UNIT 1
INTRODUCTION TO DOT NET AND C#

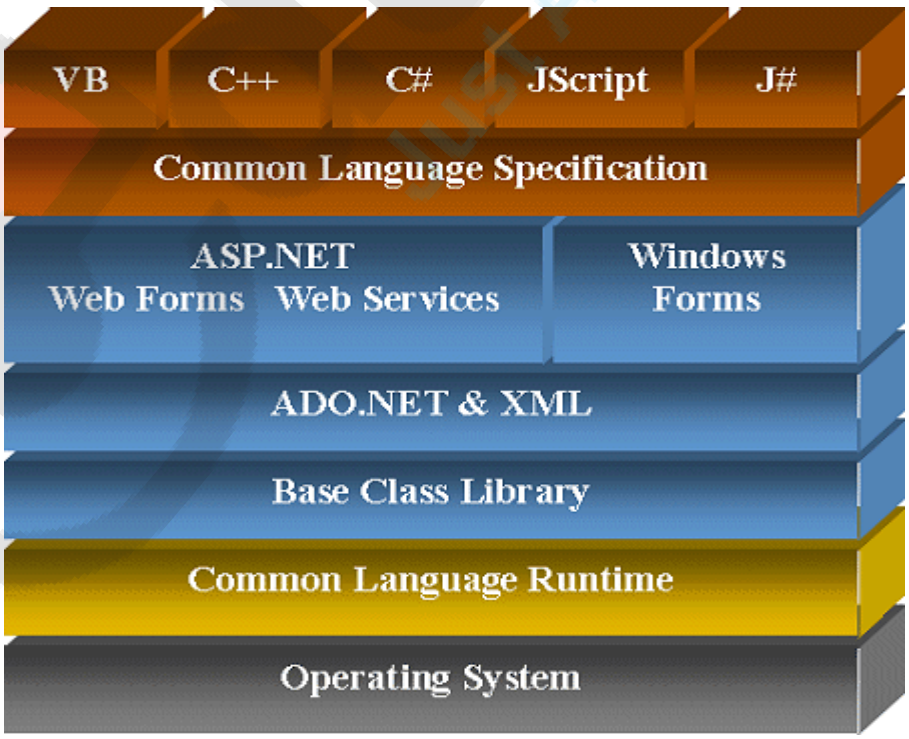
1) INTRODUCTION TO DOT NET FRAMEWORK

The .NET Framework is a technology that supports building and running the next generation of apps and XML Web services. The .NET Framework is designed to fulfill the following objectives:

- To provide a consistent object-oriented programming environment whether object code is stored and executed locally, executed locally but Internet-distributed, or executed remotely.
- To provide a code-execution environment that minimizes software deployment and versioning conflicts.
- To provide a code-execution environment that promotes safe execution of code, including code created by an unknown or semi-trusted third party.
- To provide a code-execution environment that eliminates the performance problems of scripted or interpreted environments.
- To make the developer experience consistent across widely varying types of apps, such as Windows-based apps and Web-based apps.
- To build all communication on industry standards to ensure that code based on the .NET Framework integrates with any other code.

1.1.1)Architecture of Dot NET Framework

Ans:



Net Framework is a platform that provides tools and technologies to develop Windows, Web and Enterprise applications. It mainly contains two components,

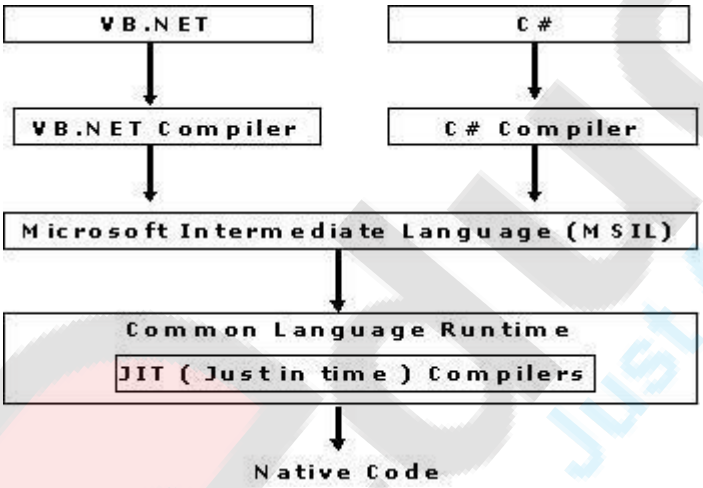
1. Common Language Runtime (CLR)
2. .Net Framework Class Library.

1. Common Language Runtime (CLR)

.Net Framework provides runtime environment called **Common Language Runtime** (CLR).It provides an environment to run all the .Net Programs. The code which runs under the CLR is called as **Managed Code**. Programmers need not to worry on managing the memory if the programs are running under the CLR as it provides memory management and thread management.

