            int.TryParse(txtFirstNum.Value, out Num1);
            int.TryParse(txtSecondNum.Value, out Num2);
            // creating object of MyService proxy class
            var ObjMyService = new MyServiceSoapClient();

            // Invoke service method through service proxy
            divSum.InnerHtml = ObjMyService.SumOfNums(Num1, Num2).ToString();
            var ObjSumClass = new SumClass { First = Num1, Second = Num2 };
            var ObjSerializer = new JavaScriptSerializer();
            var JsonStr = ObjSerializer.Serialize(ObjSumClass);
            divSumThroughJson.InnerHtml =ObjMyServiceProxy.GetSumThroughObject(JsonStr).Sum.ToString();
        }
    }
}
```

How to consume a web service from a client script

Add another project to your solution. I named it “ConsumeServiceFromClientScript”. Add a web form (let's say Default.aspx).

Creating a JavaScript proxy

JavaScript proxies can be automatically generated by using the ASP.NET AJAX ScriptManager control's Services property. You can define one or more services that a page can call asynchronously to send or receive data using the ASP.NET AJAX “ServiceReference” control and assigning the Web Service URL to the control's “**Path**” property.

Add a ScriptManager control like the following inside the form tag:

```
<asp:ScriptManager runat="server">
    <Services>
        <asp:ServiceReference Path="http://localhost/WebServiceForBlog/MyService.asmx"/>
     </Services>
</asp:ScriptManager>
```

When you add a reference to a web service (MyService.asmx) from a page using a ScriptManager control, a JavaScript proxy is generated dynamically and referenced by the page. Now if you the check page source in a browser, you can notice that:

http://localhost/MyWebServiceDemoService/MyService.asmx/js

Or:

http://localhost/MyWebServiceDemoService/MyService.asmx/jsdebug

Is included in your page with a script tag depending on whether debugging is enabled In your project.

Add an empty Web form to your project. I named it “Default.aspx”.

Default.aspx

```
<body>
   <form id="form1" runat="server">
      <asp:ScriptManager runat="server">
```

```
        <Services>
            <asp:ServiceReference Path="http://localhost/WebServiceForBlog/MyService.asmx"/>
        </Services>
    </asp:ScriptManager>
    <div>
        <table>
            <tr>
                <td>First Number:</td>
                <td><input type="text" id="txtFirstNum" /></td>
            </tr>
            <tr>
                <td>Second Number:</td>
                <td><input type="text" id="txtSecondNum" /></td>
            </tr>
            <tr>
                <td><input type="button" onclick="AddNumber();" value="Add" /></td>
                <td><div id="divSum1"></div><div id="divSum2"></div><div id="divSum3"></div><div id="div
Sum4"></div><div id="divSum5"></div></td>
            </tr>
        </table>
    </div>
</form>
<script type="text/javascript" src="Scripts/jquery-1.7.1.js"></script>
<script type="text/javascript" src="Scripts/json2.js"></script>
<script type="text/javascript">
    function AddNumber() {
        var FirstNum = $('#txtFirstNum').val();
        var SecondNum = $('#txtSecondNum').val();

        $.ajax({
            type: "POST",
            url: "http://localhost/WebServiceForBlog/MyService.asmx/SumOfNums",
            data: "First=" + FirstNum + "&Second=" + SecondNum,
     // the data in form-encoded format, it would appear on a querystring
```

```
            //contentType: "application/x-www-form-urlencoded; //charset=UTF-
8"                              //form  encoding is the default one
             dataType: "text",
             crossDomain: true,
             success: function (data) {
                 $('#divSum1').html(data);
             }
         });

         // i/p in JSON format, response as JSON object
         $.ajax({
             type: "POST",
             url: "http://localhost/WebServiceForBlog/MyService.asmx/SumOfNums",
             data: "{First:" + FirstNum + ",Second:" + SecondNum + "}",
             contentType:"application/json; charset=utf-8",// What I am ing
             dataType: "json", // What I am expecting
             crossDomain: true,
             success: function (data) {
                 $('#divSum2').html(data.d);
             }
         });

     // i/p as JSON object, response as plain text
         $.ajax({
             type: "POST",
             url: "http://localhost/WebServiceForBlog/MyService.asmx/SumOfNums",
             data: { First: FirstNum, Second: SecondNum },
             contentType: "application/x-www-form-urlencoded; charset=UTF-8",
             crossDomain: true,
             dataType: "text",
             success: function (data) {
                 $("#divSum3").html(data);
             }
         });
```

```
    // Url encoded stringified JSON object as I/p & text response
      var ObjSum = new Object();
      ObjSum.First = FirstNum;
      ObjSum.Second = SecondNum;
      $.ajax({
        type: "POST",
        url: "http://localhost/WebServiceForBlog/MyService.asmx/GetSumThroughObject",
        data: "JsonStr=" + JSON.stringify(ObjSum),
        dataType: "text",
        crossDomain: true,
        success: function (data) {
          $('#divSum4').html($(data).find('Sum').text());
        }
      });

      // Call SOAP-XML web services directly
      var SoapMessage = '<soap:Envelope xmlns:xsi="http://www.w3.org/2001/XMLSchema-
instance" xmlns:xsd="http://www.w3.org/2001/XMLSchema" xmlns:soap="http://schemas.xmlsoap.org/soap/env
elope/"> \
                          <soap:Body> \
                            <SumOfNums xmlns="http://tempuri.org/"> \
                              <First>'+FirstNum+'</First> \
                              <Second>'+SecondNum+'</Second> \
                            </SumOfNums> \
                          </soap:Body> \
                        </soap:Envelope>';
      $.ajax({
        url: "http://localhost/WebServiceForBlog/MyService.asmx?op=SumOfNums",
        type: "POST",
        dataType: "xml",
        data: SoapMessage,
        complete: ShowResult,
        contentType: "text/xml; charset=utf-8"
```

```
            });
        }
        function ShowResult(xmlHttpRequest, status) {
            var SoapResponse = $(xmlHttpRequest.responseXML).find('SumOfNumsResult').text();
            $('#divSum5').html(SoapResponse);
        }
    </script>
</body>
```

Note: For demo purposes, I have hard-coded the service URL. But it's not a good practice. You can keep the service URL in the web.config and use RegisterClientScriptBlock or RegisterStartUpScript depending on your requirements to use this service URL in client script.

Call SOAP-XML Web Services directly

In order to call the web service, we need to supply an XML message that matches the operation definition specified by the web service's WSDL. In the test page "**http://localhost/WebServiceForBlog/MyService.asmx?op=SumOfNums**" you can see a sample of SOAP 1.1/1.2 request and response format. From which you can get the operations schema. Now, you just need to exchange the type names for the operation's parameters, with their actual values. Copy the SOAP envelope to the Default.aspx page and replace the place holders (type names) with an actual value.

A sample for this type of call is included in the code snippet above. The variable "SoapMessage" in the preceding example contains the complete XML message that we're going to send to the web service. On completion of the ajax request, we will call a call back method named "ShowResult".

Note: .NET 1.x supports SOAP 1.1 only. From .NET 2.0 Both SOAP1.1 and SOAP1.2 are supported unless one is restricted explicitly. If you want to change this, you can disable either one through the web.config file:

```
<configuration>
      <system.web>
    <webServices>
           <protocols>
           <!-- To disable SOAP 1.2 -->
           <remove name="HttpSoap12"/>
           <!-- To disable SOAP 1.1 -->
           <remove name="HttpSoap"/>
           </protocols>
    </webServices>
       </system.web>
</configuration>
```

Window Services

A Windows service is a long-running application that can be started automatically when our system is started. We can pause our service and resume or even restart it if need be. Once we have created a Windows service, we can install it in our system using the **InstallUtil.exe** command line utility.

Need of a Windows service

We would typically want to use Windows services when we need to build and implement long-running jobs that would be executed at predefined intervals of time. Our Windows service would continue to run in the background while the applications in our system can execute at the same time. Note that a Windows service can continue to execute in the background even if no one has logged into our system.

Implementing a Windows service in C#

To create a Windows service in Visual Studio 2015, follow these steps:

1. Open Visual Studio IDE
2. Click on File -> New -> Project
3. Select "Windows Service" project template
4. Specify a name for the Windows service
5. Click OK

This will create a new Windows service project in Visual Studio. When working with Windows services, the two main classes we would need are the **service** and the **service installer** classes. The service class is the main service file; this would typically contain all that our service is supposed to do. The other class is the service installer that contains the necessary code and metadata needed to setup our service.

Here's how the source code of the default service (named Service1 and stored in the Service1.cs file) looks like.

```
using System.ServiceProcess;
namespace IDGWindowsService
{
   public partial class Service1 : ServiceBase
   {
      public Service1()
      {
         InitializeComponent();
      }
      protected override void OnStart(string[] args)
      {
      }
```

```
        protected override void OnStop()
        {
        }
    }
}
```

Note that all our Windows services should extend the **ServiceBase** class. As we can see in the above code listing, there are two important methods in the service class. These include the **OnStart** and **OnStop** methods. While the former is fired each time our service is started by the service control manager, the latter is fired each time the service is stopped by the service control manager.

We can then take advantage of the Timer class to specify an operation (in the OnStart() method of our service class) that we would want to execute at specified intervals of time. We can specify the interval and the event handler to be called each time the interval has elapsed. And, we should disable the timer in the OnStop() method, the method that is called each time the Windows service is stopped.

Next, double click on the service file in the solution explorer window, right click and then select "**Add Installer**" to create the service installer file. Now, select the service installer, right click, and click on "Properties" to open the properties window. We can specify the service name, description, start type, modifiers, etc. for our service from the properties window of the service installer.

The two properties that are of importance are the **service name** and the **start type**. While the former refers to the name of the service that would be installed in our system, the second refers to how the service should be started. There are three possible options for the start type of the service -- **Automatic**, **Manual**, and **Disabled**. All of these start type options are self -- explanatory anyway.

Once done, all we would now need to do is compile the solution to create the executable file for the Windows service we have implemented. Next, we should open the "**Developer Command Prompt**" of the version of Visual Studio that is installed in our system and then use the command line utility named **InstallUtil** to install the service.

If the installation of the service is successful, we would see that the following messages (amongst other messages) would be displayed in the console window, i.e., the Visual Studio Developer Command Prompt.

The Commit phase completed successfully.

The transacted install has completed.

Now, if the start type of our Windows service is set to "**Manual**", we should start our service manually. To do this, go to Start -> Run and then type "**Services.msc**". When the "Services" window opens, locate our Windows service and start it manually. This is all we need to do to build and execute our Windows service.

ASP.NET Web Forms

Web Forms are web pages built on the ASP.NET Technology. It executes on the server and generates output to the browser. It is compatible to any browser to any language supported by .NET common language runtime. It is flexible and allows us to create and add custom controls.

We can use Visual Studio to create ASP.NET Web Forms. It is an IDE (Integrated Development Environment) that allows us to drag and drop server controls to the web forms. It also allows us to set properties, events and methods for the controls. To write business logic, we can choose any .NET language like: Visual Basic or Visual C#.

Web Forms are made up of two components: the visual portion (the ASPX file), and the code behind the form, which resides in a separate class file.

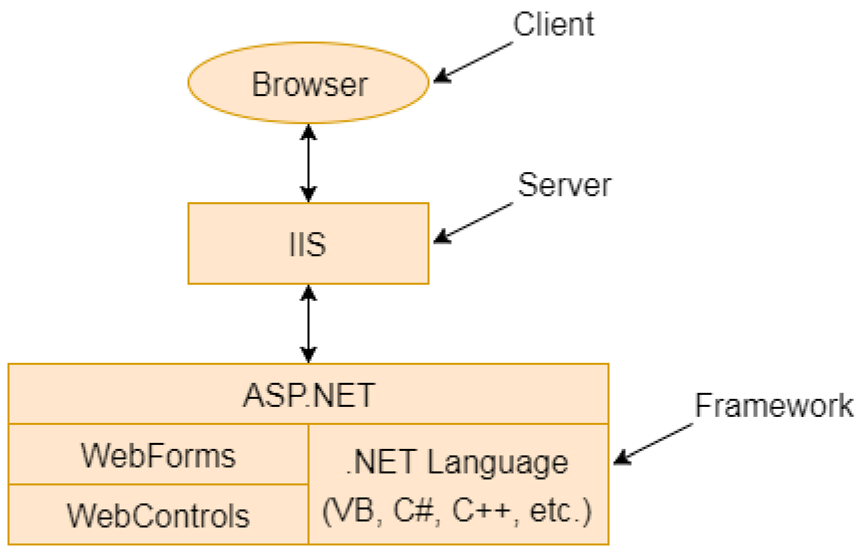


Fig: This diagram shows the components of the ASP.NET

The main purpose of Web Forms is to overcome the limitations of ASP and separate view from the application logic.

ASP.NET provides various controls like:

ASP.NET provides web forms controls, which are used to create HTML components. These controls are categories as server-based and client-based.

There are various server controls available for the web forms in ASP.NET:

- Label
- Checkbox
- Text Box
- Radio Button and many more.

Let us discuss all these controls one by one in detail.

Label:

Label control is used to display textual information or any message to the user on the web forms. It is mainly used to create captions for the other controls, such as a textbox. To create a label, you can either write code or use the drag and drop facility of the visual studio.

Syntax:

```
< asp:LabelID="Label1" runat="server" Text="Label" ></asp:Label>
```

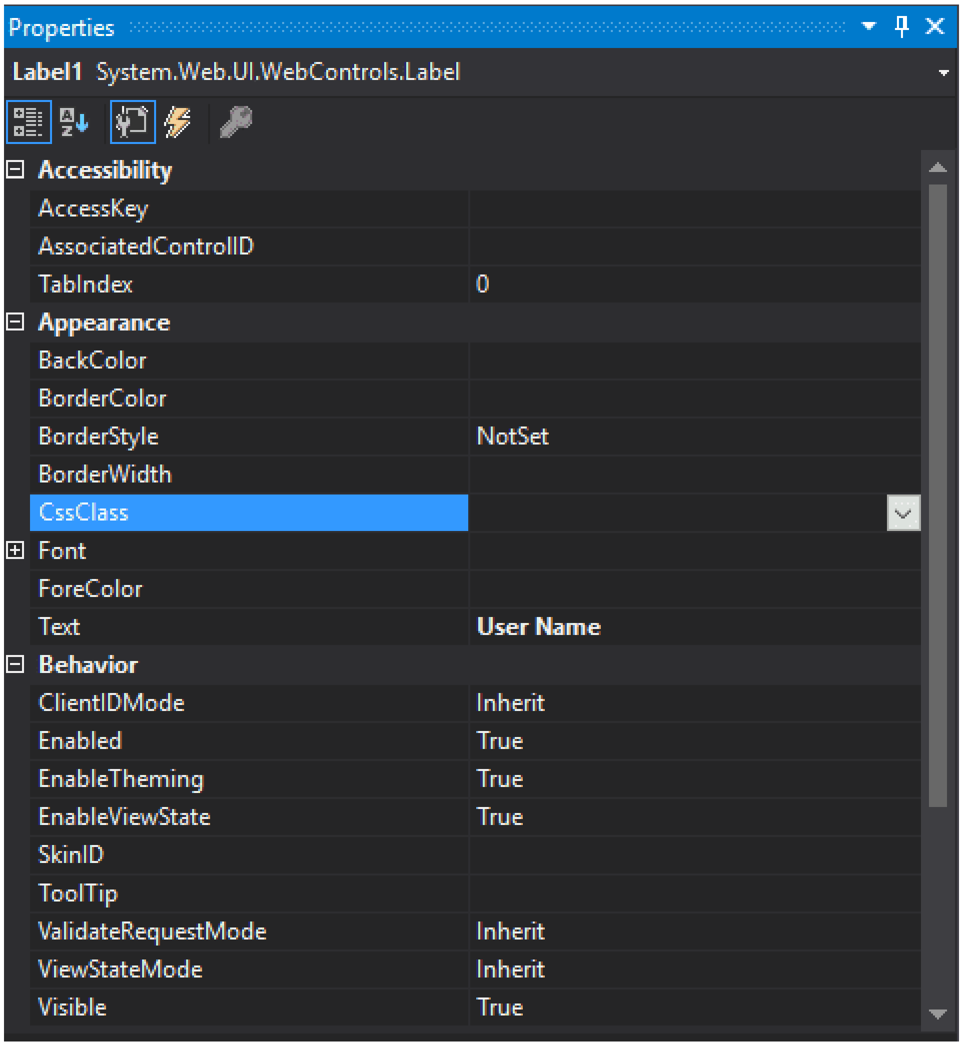
Properties of Label Control:

- AccessKey: This is used to set keyboard shortcut for the label.
- BackColor: This is used to set the background color of the label.
- BorderColor: This is used to set the border color of the label.
- BorderWidth: This is used to set the width of the border of the label.
- Font: This is used to set the font for the label text.
- ForeColor: This is used to set the color of the label text.
- Text: This is used to set the text to be shown for the label.
- ToolTip: This displays the text when the mouse is over the label.
- Visible: This is used to set visibility of control on the form.
- Height: It is used to set the height of the control.
- Width: This sets the width of the control.

Example:

```
// WebControls.aspx
<%@ Page Language="C#" AutoEventWireup="true" CodeBehind="WebControls.aspx.cs"
Inherits="WebFormsControlls.WebControls" %>
<!DOCTYPE html>
<html xmlns="https://www.w3.org/1999/xhtml">
<head runat="server">
    <title></title>
    <style type="text/css">
        .auto-style1 {
            width: 100%;
        }
        .auto-style2 {
            margin-left: 0px;
        }
        .auto-style3 {
            width: 121px;
        }
    </style>
</head>
<body>
    <form id="form1" runat="server">
        <div>
           <h4>Provide the Following Details:</h4>
            <table class="auto-style1">
                <tr>
                    <td class="auto-style3">
                        <asp:Label ID="lblName" runat="server" Text="User Name"></asp:Label></td>
                    <td>
                        <asp:TextBox ID="txtName" runat="server" CssClass="auto-style2"></asp:TextBox></td>
                </tr>
                <tr>
                    <td class="auto-style3">
                        <asp:Label ID="lblFile" runat="server" Text="Upload a File"></asp:Label></td>
                    <td>
                        <asp:FileUpload ID="fileUploadId" runat="server" /></td>
                </tr>
            </table>
        </div>
    </form>
</body>
</html>
```

The property window of label control looks like:

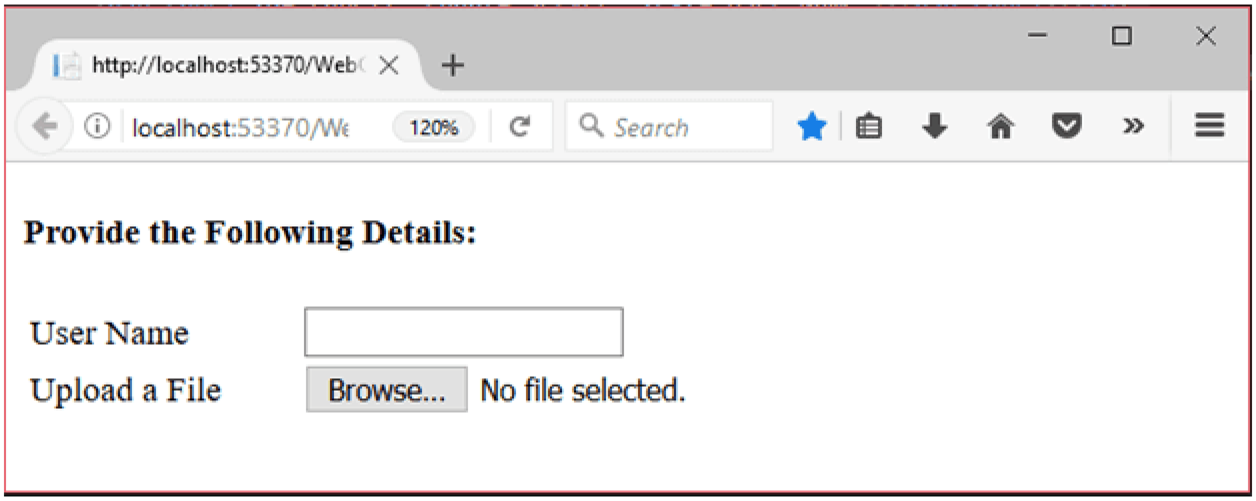


Properties

Label1 System.Web.UI.WebControls.Label

Accessibility	
AccessKey	
AssociatedControlID	
TabIndex	0
Appearance	
BackColor	
BorderColor	
BorderStyle	NotSet
BorderWidth	
CssClass	
Font	
ForeColor	
Text	**User Name**
Behavior	
ClientIDMode	Inherit
Enabled	True
EnableTheming	True
EnableViewState	True
SkinID	
ToolTip	
ValidateRequestMode	Inherit
ViewStateMode	Inherit
Visible	True

Output:



Provide the Following Details:

User Name

Upload a File [Browse...] No file selected.

Text Box:

It is an input control, which is used to take user input. To create a Text Box you can either write code or use the drag and drop facility web forms controls of visual studio IDE.

Syntax:

```
< asp:TextBoxID="TextBox1" runat="server" ></asp:TextBox>
```

Properties of Text Box Control:

- AccessKey: This is used to set keyboard shortcut for the control.
- BackColor: This is used to set background color of the control.
- BorderColor: This is used to set border color of the control.
- BorderWidth: This is used to set width of border of the control.
- Font: This is used to set font for the control text.
- ForeColor: This is used to set color of the control text.
- Text: This is used to set text to be shown for the control.
- ToolTip: This displays the text when mouse is over the control.
- Visible: This is used to set visibility of control on the form.
- Height: This is used to set height of the control.
- Width: This is used to set width of the control.
- MaxLength: This is used to set maximum number of characters that can be entered.
- Readonly: This is used to make control readonly.

Example

```
// WebControls.aspx
<%@ Page Language="C#" AutoEventWireup="true" CodeBehind="WebControls.aspx.cs"
Inherits="WebFormsControlls.WebControls" %>
<!DOCTYPE html>
<html xmlns="https://www.w3.org/1999/xhtml">
<head runat="server">
   <title></title>
</head>
<body>
   <form id="form1" runat="server">
      <div>
         <asp:Label ID="lblName" runat="server">User Name</asp:Label>
<asp:TextBox ID="txtName" runat="server" ToolTip="Enter User Name"></asp:TextBox>
```

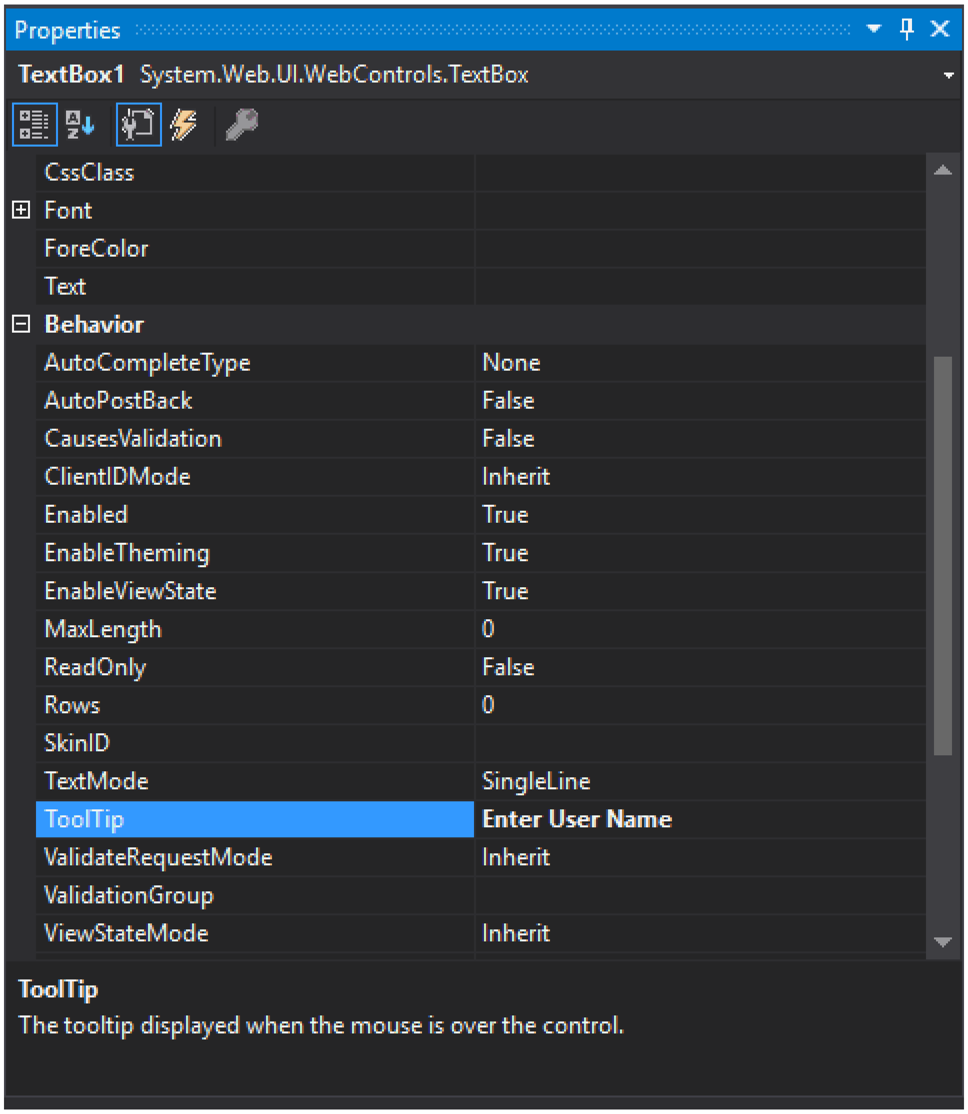
```
        </div>
        <p>
        <asp:Button ID="btnSubmit" runat="server" Text="Submit" OnClick="SubmitButton_Click" />
        </p>
        <br />
    </form>
     <asp:Label ID="lblInput" runat="server"></asp:Label>
</body>
</html>
```

Code Behind

// WebControls.aspx.cs