Programmatically, when our program needs memory, CLR allocates the memory for scope and de-allocates the memory if the scope is completed.

Language Compilers (e.g. C#, VB.Net, J#) will convert the Code/Program to **Microsoft Intermediate Language** (MSIL) intern this will be converted to **Native Code** by CLR. See the below Fig.



2. .Net Framework Class Library (FCL)

This is also called as Base Class Library and it is common for all types of applications i.e. the way you access the Library Classes and Methods in VB.NET will be the same in C#, and it is common for all other languages in .NET.

The following are different types of applications that can make use of .net class library.

1. Windows Application.
2. Console Application

3. Web Application.
4. XML Web Services.
5. Windows Services.

In short, developers just need to import the BCL in their language code and use its predefined methods and properties to implement common and complex functions like reading and writing to file, graphic rendering, database interaction, and XML document manipulation.

Below are the few more concepts that we need to know and understand as part of this .Net framework.

3. Common Type System (CTS)

It describes set of data types that can be used in different .Net languages in common. (i.e), CTS ensures that objects written in different .Net languages can interact with each other.

For Communicating between programs written in any .NET complaint language, the types have to be compatible on the basic level.

The common type system supports two general categories of types:

Value types:

Value types directly contain their data, and instances of value types are either allocated on the stack or allocated inline in a structure. Value types can be built-in (implemented by the runtime), user-defined, or enumerations.

Reference types:

Reference types store a reference to the value's memory address, and are allocated on the heap. Reference types can be self-describing types, pointer types, or interface types. The type of a reference type can be determined from values of self-describing types. Self-describing types are further split into arrays and class types. The class types are user-defined classes, boxed value types, and delegates.

Microsoft ADO.net and XML
Access relational databases
Disconnected data model
Work with XML
Microsoft Userinterface (Asp.Net & Windows Forms,wpf)
Create application's front-end
Web-based user interface, Windows GUI,
Web services
Programming Languages
Use your favorite language
Microsoft .net framework support multiple Programming Languages Support.
C#,V.b.net,J#,F#,c++ other...

Microsoft.Net Framework Components

Following are the major components of .NET Framework:

Common Language Specification (CLS)

.Net Framework Languages

.Net Framework Base Class Library (BCL - FCL)

Common Language Runtime (CLR)

4. Common Language Specification (CLS)

It is a sub set of CTS and it specifies a set of rules that needs to be adhered or satisfied by all language compilers targeting CLR. It helps in cross language inheritance and cross language debugging.

Common language specification Rules:

It describes the minimal and complete set of features to produce code that can be hosted by CLR. It ensures that products of compilers will work properly in .NET environment.

Sample Rules:

1. Representation of text strings
2. Internal representation of enumerations
3. Definition of static members and this is a subset of the CTS which all .NET languages are expected to support.
4. Microsoft has defined CLS which are nothing but guidelines that language to follow so that it can communicate with other .NET languages in a seamless manner.

1.1.2) CLR

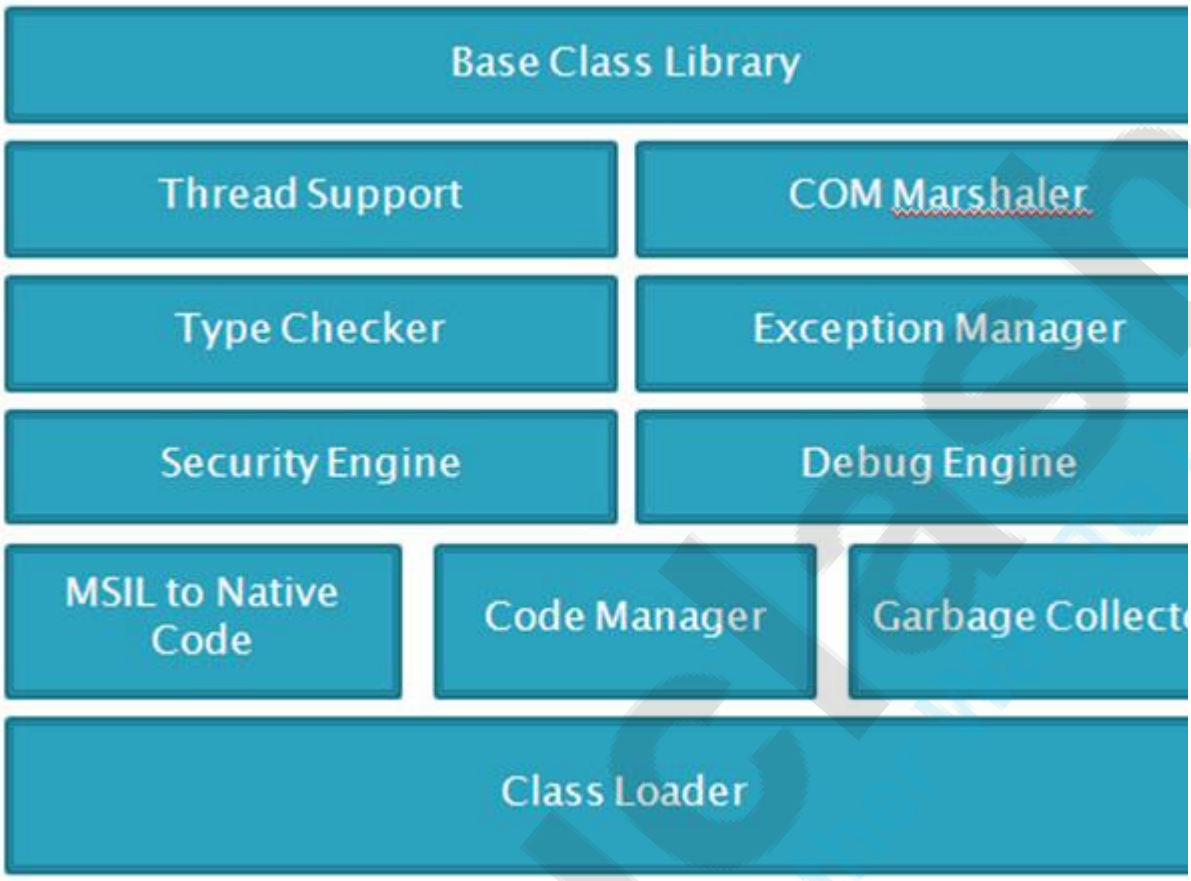
About CLR

CLR (Common Language Runtime) is a heart of Dot Net Framework.

It is a core runtime environment of .NET Framework for executing applications. The main function of Common Language Runtime (CLR) is to convert the Managed Code into native code and then execute the Program. It acts as a layer between Operating Systems and the applications written in .Net languages.

CLR handles the execution of code and provides useful services for the implementation of the program. In addition to executing code, CLR provides services such as memory management, thread management, security management, code verification, compilation, and other system services.

Role of CLR in DOT.NET Framework



Base Class Libraries: It provides class libraries supports to an application when needed.
MSIL Code to Native Code: The Common Language Runtime is the engine that compiles the source code in to an intermediate language. This intermediate language is called the Microsoft Intermediate Language.

During the execution of the program this MSIL is converted to the native code or the machine code. This conversion is possible through the Just-In-Time compiler. During compilation the end result is a Portable Executable file (PE).

Thread Support: Threads are managed under the Common Language Runtime. Threading means parallel code execution. Threads are basically light weight processes responsible for multi-tasking within a single

application.

COM Marshaler: It allows the communication between the application and COM objects.

Code Manager: CLR manages code. When we compile a .NET application you don't generate code that can actually execute on your machine. You actually generate Microsoft Intermediate Language (MSIL or just IL). All .NET code is IL code. IL code is also called Managed Code, because the .NET Common Language Runtime manages it.

Debug Engine: CLR allows us to perform debugging an application during runtime.

Common Language Specification (CLS) :
CLS use to communicate Objects written in different .Net languages.
Common Language Specification (CLS) defines the rules and standards to which languages must adhere to in order to be compatible with other .NET languages. This enables C# developers to inherit from classes defined in VB.NET or other .NET compatible languages.

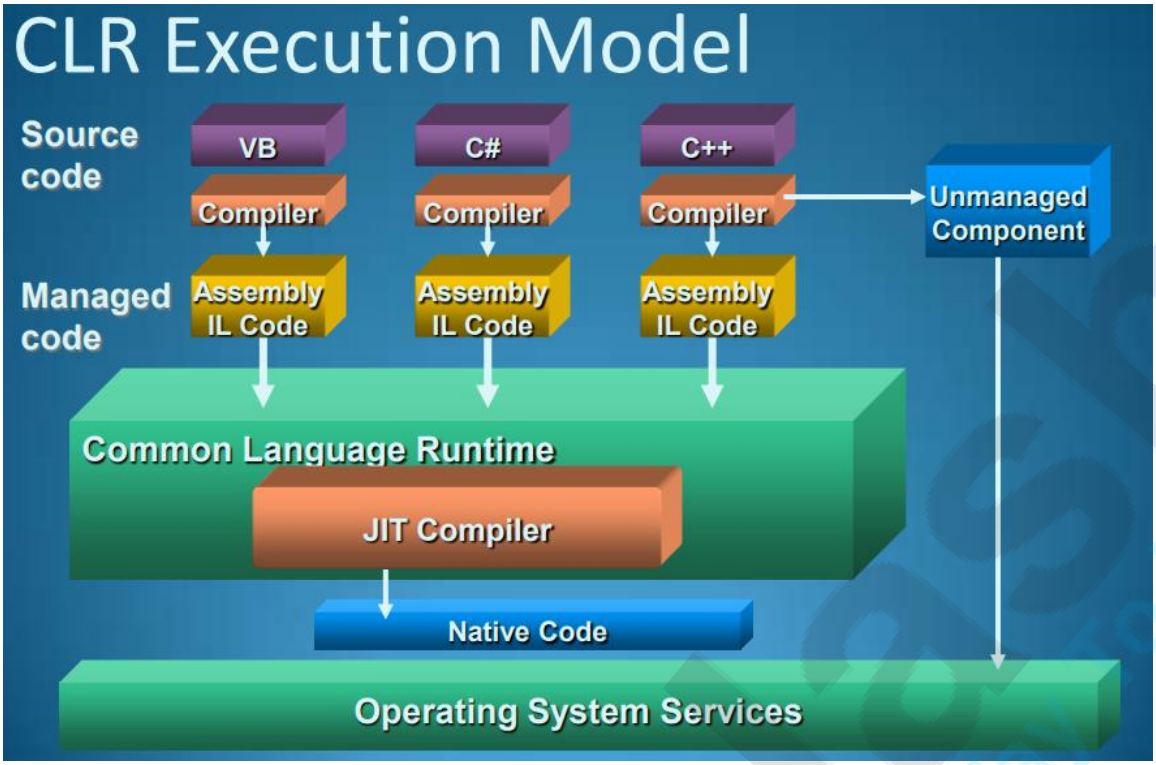
CTS (Common Type System):
It specifies data types which are created in two different languages get compiled in to base common data type system.