```
using System;

using System.Collections.Generic;

using System.Linq;

using System.Web;

using System.Web.UI;

using System.Web.UI.WebControls;

namespace WebFormsControlls

{

    public partial class WebControls : System.Web.UI.Page

    {

        protected void SubmitButton_Click(object sender, EventArgs e)

        {
            lblInput.Text = lblName.Text;

        }

    }

}
```

The property window of text box control looks like:



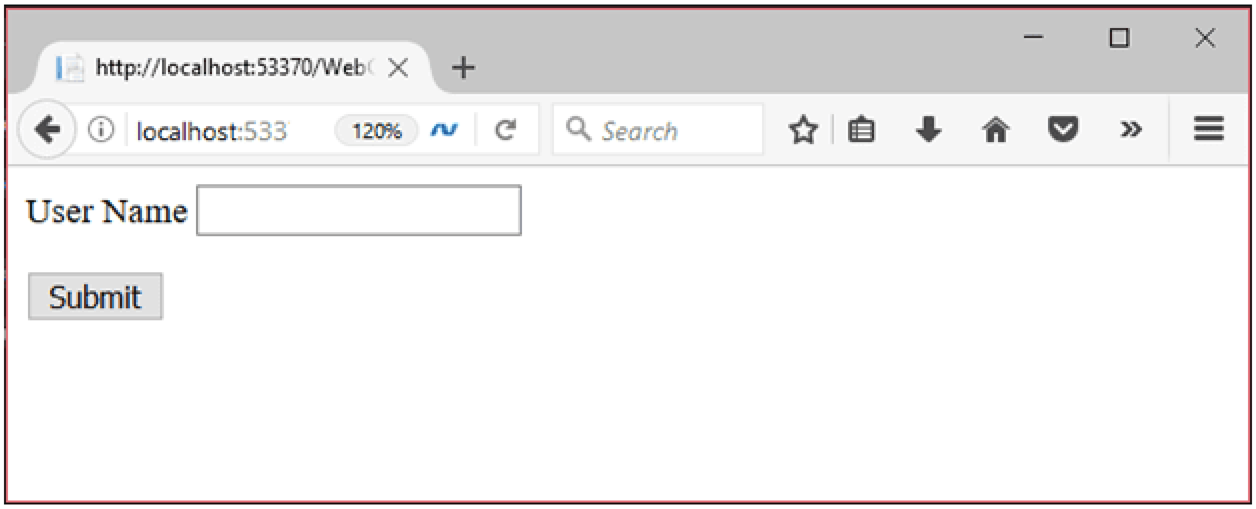
Properties

TextBox1 System.Web.UI.WebControls.TextBox

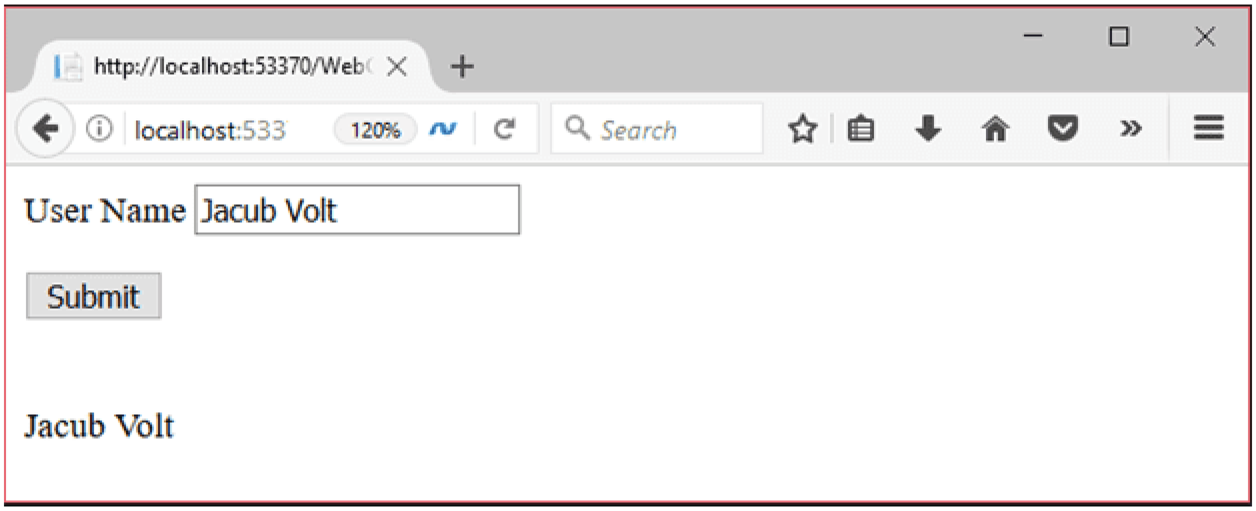
CssClass	
Font	
ForeColor	
Text	
Behavior	
AutoCompleteType	None
AutoPostBack	False
CausesValidation	False
ClientIDMode	Inherit
Enabled	True
EnableTheming	True
EnableViewState	True
MaxLength	0
ReadOnly	False
Rows	0
SkinID	
TextMode	SingleLine
ToolTip	**Enter User Name**
ValidateRequestMode	Inherit
ValidationGroup	
ViewStateMode	Inherit

ToolTip
The tooltip displayed when the mouse is over the control.

Output:



It displays the user input when the user submits the input to the server.



User Name: Jacub Volt

Jacub Volt

List Box: It allows the selection of single as well as multiple items in the list, unlike the dropdown control, which allows the selection of single item at a time.

Syntax:

```
<asp: ListBox id="ListBox1" Rows="6" Width="100px" SelectionMode="Single" runat="server"> </asp: ListBox>
```

Properties of List Box Control:

- Items.Count: It is used to return the total number of items in the list box.
- Items.Clear: It will clear all the selected items from the list box.
- SelectedItem.Text: It will return the text of the selected item.
- Items.Remove(name): It removes the item with text name.
- Items.RemoveAt(int index): It removes the item present at the given index.

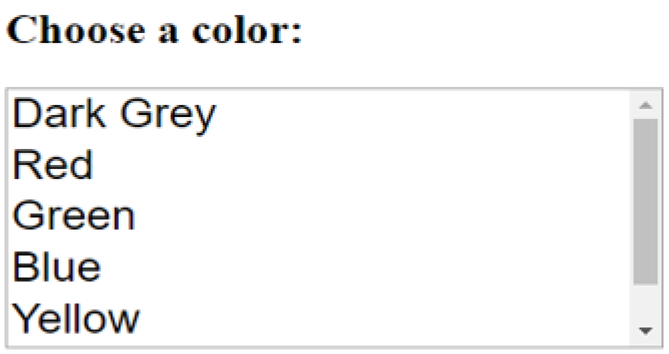
- Items.Insert(int index, text): It inserts the text at the given index.
- SelectedItem: It returns the index of the selected item.
- SelectionMode: It specifies the selection mode single or multiple.

Example:

```
<%@ Page Language="C#" AutoEventWireup="true" CodeBehind="WebForm1.aspx.cs"
Inherits="WebApplication1.WebForm1" %>

<!DOCTYPE html>

<html xmlns="https://www.w3.org/1999/xhtml">

<head runat="server">

<title> An example of ASP.Net ListBox</title>

</head>

<body>

<form id="submitColor" runat="server">

<div>

<h2>Choose a color:</h2>

<asp: ListBox ID ='lstColor’ runat = 'server' AutoPostBack = 'true' Font-Size = 'X-Large' Rows = '5'

ForeColors = 'Tomato' Width = '350' >

<asp: ListItem> Dark Grey </asp: ListItem>

<asp: ListItem> Red </asp: ListItem>

<asp: ListItem> Green </asp: ListItem>

<asp: ListItem> Blue </asp:ListItem>
```

```
<asp: ListItem> Yellow </asp: ListItem>

<asp: ListItem> Black </asp: ListItem>

</asp: ListBox>

</div>

</form>

</body>

</html>
```

Output:



Radio Button: Radio Button control is an input control that is used to takes input from the user. It allows the user to select a choice from the group of choices.

Syntax:

```
< asp:RadioButtonID="RadioButton1" runat="server" Text="Male" GroupName="gender"/>
```

Properties of Radio Button Control:

- AccessKey: This is used to set keyboard shortcut for the control.
- BackColor: This is used to set the background color of the control.
- BorderColor: This is used to set the border color of the control.
- BorderWidth: This is used to set the width of the border of the control.
- Font: This is used to set the font for the control text.

- ForeColor: This is used to set the color of the control text.
- Text: This is used to set the text to be shown for the control.
- ToolTip: This displays the text when the mouse is over the control.
- Visible: This is used to set visibility of control on the form.
- Height: It is used to set the height of the control.
- Width: This sets the width of the control.
- GroupName: This is used to set the name of the radio button group.

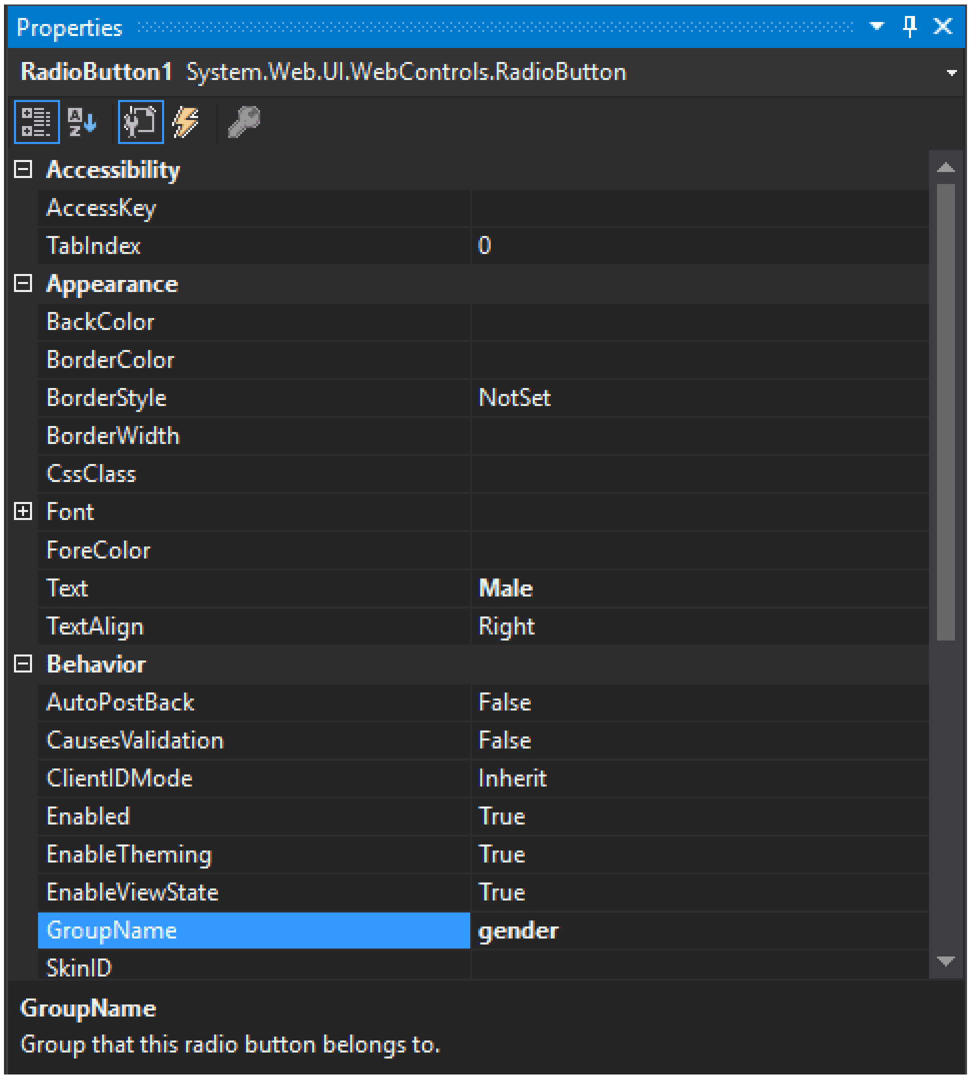
Example:

```
// WebControls.aspx
<%@ Page Language="C#" AutoEventWireup="true" CodeBehind="WebControls.aspx.cs"
Inherits="WebFormsControlls.WebControls" %>
<!DOCTYPE html>
<html xmlns="https://www.w3.org/1999/xhtml">
<head runat="server">
    <title></title>
</head>
<body>
    <form id="submitGender" runat="server">
        <div>
            <asp:RadioButton ID="rdMale" runat="server" Text="Male" GroupName="gender" />
            <asp:RadioButton ID="rdFemale" runat="server" Text="Female" GroupName="gender" />
        </div>
        <p>
            <asp:Button ID="btnClick" runat="server" Text="Submit" OnClick="Button1_Click" style="width: 61px"
/>
        </p>
    </form>
    <asp:Label runat="server" id="genderId"></asp:Label>
</body>
</html>
```

Code Behind