Type Checker
Type checker will verify types used in the application with CTS or CLS standards supported by CLR, this provides type safety.

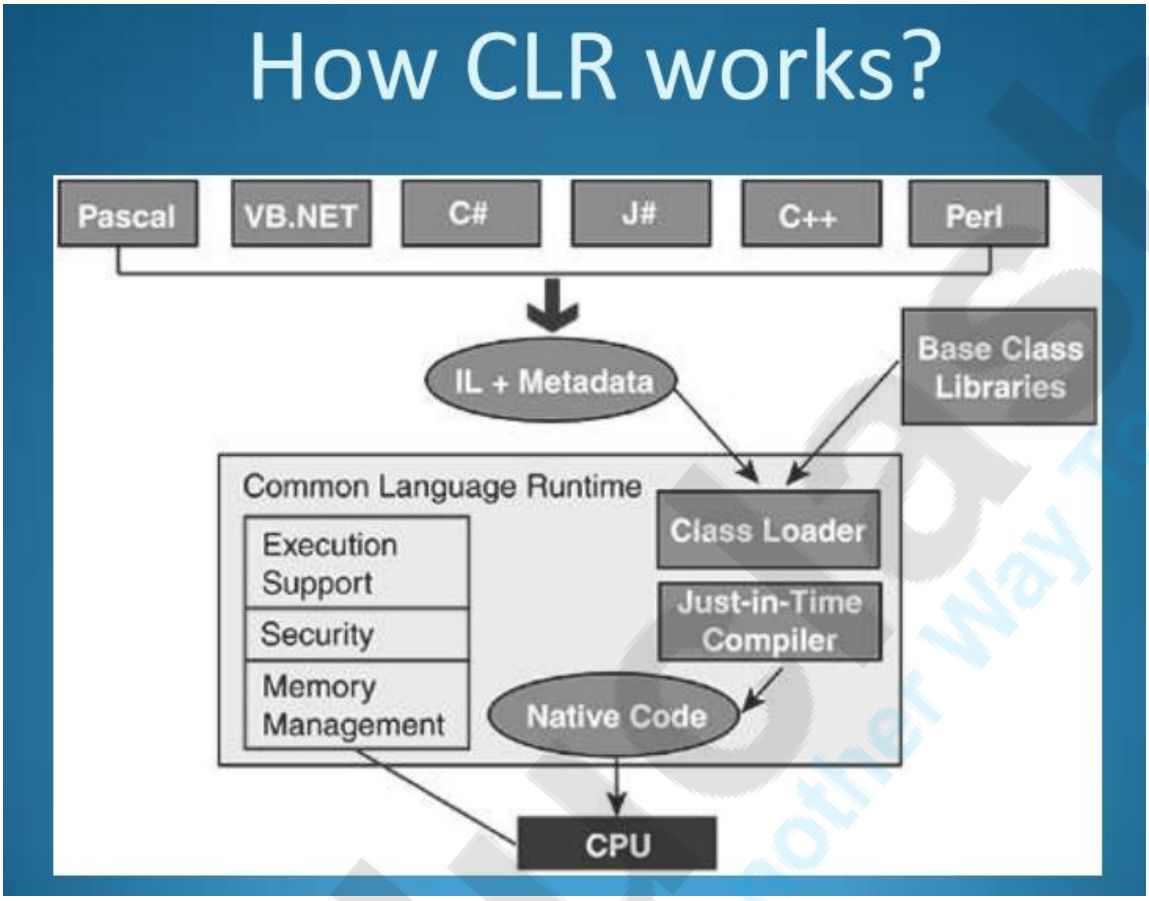
Exception Manager:
Exception Manager will handle exceptions thrown by application by while executing Try-catch block provided by an exception.
In "Try" block used where a part of code expects an error
In "Catch" block throws an exception caught from "try" block, if there is no catch block, it will terminate application.