```
// WebControls.aspx.cs
using System;
using System.Collections.Generic;
using System.Linq;
using System.Web;
using System.Web.UI;
using System.Web.UI.WebControls;
namespace WebFormsControlls
{
    public partial class WebControls : System.Web.UI.Page
    {
        protected void Button1_Click(object sender, EventArgs e)
        {
            genderId.Text = "";
            if (rdMale.Checked)
            {
                genderId.Text = "Your gender is "+rdMale.Text;
            }
            else genderId.Text = "Your gender is "+rdFemale.Text;

        }
    }
}
```

The property window of radio button control looks like:



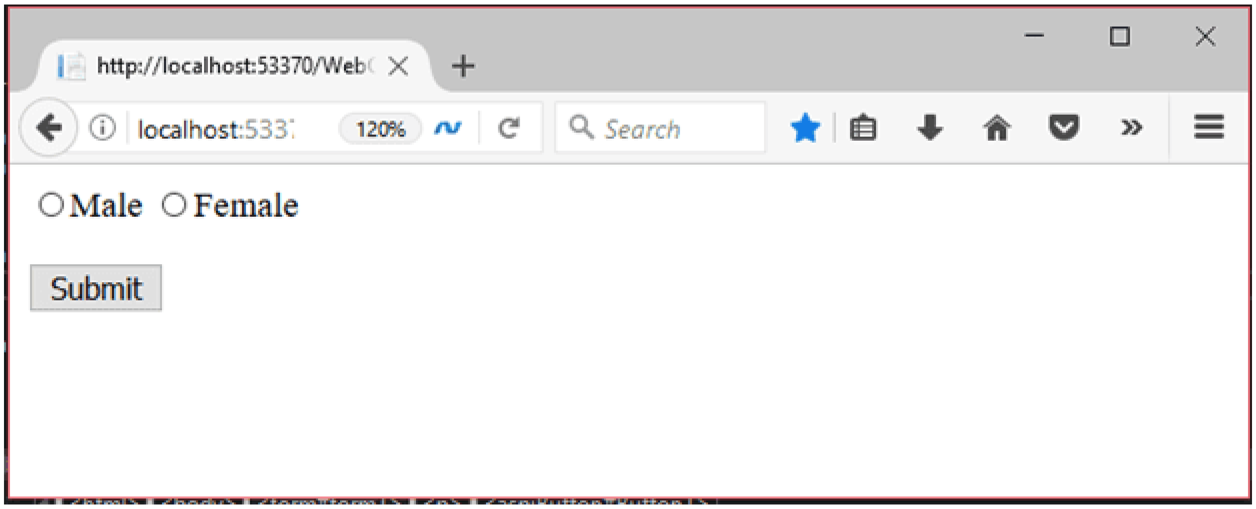
Properties

RadioButton1 System.Web.UI.WebControls.RadioButton

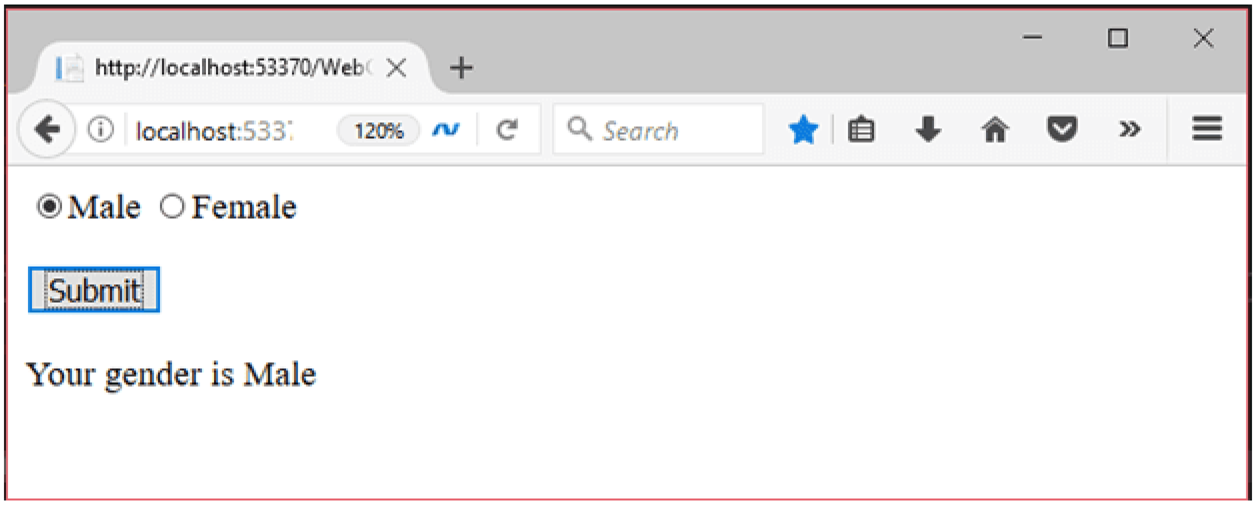
Accessibility	
AccessKey	
TabIndex	0
Appearance	
BackColor	
BorderColor	
BorderStyle	NotSet
BorderWidth	
CssClass	
Font	
ForeColor	
Text	**Male**
TextAlign	Right
Behavior	
AutoPostBack	False
CausesValidation	False
ClientIDMode	Inherit
Enabled	True
EnableTheming	True
EnableViewState	True
GroupName	**gender**
SkinID	

GroupName
Group that this radio button belongs to.

Output:



When the user selects the gender.



Your gender is Male

Checkbox: Checkbox control is used to get multiple inputs from the user. It also allows the user to select choices from the set of choices. It takes the user input in yes or no format and is useful when we want multiple choices from the user.

Syntax:

```
< asp:CheckBox ID="CheckBox2" runat="server" Text="J2EE"/>
```

Properties of Checkbox Control:

- AccessKey: This is used to set keyboard shortcut for web forms controls
- BackColor: This is used to set the background color of the control.
- BorderColor: This is used to set the border color of the control.
- BorderWidth: This is used to set the width of the border of the control.
- Font: This is used to set the font for the control text.
- ForeColor: This is used to set the color of the control text.
- Text: This is used to set the text to be shown for the control.
- ToolTip: This displays the text when the mouse is over the control.
- Visible: This is used to set visibility of control on the form.
- Height: It is used to set the height of the control.
- Width: This sets the width of the control.
- Checked: This is used to set check state of the control, either true or false.

Example:

// WebControls.aspx

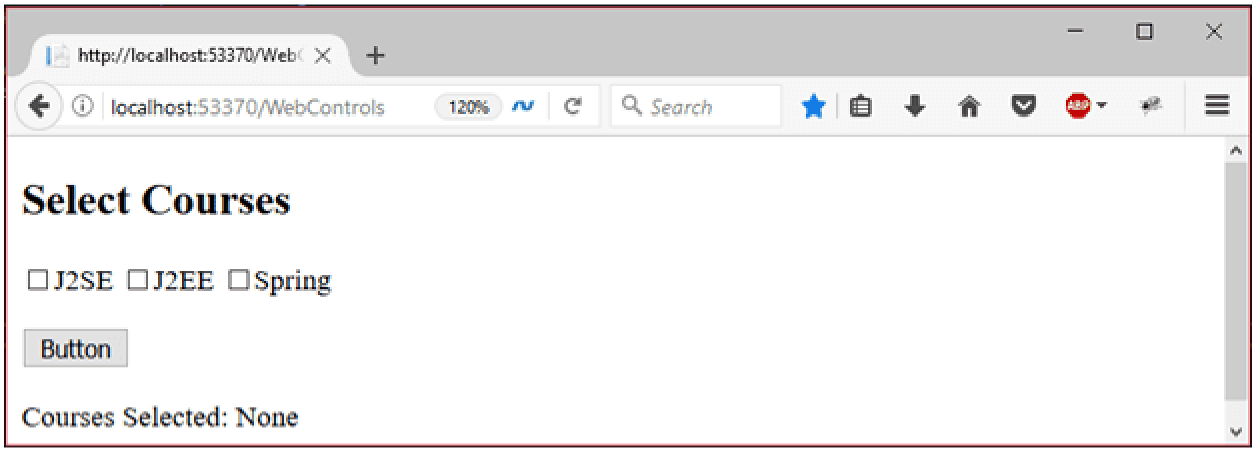
```
<%@ Page Language="C#" AutoEventWireup="true" CodeBehind="WebControls.aspx.cs"
Inherits="WebFormsControlls.WebControls" %>
<!DOCTYPE html>
<html xmlns="https://www.w3.org/1999/xhtml">
<head runat="server">
    <title></title>
</head>
<body>
    <form id="submitCourse" runat="server">
        <div>
            <h2>Select Courses</h2>
            <asp:CheckBox ID="chkJ2SE" runat="server" Text="J2SE" />
            <asp:CheckBox ID="chkJ2EE" runat="server" Text="J2EE" />
            <asp:CheckBox ID="chkSpring" runat="server" Text="Spring" />
        </div>
        <p>
            <asp:Button ID="btnClick" runat="server" Text="Button" OnClick="Button1_Click" />
        </p>
    </form>
    <p>
        Courses Selected: <asp:Label runat="server" ID="ShowCourses"></asp:Label>
    </p>
</body>
</html>
```

Code Behind

```
// WebControls.aspx.cs
using System;
using System.Collections.Generic;
using System.Linq;
using System.Web;
using System.Web.UI;
using System.Web.UI.WebControls;
```

```
namespace WebFormsControlls
{
    public partial class WebControls : System.Web.UI.Page
    {
        protected void Page_Load(object sender, EventArgs e)
        {
            ShowCourses.Text = "None";
        }
        protected void Button1_Click(object sender, EventArgs e)
        {
            var message = "" ;
            if (chkJ2SE.Checked)
            {
                message = chkJ2SE.Text+" ";
            }
            if (chkJ2EE.Checked)
            {
                message += chkJ2EE.Text + " ";
            }
            if (chkSpring.Checked)
            {
                message += chkSpring.Text;
            }
            ShowCourses.Text = message;
        }
    }
}
```

Output:



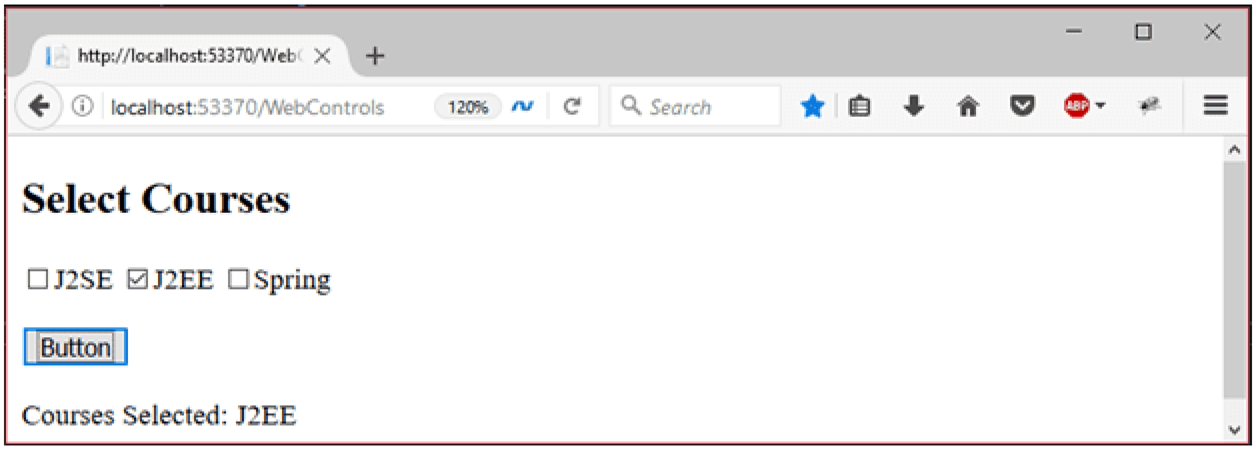
Select Courses

☐J2SE ☐J2EE ☐Spring

Button

Courses Selected: None

After Selection



Select Courses

☐J2SE ☑J2EE ☐Spring

Button

Courses Selected: J2EE

Button: Button control is used to perform events and is also used to submit client requests to the server.

Syntax:

```
< asp:ButtonID="Button1" runat="server" Text="Submit" BorderStyle="Solid" ToolTip="Submit"/>
```

Properties of Button Control:

- AccessKey: This is used to set keyboard shortcut for the control.
- BackColor: This is used to set the background color of the control.
- BorderColor: This is used to set the border color of the control.
- BorderWidth: This is used to set the width of the border of the control.
- Font: This is used to set the font for the control text.
- ForeColor: This is used to set the color of the control text.
- Text: This is used to set the text to be shown for the control.
- ToolTip: This displays the text when the mouse is over the control.

- Visible: This is used to set visibility of control on the form.
- Height: It is used to set the height of the control.
- Width: This sets the width of the control.

Example:

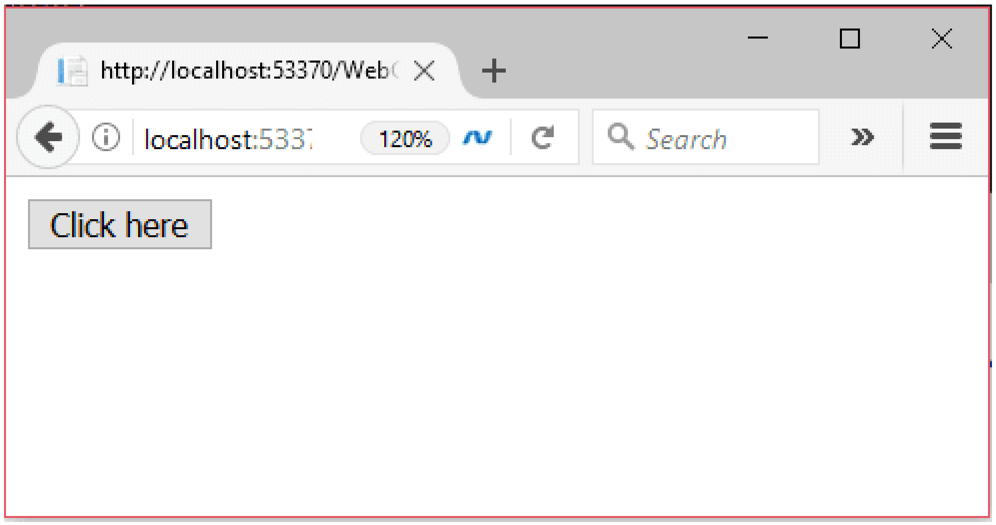
```
// WebControls.aspx
<%@ Page Language="C#" AutoEventWireup="true" CodeBehind="WebControls.aspx.cs"
Inherits="WebFormsControlls.WebControls" %>
<!DOCTYPE html>
<html xmlns="https://www.w3.org/1999/xhtml">
<head runat="server">
    <title></title>
</head>
<body>
    <form id="submitButton" runat="server">
        <div>
            <asp:Button ID="btnClick" runat="server" Text="Click here" OnClick="Button1_Click" />
        </div>
    </form>
    <br />
    <asp:Label ID="lblClick" runat="server"></asp:Label>
</body>
</html>
```

Code Behind

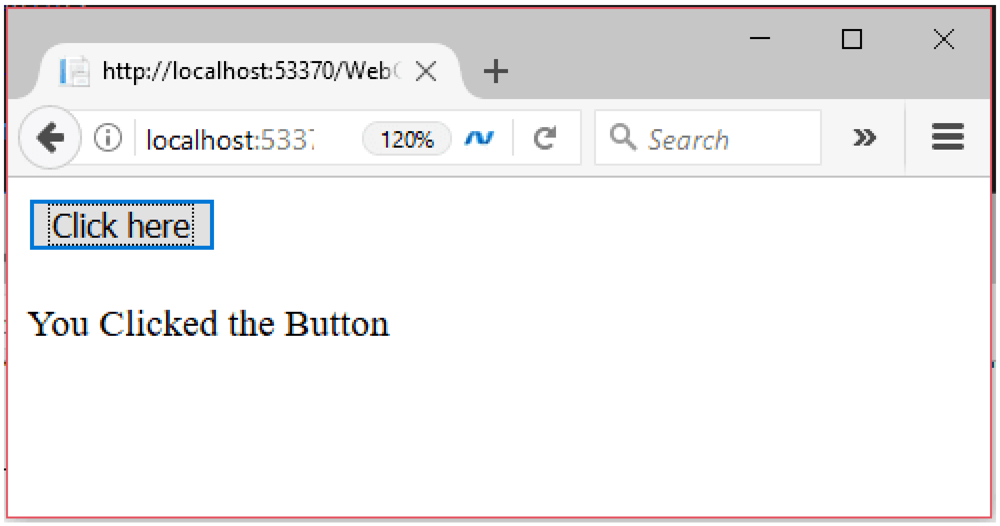
```
// WebControls.aspx.cs
using System;
using System.Collections.Generic;
using System.Linq;
using System.Web;
using System.Web.UI;
using System.Web.UI.WebControls;
namespace WebFormsControlls
```

```
{
    public partial class WebControls : System.Web.UI.Page
    {
        protected void Button1_Click(object sender, EventArgs e)
        {
            lblClick.Text = "You Clicked The Button.";
        }
    }
}
```

Output:



After clicking on the button



You Clicked the Button

Event Handler in ASP.NET: ASP.NET provides an important feature for event handling to WebForms. While working with a web form, you can add events to controls. An event is something, which happens when an action is performed. The most common event is the clicking of a button on a form.

Example: When you enter the first value and the second value and click on the button Sum, then an event will happen, which will add the values and populate it in the text box Sum.

// EventHandling.aspx

```
<%@ Page Language="C#" AutoEventWireup="true" CodeBehind="EnventHandling.aspx.cs"
Inherits="asp.netexample.EnventHandling" %>
<!DOCTYPE html>
<html xmlns="https://www.w3.org/1999/xhtml">
<head runat="server">
<title></title>
<style type="text/css">
.auto-style1 {
width: 100%;
        }
.auto-style2 {
width: 108px;
        }
</style>
</head>
<body>
<form id="submitSum" runat="server">
<div>
<table class="auto-style1">
<tr>
<td class="auto-style2">First value</td>
<td>
<asp:TextBox ID="firstvalue" runat="server"></asp:TextBox>
</td>
</tr>
<tr>
<td class="auto-style2">Second value</td>
```

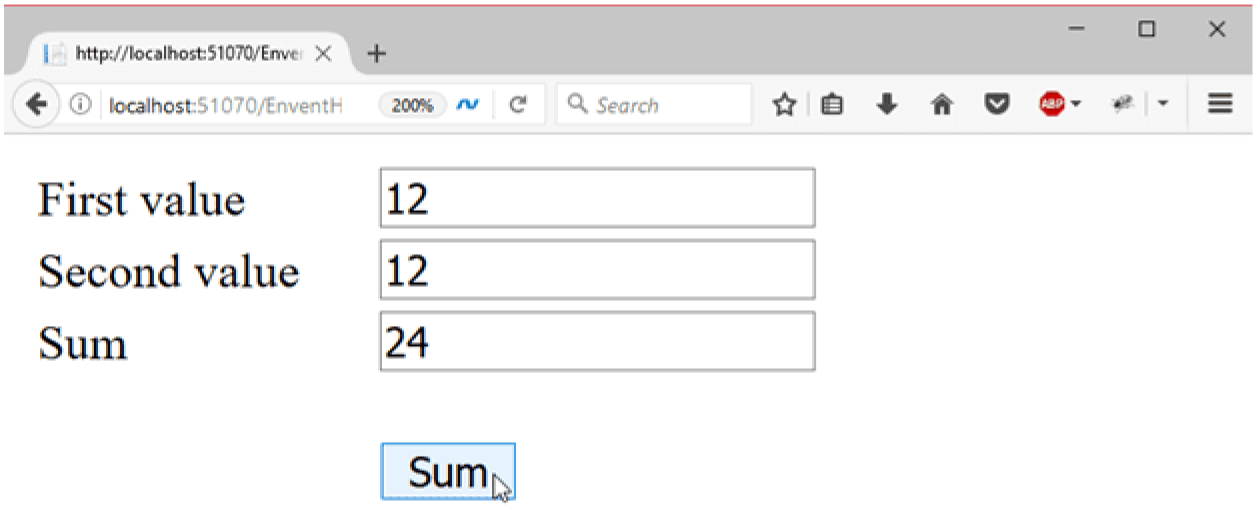
```
<td>
<asp:TextBox ID="secondvalue" runat="server"></asp:TextBox>
</td>
</tr>
<tr>
<td class="auto-style2">Sum</td>
<td>
<asp:TextBox ID="total" runat="server"></asp:TextBox>
</td>
</tr>
<tr>
<td class="auto-style2"> </td>
<td>
<br/>
<asp:Button ID="Button1" runat="server" OnClick="Button1_Click"Text="Sum"/>
</td>
</tr>
</table>
</div>
</form>
</body>
</html>
```

Code Behind

```
// EventHandling.aspx.cs
using System;
using System.Collections.Generic;
using System.Linq;
using System.Web;
using System.Web.UI;
using System.Web.UI.WebControls;
namespace asp.netexample
{
```