Security Engine: It enforces security permissions at code level security, folder level security, and machine level security using Dot Net Framework setting and tools provided by Dot Net.

Garbage_Collector
Garbage Collector handles automatic memory management and it will release memory of unused objects in an application, this provides automatic memory management.


CLR Execution Model
Source code
VB
C#
C++
Compiler
Compiler
Compiler
Unmanaged Component
Managed code
Assembly IL Code
Assembly IL Code
Assembly IL Code
Common Language Runtime
JIT Compiler
Native Code
Operating System Services

Working of CLR:



How does CLR work?

Within the domain of CLR are executables (consisting of code, data, and metadata), assemblies (consisting of a manifest and zero or more modules), and the Common Type System (CTS) convention set. When programmers write code in their favorite languages, that code is translated into IL prior to being compiled into a portable executable (PE).

Executables

The main difference between a Windows PE and a .NET PE is that the Windows PE is executed by the operating system, but .NET PEs are turned over to the .NET Framework's CLR. Recognition of a PE as being .NET or Windows occurs because of the Common Object File Format (COFF) used by Windows operating systems. The COFF specifies two parts of any file: the file data itself and a bunch of header data describing the contents of the data portion. **Note:** To allow all Microsoft platforms to handle COFF modifications that enable .NET PEs, Microsoft has released new loaders for all of .NET's supported systems (98, 2000, and Me).

Metadata

Metadata is information about a PE. In COM, metadata is communicated through no standardized type libraries. In .NET, this data is contained in the header portion of a COFF-compliant PE and follows certain guidelines; it contains information such as the assembly's name, version, language (spoken, not computer-a.k.a., "culture"), what external types are referenced, what internal types are exposed, methods, properties, classes, and much more.

The CLR uses metadata for a number of specific purposes. Security is managed through a public key in the PE's header. Information about classes, modules, and so forth allows the CLR to know in advance what structures are necessary.

The class loader component of the CLR uses metadata to locate specific classes within assemblies, either locally or across networks. Just-in-time (JIT) compilers use the metadata to turn IL into executable code.

Other programs take advantage of metadata as well. A common example is placing a Microsoft Word document on a Windows 2000 desktop. If the document file has completed comments, author, title, or other Properties metadata, the text is displayed as a tool tip when a user hovers the mouse over the document on the desktop. You can use the Ildasm.exe utility to view the metadata in a PE. Literally, this tool is an IL disassembler.

Interoperability

Different types of files are handled by two different virtual systems in Windows and .NET. If a Windows executable is to interoperate with the .NET Framework, it interfaces with a COM wrapper for the desired .NET functionality, instead of accessing the functionality directly. Similarly, if a .NET application utilizes Windows (COM) objects, it needs a set of classes that expose the functionality, instead of accessing it directly. This communication between .NET and Windows is called "interoperability". Included in the .NET SDK are two sets of two tools each. One set is for .NET-to-COM operations, and the other is for COM-to-.NET operations.

The first pair of tools consists of Regasm.exe and Tlbexp.exe. Regasm.exe registers a .NET Assembly in the Windows registry. Once this is done, the assembly is exposed as a COM object to the Windows OS. Developers who wish to access .NET Assemblies as COM objects in their own applications can use the Tlbexp.exe utility to export a Type Library TLB file to be referenced by their applications. The properties and methods of .NET Assembly are available, just as with any other COM object.

The second pair of tools consists of TlbImp.exe and Xsd.exe. TlbImp.exe is run against a TLB file to create a .NET Assembly in the form of a dynamic-link library (DLL) file.

Custom types in .NET are described through XML Schema Definitions (XSDs). When you run the Xsd.exe utility against an existing XSD file with the "/c" switch,

the schema is converted to a C# class definition. As a sidenote, Xsd.exe also generates an XSD file from a .NET Assembly (using metadata in the COFF headers) when run against a .NET PE.

Assemblies

A .NET Assembly contains all the metadata about the modules, types, and other elements it contains in the form of a "manifest". The CLR loves assemblies because differing programming languages are just perfect for creating certain kinds of applications. For example, COBOL stands for Common Business-Oriented Language because it's tailor-made for creating business apps. However, it's not much good for creating drafting programs. Regardless of what language you used to create your modules, they can all work together within one Portable Executable Assembly.

Hierarchy

There's a hierarchy to the structure of .NET code. That hierarchy is "Assembly -> Module -> Type -> Method". Let's say you want your computer to calculate your mortgage. You'd create a method called "fnCalculateMortgage" that returns an amortization table.

You *could* create a whole stand-alone application for this purpose, or you could make it one method of a larger collection of functions (called a "type") that you name "libFinancialFunctions". This library of financial functions could include real-time functions to transfer funds between accounts and other financial functions. The type, in turn, is contained within a module of IL code called "MyAccounting" that contains all the financial and accounting functions your business uses. Finally, the MyAccounting module could be one of several in the final assembly, called "MyMIS", which contains all your business management and operations functions.

Intermediate language

Assemblies are made up of IL code modules and the metadata that describes them. Although programs may be compiled via an IDE or the command line, in fact, they are simply translated into IL, *not* machine code. The actual machine code is not generated until the function that requires it is called. This is the just-in-time, or JIT, compilation feature of .NET.

JIT compilation happens at runtime for a variety of reasons, one of the most ambitious being Microsoft's desire for cross-platform .NET adoption. If a CLR is built for another operating system (UNIX or Mac), the same assemblies will run in addition to the Microsoft platforms. The hope is that .NET assemblies are write-once-run-anywhere applications. This is a .NET feature that works behind-the-scenes, ensuring that developers are not limited to writing applications for one single line of products. No one has demonstrated whether or not this promise will ever truly materialize.

CTS/CLS

The MSIL Instruction Set Specification is included with the .NET SDK, along with the IL Assembly Language Programmers Reference. If a developer wants to write custom .NET programming languages, these are the necessary specifications and syntax. The CTS and CLS define the types and syntaxes that every .NET language needs to embrace. An application may not expose these features, but it must consider them when communicating through IL.

Just In Time Compilers (JITers):

When our IL compiled code needs to be executed, CLR invokes JIT compilers which compile the IL code to native executable code (.exe or .dll) for the specific machine and OS. JITers in many ways are different from traditional compilers as they, as their name suggests, compile the IL to native code only when desired e.g., when a function is called, IL of function's body is converted to native code; just in time of need. So, the part of code that is not used by particular run is not converted to native code. If some IL code is converted to native code then the next time when its needed to be used, the CLR uses the same copy without re-compiling.

Features of CLR:

CLR Features

This section describes, more specifically, what the CLR does for you. Table 1.1 summarizes CLR features with descriptions and chapter references (if applicable) in this book where you can find more detailed information.

Table 1.1. CLR Features

Feature	Description
.NET Framework Class Library support	Contains built-in types and libraries to manage assemblies, memory, security, threading, and other runtime system support
Debugging	Facilities for making it easier to debug code. (Chapter 7)
Exception management	Allows you to write code to create and handle exceptions. (Chapter 11)
Execution management	Manages the execution of code
Garbage collection	Automatic memory management and garbage collection (Chapter 15)
Interop	Backward-compatibility with COM and Win32 code. (Chapter 41)
Just-In-Time (JIT) compilation	An efficiency feature for ensuring that the CLR only compiles code just before it executes
Security	Traditional role-based security support, in addition to Code Access Security (CAS) (Chapter 44)
Thread management	Allows you to run multiple threads of execution (Chapter 39)
Type loading	Finds and loads assemblies and types
Type safety	Ensures references match compatible types, which is very useful for reliable and secure code (Chapter 4)