```
public partial class EnventHandling : System.Web.UI.Page
    {
protected void Button1_Click(object sender, EventArgs e)
        {
int a = Convert.ToInt32(firstvalue.Text);
int b = Convert.ToInt32(secondvalue.Text);
            total.Text = (a + b).ToString();
        }
    }
}
```

Output:



First value 12
Second value 12
Sum 24
Sum

ADO.NET

ADO.NET stands for **ActiveX Data Object**. It is a database access technology created by Microsoft as part of its .NET framework that can access any kind of data source. ADO.NET provides a bridge between the front end controls and the back end database.

The following are a few of the .NET applications that use ADO.NET to connect to a database, execute commands and retrieve data from the database.

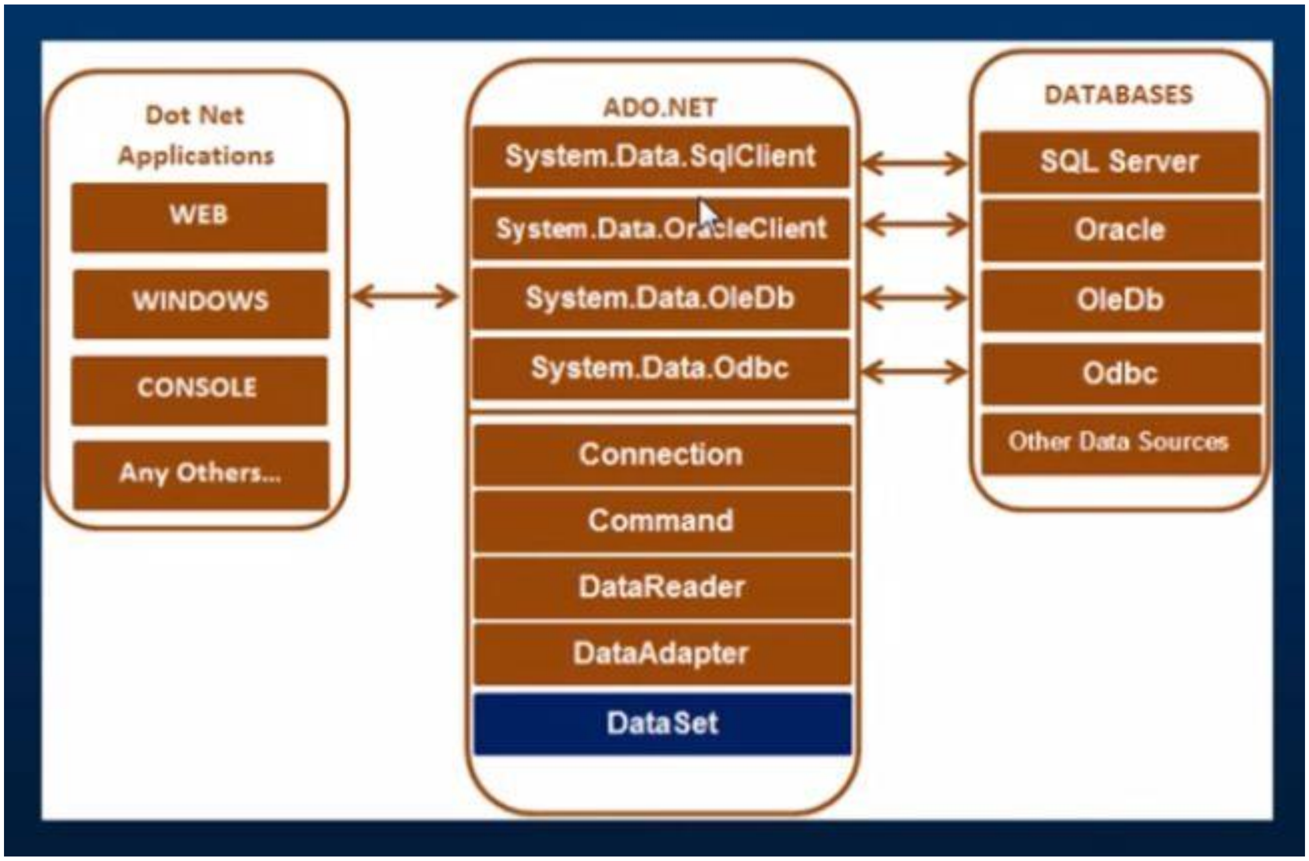
- ASP.NET Web Applications
- Console Applications
- Windows Applications.

Various Connection Architectures

There are the following two types of connection architectures:

1. **Connected architecture:** the application remains connected with the database throughout the processing.
2. **Disconnected architecture:** the application automatically connects/disconnects during the processing. The application uses temporary data on the application side called a **DataSet**.

Understanding ADO.NET and its class library



In this diagram, we can see that there are various types of applications (Web Application, Console Application, Windows Application and so on) that use ADO.NET to connect to databases (SQL Server, Oracle, OleDb, ODBC, XML files and so on).

<u>Important Classes in ADO.NET</u>

We can also observe various classes in the preceding diagram. They are:

1. Connection Class
2. Command Class
3. DataReader Class
4. DataAdaptor Class
5. DataSet.Class

1. Connection Class

In ADO.NET, we use these connection classes to connect to the database. These connection classes also manage transactions and connection pooling.

2. Command Class

The Command class provides methods for storing and executing SQL statements and Stored Procedures. The following are the various commands that are executed by the Command Class.

- **ExecuteReader:** Returns data to the client as rows. This would typically be an SQL select statement or a Stored Procedure that contains one or more select statements. This method returns a DataReader object that can be used to fill a DataTable object or used directly for printing reports and so forth.
- **ExecuteNonQuery:** Executes a command that changes the data in the database, such as an update, delete, or insert statement, or a Stored Procedure that contains one or more of these statements. This method returns an integer that is the number of rows affected by the query.
- **ExecuteScalar:** This method only returns a single value. This kind of query returns a count of rows or a calculated value.
- **ExecuteXMLReader:** (SqlClient classes only) Obtains data from an SQL Server 2000 database using an XML stream. Returns an XML Reader object.

3. DataReader Class

The DataReader is used to retrieve data. It is used in conjunction with the Command class to execute an SQL Select statement and then access the returned rows.

4. DataAdapter Class

The DataAdapter is used to connect DataSets to databases. The DataAdapter is most useful when using data-bound controls in Windows Forms, but it can also be used to provide an easy way to manage the connection between your application and the underlying database tables, views and Stored Procedures.

5. DataSet Class

The DataSet is the heart of ADO.NET. The DataSet is essentially a collection of DataTable objects. In turn each object contains a collection of DataColumn and DataRow objects. The DataSet also contains a Relations collection that can be used to define relations among Data Table Objects.

<u>Advantages of ADO .NET</u>

ADO.NET offers several advantages over previous Microsoft data access technologies, including ADO. The following points will outline these advantages.

- ➢ **Single Object- Oriented API**

The ADO.NET provides a single object-oriented set of classes. There are different data providers to work with different data sources, but the programming model for all these data providers to work in the same way. So if we know how to work with one data provider, we can easily work with others. It’s just a matter of changing class names and connection strings.

The ADO.NET classes are easy to use and to understand because of their object- oriented nature.

We can use more than one data provider to access a single data source. For example, we can use ODBC or OleDb data providers to access Microsoft access databases.

- **Managed Code**

The ADO .NET classes are managed classes. They take all the advantages of .NET CLR, such as language independency and automatic resource management. All .NET languages access the same API. So if you know how to use these classes in C#, you'll have no problem using them in VB.NET. Another big advantage is you don't have to worry about memory allocation and freeing it. The CLR will take care of it for you.

- **Deployment**

In real life, writing database application using ODBC, DAO, and other previous technologies and deploying on client machines was a big problem was somewhat taken care in ADO except that three are different versions of MDAC. Now we don't have to worry about that. Installing distributable .NET components will take care of it.

- **XML Support**

Today, XML is an industry standard and the most widely used method of sharing data among applications over the Internet. In ADO .NET data is cached and transferred in XML format. All components and applications can share this data and we can transfer data via different protocols such as HTTP.

- **Visual Data Components**

Visual Studio .NET offers ADO .NET components and data– bound controls to work in visual form. That means we can use these components as we use any windows controls. We drag and drop these components on windows and web forms set their properties and write events. It helps programmers to write less code and develop applications in no time. VS .NET also offers the data form wizard, which you can use to write full-fledged database applications without writing a single line of code. Using these components you can directly bind these components with data-bound controls by setting these control's properties at design-time.

- **Performance and Scalability**

Performance and scalability are two major factors when developing web-based applications and services. Transferring data one source to another is a costly affair over the Internet because of connection bandwidth limitations and rapidly increasing traffic. Using disconnected cached data in XML takes care of both of these problems.

- **Connections and Disconnected data**

With ADO .NET you use as few connections as possible and have more disconnected data. Both the ADO and ADO .NET models support disconnected data but ADO'S record set object wasn't actually designed to work with disconnected data. So there are performance problems with that. However, ADO.NET's dataset is specifically designed to work with disconnected data and you can treat a dataset as a local copy of a database. In ADO.NET, you store data in a dataset and close the make final changes to the data source. The ADO model is not flexible enough for XML users; ADO uses OLE-DB persistence provider to support XML.

- **DataReader versus DataSet**

The ADO.NET DataReader is used to retrieve read-only (cannot update data back to a datasource) and forward-only (cannot read backward/random) data from a database. You create a DataReader by calling Command.ExecuteReader after creating an instance of the Command object.

- **LINQ to DataSet**

LINQ to DataSet API provides queries capabilities on a cached DataSet object using LINQ queries. The LINQ queries are written in C#. Learn more here: LINQ to DataSet

- **LINQ to SQL**

LINQ to SQL API provides queries against relational databases without using a middle layer database library.

Accessing Data using ADO.NET

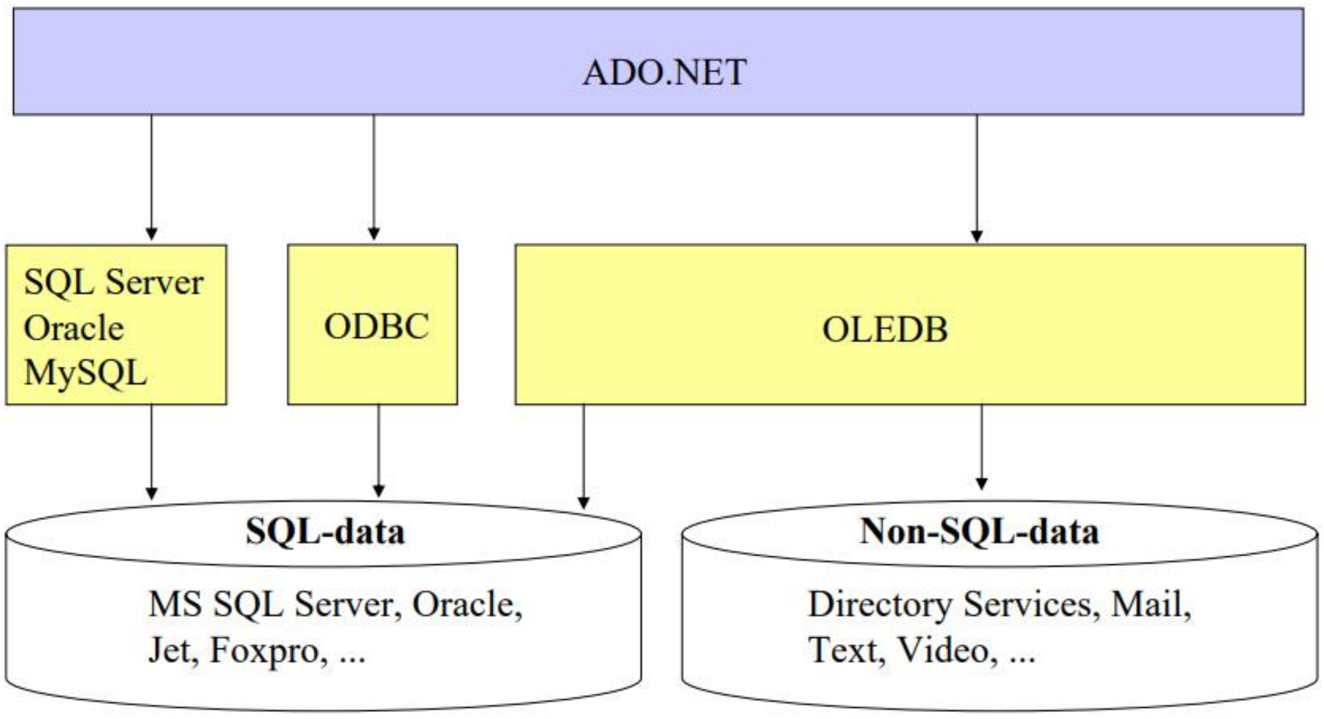
Many real-world applications need to interact with a database. The .NET Framework provides a rich set of objects to manage database interaction; these classes are collectively referred to as ADO.NET.

ADO.NET looks very similar to ADO, its predecessor. The key difference is that ADO.NET is disconnected data architecture. In a disconnected architecture, data is retrieved from a database and cached on your local machine. You manipulate the data on your local computer and connect to the database only when you wish to alter records or acquire new data.

History of Universal Data Access (Microsoft)

- ODBC
  - OLE DB
    - ADO (*Active Data Objects*)
      - ADO.NET

ADO	**ADO.NET**
connection-oriented	connection-oriented + connectionless
sequential access	main-memory representation with direct access
only one table supported	more than one table supported
COM-marshalling	XML-marshalling

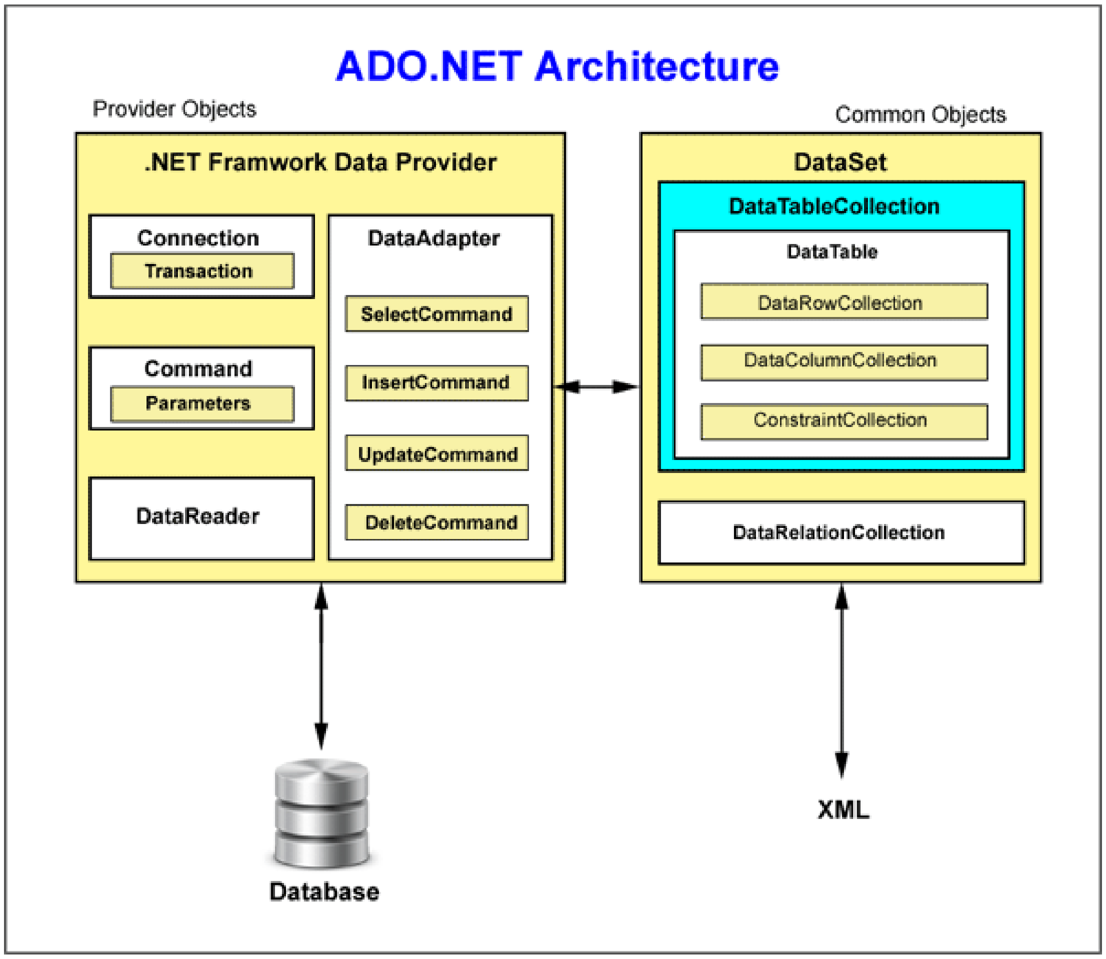

ADO.NET
SQL Server
Oracle
MySQL
ODBC
OLEDB
SQL-data
MS SQL Server, Oracle, Jet, Foxpro, ...
Non-SQL-data
Directory Services, Mail, Text, Video, ...

ActiveX Data Object.NET (ADO.NET) is a software **library** in the .NET framework consisting of software components **providing data access services**.ADO.NET is designed to enable developers to write managed code for obtaining **disconnected access to data sources**, which **can be relational or non-relational** (such as XML or application data). This feature of ADO.NET helps to create data-sharing, distributed applications.

The **architecture** of ADO.NET is **based on two primary elements**: **DataSet** and **.NET framework data provider**.

Dataset provides the following components:

- a complete set of data including related tables, constraints and their relationships
- functionality-like access to remote data from XML Web service
- manipulation of data dynamically
- data processing in a connectionless manner
- provision for hierarchical XML view of relational data
- usage of tools like XSLT and XPath Query to operate on the data



The .NET framework data provider includes the following components for data manipulation:

- **Connection**: This provides connectivity to the data source
- **Command**: This executes the database statements needed to retrieve data, modify data or execute stored procedures.
- **DataReader**: This retrieves data in forward only and read-only form.
- **DataAdapter**: This acts as bridge between dataset and data source to load the dataset and reconcile changes made in dataset back to the source.

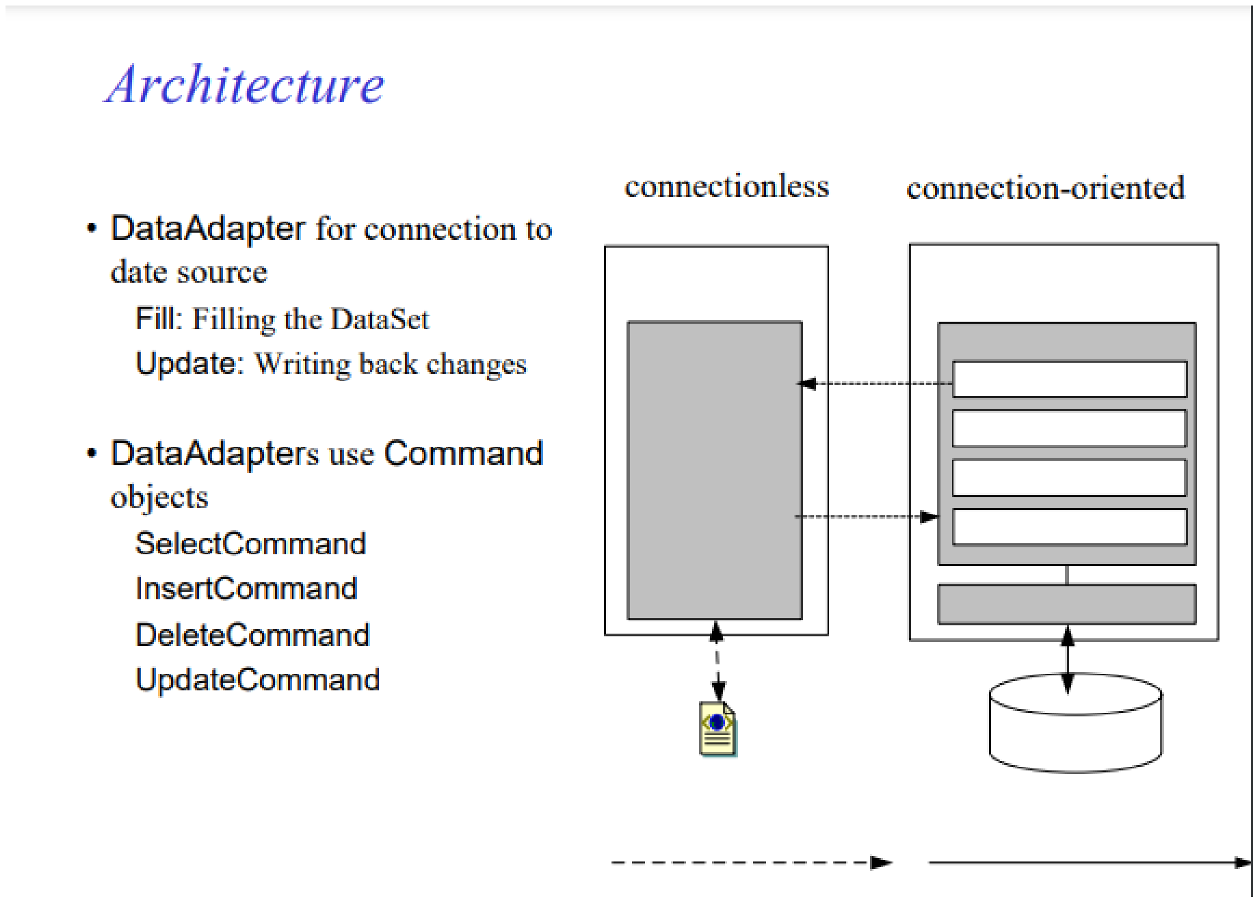
SqlDataAdapter.Fill()

It is used to retrieve the data from the database. Whenever we think that we need to get the data from any data source in ADO.NET and fill this data into DataTable or DataSet, then we need to use *objSqlDataAdapter.Fill(objDataSet)*;. When we pass multiple select statement query to get the data, then it executes and gets one by one data and pass it to corresponding tables.

So, first, we need to create a Stored Procedure to get the data from the database.

```
CREATE PROCEDURE GetEmployeeList
AS
BEGIN
```

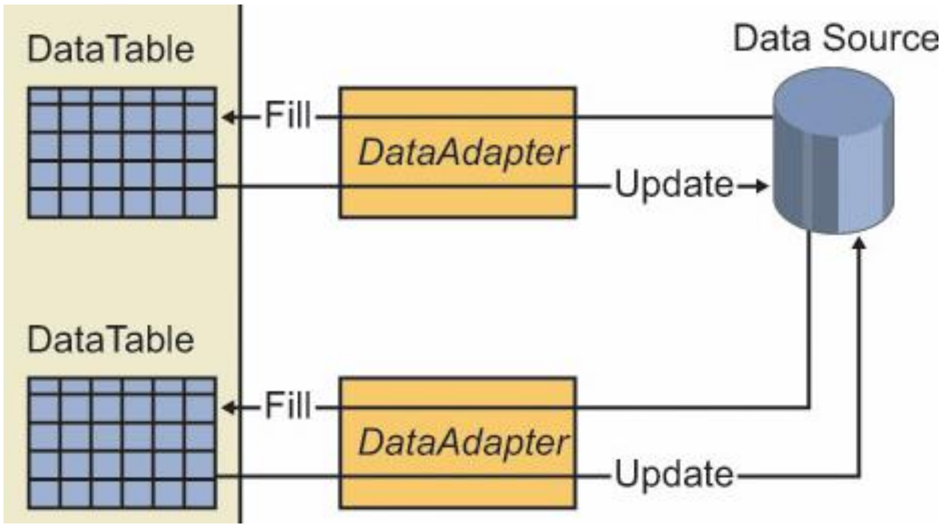
```
SELECT * FROM dbo.Employees
END
```



Architecture

- DataAdapter for connection to date source
  - Fill: Filling the DataSet
  - Update: Writing back changes
- DataAdapters use Command objects
  - SelectCommand
  - InsertCommand
  - DeleteCommand
  - UpdateCommand

connectionless
connection-oriented



Here we need to write our code on *Employee.aspx.cs* for getting the data.

Example

```
using System;

using System.Collections.Generic;
using System.Linq;

using System.Web;

using System.Web.UI;

using System.Web.UI.WebControls;

using System.Configuration;

using System.Data.SqlClient;

using System.Data;

namespace EmployeeDemo

{

    public partial class Emploee: System.Web.UI.Page

    {

        string connectionString = ConfigurationManager.ConnectionStrings["DefaultConnection"].ConnectionStrin
g;

        protected void Page_Load(object sender, EventArgs e)

        {

            if (!this.IsPostBack)

            {

                GetEmployeesList();

            }

        }

        public DataSet GetEmployeesList()

        {

            DataSet dsEmployee = new DataSet();

            using(SqlConnection con = new SqlConnection(connectionString))

            {

                SqlCommand objSqlCommand = new SqlCommand("GetEmployeeList", con);
                objSqlCommand.CommandType = CommandType.StoredProcedure;

                SqlDataAdapter objSqlDataAdapter = new SqlDataAdapter(objSqlCommand);

                try

                {

                    objSqlDataAdapter.Fill(dsEmployee);

                    dsEmployee.Tables[0].TableName = "Employees";

                    grvEmployee.DataSource = dsEmployee;

                    grvEmployee.DataBind();

                }

                catch (Exception ex)

                {

                    return dsEmployee;

                }

            }

            return dsEmployee;

        }       }     }
```

DataAdapters and DataReaders

We can use the ADO.NET **DataReader** to retrieve a read-only, forward-only stream of data from a database. Results are returned as the query executes and are stored in the network buffer on the client until we request them using the **Read** method of the **DataReader**. Using the **DataReader** can increase application performance both by retrieving data as soon as it is available, and (by default) storing only one row at a time in memory, reducing system overhead.

A **DataAdapter** is used to retrieve data from a data source and populate tables within a DataSet. The DataAdapter also resolves changes made to the DataSet back to the data source. The DataAdapter uses the Connection object of the .NET Framework data provider to connect to a data source, and it uses Command objects to retrieve data from and resolve changes to the data source.

DataAdapter will acts as a **Bridge** between **DataSet** and **database**. This **dataadapter** object is used to read the data from database and bind that data to dataset. Dataadapter is a disconnected oriented architecture.

Distributed Application in C# (.NET Remoting)

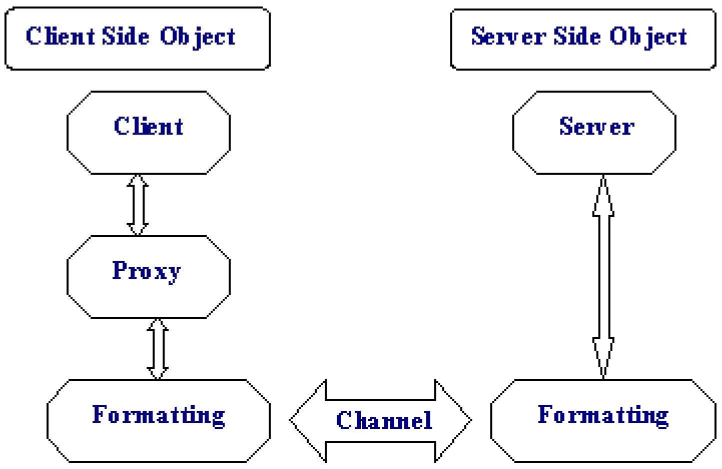
Introduction

.NET Remoting is a mechanism for communicating between objects which are not in the same process. It is a generic system for different applications to communicate with one another. The applications can be located on the same computer, different computers on the same network or on computers across separate networks.

Microsoft .NET Remoting provides a framework that allows objects to interact with each other across application domains. Remoting was designed in such a way that it hides the most difficult aspects like managing connections, marshaling data, and reading and writing XML and SOAP. The framework provides several services, including object activation and object lifetime support, as well as communication channels which are responsible for transporting messages to and from remote applications.

Remote Objects

Any object outside the application domain of the caller should be considered remote, even if the objects are executing on the same machine. Inside the application domain, all objects are passed by reference while primitive data types are passed by value. Since local object references are only valid inside the application domain where they are created, they cannot be passed to or returned from remote method calls in that form. All local objects that have to cross the application domain boundary have to be passed by value and should be marked with the **[serializable]** custom attribute, or they have to implement the **ISerializable** interface. When the object is passed as a parameter, the framework serializes the object and transports it to the destination application domain, where the object will be reconstructed. Local objects that cannot be serialized cannot be passed to a different application domain and are therefore nonremotable.



Any object can be changed into a remote object by deriving it from MarshalByRefObject(Enables access to objects across application domain boundaries in applications that support remoting). When a client activates a remote object, it receives a proxy to the remote object. All operations on this proxy are appropriately indirected to enable the Remoting infrastructure to intercept and forward the calls appropriately.

Using the .NET Framework to Develop Distributed Applications

The .NET Framework provides various mechanisms to support distributed application development. Most of this functionality is present in the following three namespaces of the Framework Class Library (FCL):

- The **System.Net** Namespace—This namespace includes classes to create standalone listeners and custom protocol handlers to start from scratch and create your own framework for developing a distributed application. Working with the System.Net namespace directly requires a good understanding of network programming.
- The **System.Runtime.Remoting** Namespace—This namespace includes the classes that constitute the .NET remoting framework. The .NET remoting framework allows communication between objects living in different application domains, whether or not they are on the same computer. Remoting provides an abstraction over the complex network programming and exposes a simple mechanism for inter-application domain communication. The key objectives of .NET remoting are flexibility and extensibility.
- The **System.Web.Services** Namespace—This namespace includes the classes that constitutes the ASP.NET Web services framework. ASP.NET Web services allow objects living in different application domains to exchange messages through standard protocols such as HTTP and SOAP. ASP.NET Web services, when compared to remoting, provide a much higher level of abstraction and simplicity. The key objectives of ASP.NET Web services are the ease of use and interoperability with other systems.

Both **.NET remoting** and **ASP.NET Web services** provide a complete framework for designing distributed applications. Most programmers will use either .NET remoting or ASP.NET Web services rather than build a distributed programming framework from scratch with the System.Net namespace classes.

The functionality offered by .NET remoting and ASP.NET Web services appears very similar. In fact, ASP.NET Web services are actually built on the .NET remoting infrastructure. It is also possible to use .NET remoting to design Web services. Given the amount of similarity, how do you choose one over the other in your project? Simply put, the decision depends on the type of application you want to create.

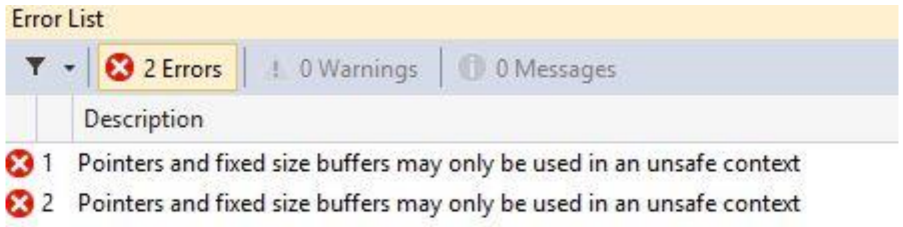
- .NET Remoting when both the end points (client and server) of a distributed application are in your control. This might be a case when an application has been designed for use within a corporate network.
- ASP.NET Web services when one end point of a distributed application is not in your control. This might be a case when your application is interoperating with your business partner's application.

Unsafe Code in C#

Before we start any discussion of Unsafe Code let us see an example.