1.1.3) COMMON TYPE SYSTEM [CTS]: -

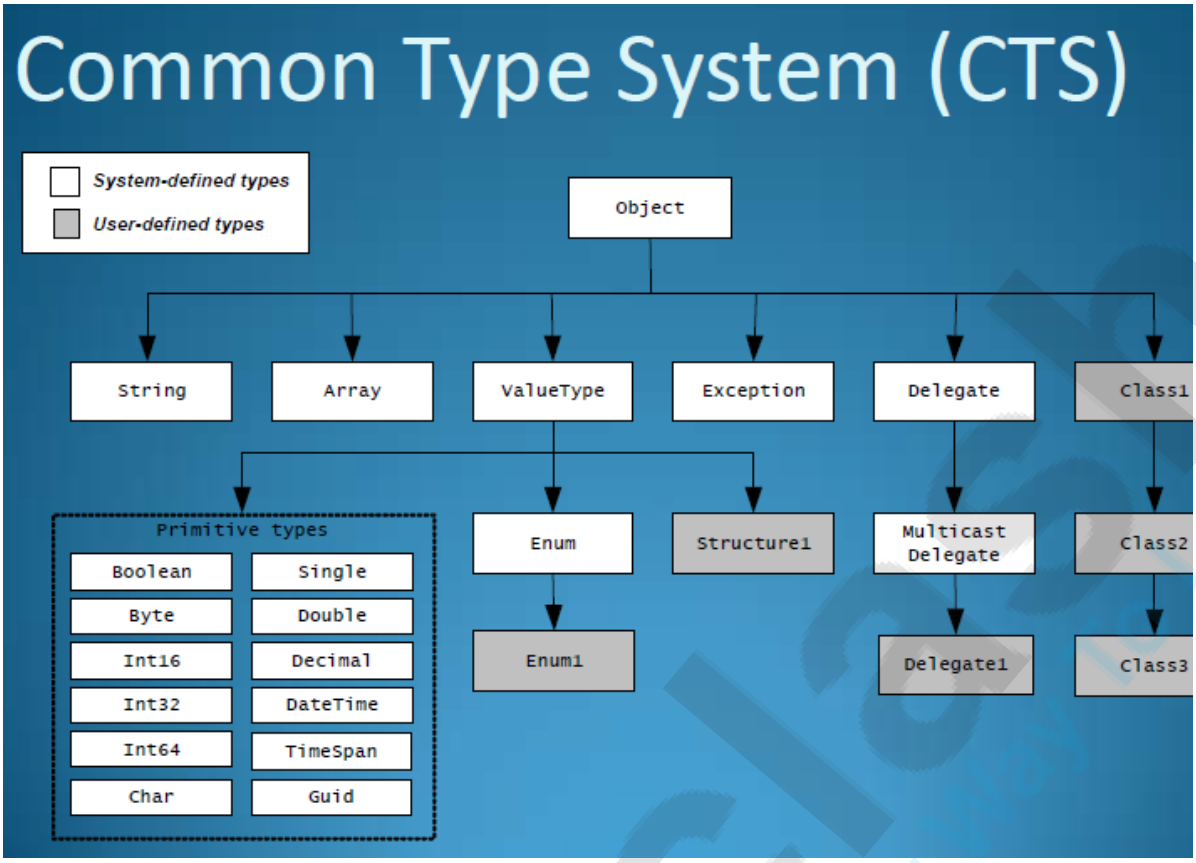
The Common Type System (CTS) is a standard for defining and using data types in the .NETframework. CTS defines a collection of data types, which are used and managed by the run time to facilitate cross-language integration.

CTS provides the types in the .NET Framework with which .NET applications, components and controls are built in different programming languages so information is shared easily. In contrast to low-level languages like C and C++ where classes/structs have to be used for defining types often used (like date or time), CTS provides a rich hierarchy of such types without the need for any inclusion of header files or libraries in the code.

CTS is a specification created by Microsoft and included in the European Computer Manufacturer‘s Association standard. It also forms the standard for implementing the .NET framework.

CTS is designed as a singly rooted object hierarchy with System.Object as the base type from which all other types are derived. CTS supports two different kinds of types:

1. Value Types: Contain the values that need to be stored directly on the stack or allocated inline in a structure. They can be built-in (standard primitive types), user-defined (defined in source code) or enumerations (sets of enumerated values that are represented by labels but stored as a numeric type).
2. Reference Types: Store a reference to the value‘s memory address and are allocated on the heap. Reference types can be any of the pointer types, interface types or self-describing types (arrays and class types such as user-defined classes, boxed value types and delegates).

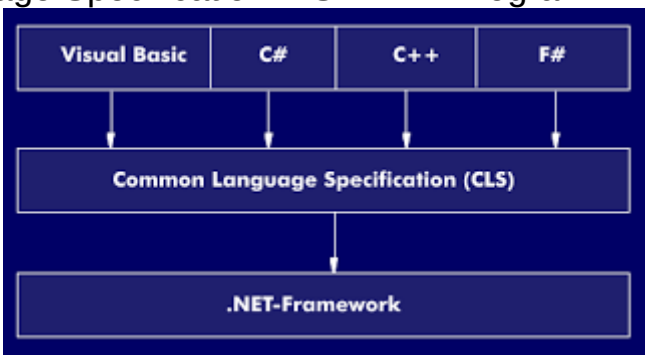


CTS Data Types

CTS Data Type	VB .NET Keyword	C# Keyword	Managed Extensions for C++ Keyword
System.Byte	Byte	byte	unsigned char
System.SByte	SByte	sbyte	signed char
System.Int16	Short	short	short
System.Int32	Integer	int	int or long
System.Int64	Long	long	__int64
System.UInt16	UShort	ushort	unsigned short
System.UInt32	UInteger	uint	unsigned int or unsigned long
System.UInt64	ULong	ulong	unsigned __int64
System.Single	Single	float	Float
System.Double	Double	double	Double
System.Object	Object	object	Object^
System.Char	Char	char	wchar_t
System.String	String	string	String^
System.Decimal	Decimal	decimal	Decimal
System.Boolean	Boolean	bool	Bool

1.1.4) COMMON LANGUAGE SPECIFICATION - CLS

Common Language Specification: ASP.NET Programming Concept



The CLS rules basically define a subset of the Common Type System. The Common Language Specification is the basic set of rules to which any language which is targeting the Common Language Infrastructure (CLI) should confirm in order to interoperate with other CLS-compliant languages. It has been defined as a set of basic language features needed by many applications, which the ASP.NET developers need to develop the applications and services.