```
static void Main(string[] args)
 {
    int ab = 32;
    int* p = &ab;
    Console.WriteLine("value of ab is {0}", *p);
    Console.ReadLine();
 }
```

When I compile this code I encounter the following errors:



```
1  Pointers and fixed size buffers may only be used in an unsafe context
2  Pointers and fixed size buffers may only be used in an unsafe context
```

Why these errors occur in the program

These errors occur because we executed our program in **Safe mode** but we must execute our program in **Unsafe** mode if we are using pointers.

Now a question is, **“What is meant by Safe and Unsafe Code?”**

First we understand what Managed and Unmanaged Code are.

Managed Code

Managed code is that code that executes under the supervision of the CLR. The CLR is responsible for various housekeeping tasks, like:

- Managing memory for the objects
- Performing type verification
- Doing garbage collection

Unmanaged Code

On the other hand, unmanaged code is code that executes outside the context of the CLR. The best example of this is our traditional Win32 DLLs like kernel32.dll and user32.dll.

In unmanaged code a programmer is responsible for:

Calling the memory allocation function

Making sure that the casting is done right

Making sure that the memory is released when the work is done

Now to understand what UnsafeMode is.

Definition of Unsafe Mode

Unsafe is a C# programming language keyword to denote a section of code that is not managed by the Common Language Runtime (CLR) of the .NET Framework, or unmanaged code. Unsafe is used in the declaration of a type or member or to specify a block code. When used to specify a method, the context of the entire method is unsafe.

To maintain type safety and security, C# does not support pointer arithmetic, by default. However, using the unsafe keyword, you can define an unsafe context in which pointers can be used.

Unsafe code can create issues with stability and security, due to its inherent complex syntax and potential for memory related errors, such as stack overflow, accessing and overwriting system memory. Extra developer care is paramount for averting potential errors or security risks.

Example

The following is an example of Unsafe Mode:

```
unsafe static void Main()
{
  fixed (char* value = "safe")
  {
    char* ptr = value;
    while (*ptr != '\0')
    {
      Console.WriteLine(*ptr);
      ++ptr;
    }
  }
}
```

Output

s

a

f

e

Writing unsafe code requires the use of the two special keywords **unsafe** and **fixed**. Now to understand the meaning of both of those keywords.

Unsafe: The Unsafe keyword tells the compiler that this code will run in unsafe mode.

Fixed: We use fixed buffers inside an unsafe context. With a fixed buffer, we can write and read raw memory without the managed overhead. Enter the fixed keyword. When used for a block of statements, it tells the CLR that the object in question cannot be relocated.

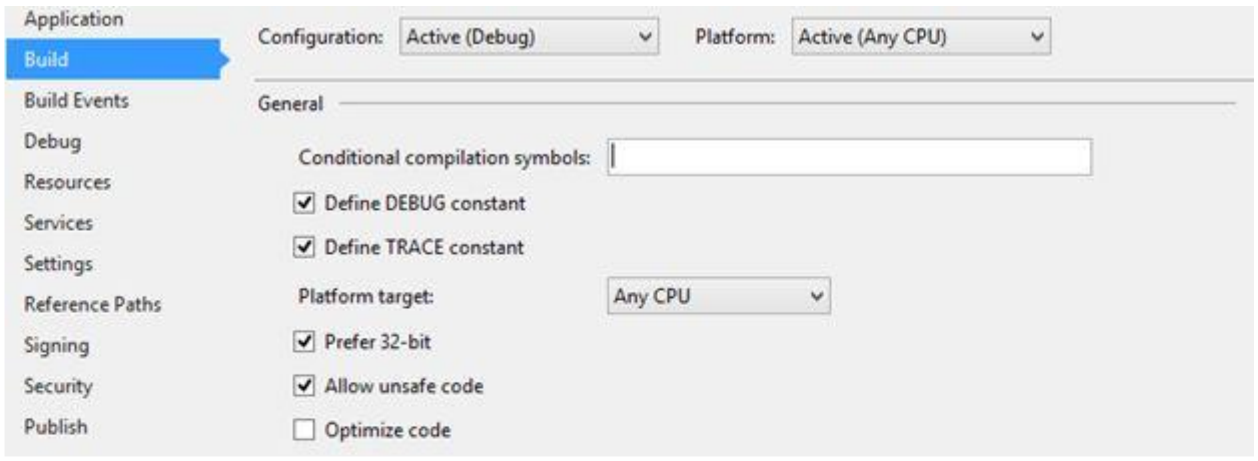
How to run a program in unsafe mode

First go to the View tab.

Select the Solution Explorer option.

Expand the Solution Explorer a double-click on the Property option.

Now select the option of “Allow unsafe code” and mark it Check.



Configuration: Active (Debug)
Platform: Active (Any CPU)

Conditional compilation symbols:

✓ Define DEBUG constant

✓ Define TRACE constant

Platform target: Any CPU

✓ Prefer 32-bit

✓ Allow unsafe code

☐ Optimize code

Let us see some examples and understand how to use it .

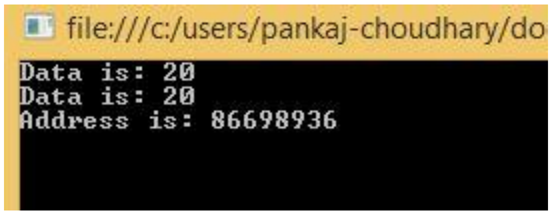
Example 1

Retrieving the Data Value Using a Pointer

You can retrieve the data stored at the location referenced by the pointer variable, using the ToString()method. The following example shows this:

```
static unsafe void Main(string[] args)
 {
   int var = 20;

   int* p = &var;

   Console.WriteLine("Data is: {0} ", var);

   Console.WriteLine("Data is: {0} ", p->ToString());

   Console.WriteLine("Address is: {0} ", (int)p);

   Console.ReadKey();

 }
```

Output



file:///c:/users/pankaj-choudhary/do

```
Data is: 20
Data is: 20
Address is: 86698936
```

Example 2

Instead using the unsafe keyword for the Main function we can use it for a specific block of code, such as:

```
static void Main(string[] args)

{

  Unsafe

  {

     int var = 20;

     int* p = &var;

     Console.WriteLine("Data is: {0} ", var);

     Console.WriteLine("Data is: {0} ", p->ToString());

     Console.WriteLine("Address is: {0} ", (int)p);

     Console.ReadKey();

  }
}
```

Output

The output will remain the same as above.

Example 3

In this program a function with the unsafe modifier is called from a normal function. This program shows that a managed code can call unmanaged functions.

```
class Program
{
  static unsafe void Main(string[] args)
  {
     Unsafe();
     Console.ReadKey();
  }
  public static unsafe void Unsafe()
  {
     int var = 20;
     int* p = &var;
     Console.WriteLine("Data is: {0} ", var);
     Console.WriteLine("Data is: {0} ", p->ToString());
     Console.WriteLine("Address is: {0} ", (int)p);
  }
}
```

Output

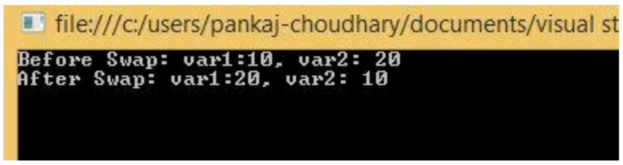
The output will the remain same as above.

Example 4

You can pass a pointer variable to a method as a parameter.

```
static unsafe void Main(string[] args)
{
    Program p = new Program();
    int var1 = 10;
    int var2 = 20;
    int* x = &var1;
    int* y = &var2;
    Console.WriteLine("Before Swap: var1:{0}, var2: {1}", var1, var2);
    p.swap(x, y);
    Console.WriteLine("After Swap: var1:{0}, var2: {1}", var1, var2);
    Console.ReadKey();
}

public unsafe void swap(int* p, int* q)
{
    int temp = *p;
    *p = *q;
    *q = temp;
}
```

Output



```
Before Swap: var1:10, var2: 20
After Swap: var1:20, var2: 10
```

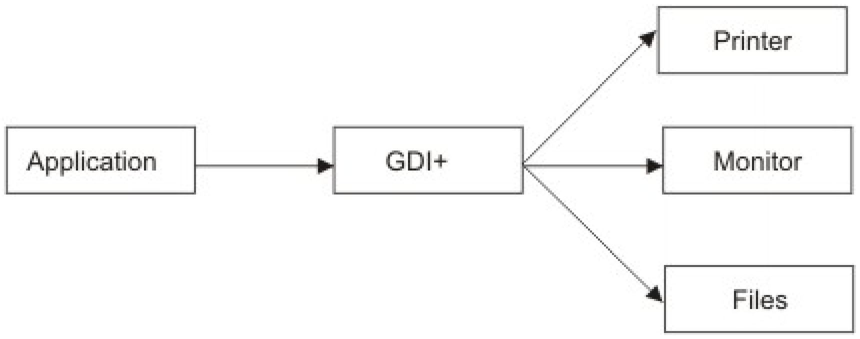
Graphical Device Interface (GDI) In C#

The **Graphics Device Interface** (**GDI**) is responsible for representing graphical objects and transmitting them to output devices such as monitors and printers.

GDI is responsible for tasks such as drawing lines and curves, rendering fonts and handling palettes.

GDI+ is the gateway to interact with graphics device interfaces in the .NET Framework. If you're going to write .NET applications that interact with graphics devices such as monitors, printers, or files, you will have to use GDI+.

GDI+ is a library that provides an interface that allows programmers to write Windows and Web graphics applications that interact with graphical devices such as printers, monitors, or files.



Now let's see how GDI+ works.

Suppose your program draws a line. A line is displayed as a set of pixels drawn in sequence from the starting location to the ending location. To draw a line on a monitor, the monitor needs to know where to draw the pixels. Instead of telling the monitor to draw pixels your program calls the DrawLine method of GDI+, and GDI+ draws the line from point A to point B. GDI+ reads the point A and point B locations, converts them to a sequence of pixels, and tells the monitor to display the sequence of pixels.

GDI+ allows you to write device-independent managed applications and is designed to provide high performance, ease of use, and multilingual support.

All GDI+ classes reside in the following namespaces:

- System.Drawing
- System.Text
- System.Printing
- System.Internal
- System.Imaging

• System.Drawing2D

• System.Design

The Graphics Class

The Graphics class encapsulates GDI+ drawing surfaces. Before drawing any object (for example circle or rectangle) we have to create a surface using Graphics class. Generally we use Paint event of a Form to get the reference of the graphics. Another way is to override OnPaint method.

Here is how you get a reference of the Graphics object:

```
private void form1_Paint(object sender, PaintEventArgs e)
{
        Graphics g = e.Graphics;
}
                        OR
protected override void OnPaint(PaintEventArgs e)
{
        Graphics g = e.Graphics;
}
```

Once you have the Graphics reference, you can call any of this class's members to draw various objects. Here are some of Graphics class's methods:

Methods	Description
DrawArc	Draws an arc from the specified ellipse.
DrawBezier	Draws a cubic bezier curve.
DrawBeziers	Draws a series of cubic Bezier curves.
DrawClosedCurve	Draws a closed curve defined by an array of points.

DrawCurve	Draws a curve defined by an array of points.
DrawEllipse	Draws an ellipse.
DrawImage	Draws an image.
DrawLine	Draws a line.
DrawPath	Draws the lines and curves defined by a GraphicsPath.
DrawPie	Draws the outline of a pie section.
DrawPolygon	Draws the outline of a polygon.
DrawRectangle	Draws the outline of a rectangle.
DrawString	Draws a string.
FillEllipse	Fills the interior of an ellipse defined by a bounding rectangle.
FillPath	Fills the interior of a path.
FillPie	Fills the interior of a pie section.
FillPolygon	Fills the interior of a polygon defined by an array of points.
FillRectangle	Fills the interior of a rectangle with a Brush.
FillRectangles	Fills the interiors of a series of rectangles with a Brush.
FillRegion	Fills the interior of a Region.