When it comes to communication between different objects in .NET languages, those objects must expose all the features that are common to all the languages. The CLS ensures complete interoperability among applications, regardless of the language used to create the application.

Common Language Specification (CLS) is a set of basic language features that .Net Languages needed to develop Applications and Services , which are compatible with the .Net Framework. When there is a situation to communicate Objects written in different .Net Complaint languages , those objects must expose the features that are common to all the languages . Common Language Specification (CLS) ensures complete interoperability among applications, regardless of the language used to create the application.

Common Language Specification (CLS) defines a subset of Common Type System (CTS) . Common Type System (CTS) describes a set of types that can use different .Net languages have in common , which ensure that objects written in different languages can interact with each other. Most of the members defined by types in the .NET Framework Class Library (FCL) are Common Language Specification (CLS) compliant Types.

1.1.5) Assembly

The .NET assembly is the standard for components developed with the Microsoft.NET. Dot NET assemblies may or may not be executable, i.e., they might exist as the executable (.exe) file or dynamic link library (DLL) file. All the .NET assemblies contain

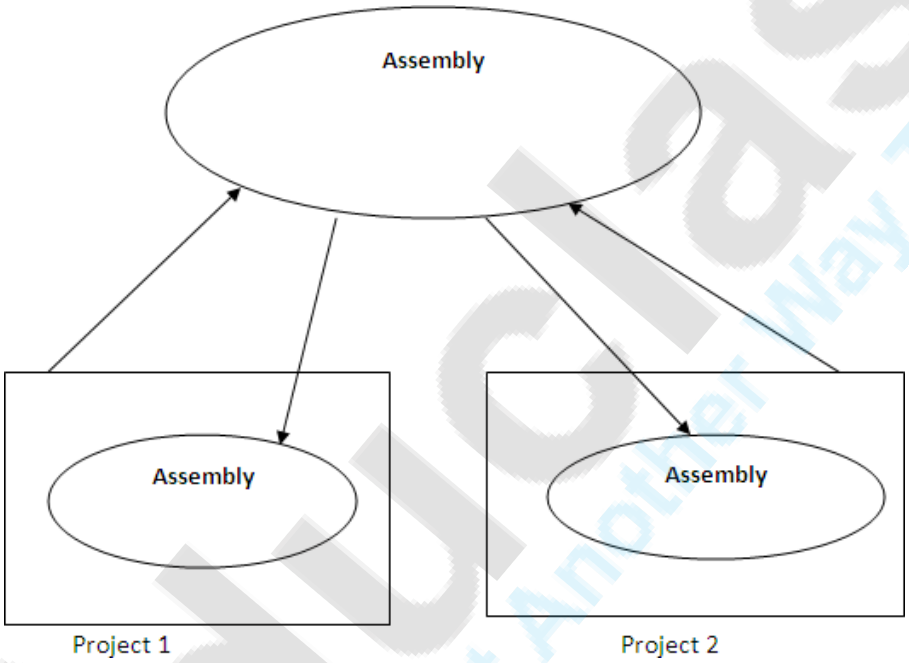
the definition of types, versioning information for the type, meta-data, and manifest. The designers of .NET have worked a lot on the component (assembly) resolution.

There are two kind of assemblies in .NET;

- private
- shared

Private assemblies are simple and copied with each calling assemblies in the calling assemblies folder.

By default every assembly is a private assembly. If we add a reference to a private assembly to any project, a copy of the assembly is given to the project. So each project maintains a private copy of the assembly such as shown below:



Shared Assembly:

A shared assembly is an assembly that resides in a centralized location known as the GAC (Glob Assembly Cache) and that provides resources to multiple applications. If an assembly is shared th multiple copies will mot be created even when used by multiple applications. The GAC folder is und the Windows folder.

<drive>:\windows\assembly <-- GAC folder

We can find all base class libraries under GAC.

We can only get a strong named assembly into the GAC.

GAC contains multiple assemblies and it is identified by PUBLIC KEY TOKEN.

How to generate a public key token:

We have a tool named strong name utility to do this, which is a command line tool and should be us from a command prompt as following.

Sn -k <file name>

E.g.:<drive>:\<folder> > sn -k key.snk

The above atatement generates a key value and writes it into "key.snk".

We can use sn or snk for the extension of key files.

Creating A shared assembly

Step 1: Generate a key file. Open a VS command prompt. Go into your folder and generate a key f as:

<drive>:\<folder> sn -k key.snk

Step 2: create a project and associate a key file to it before compilation so that the generat assembly will be strong named.

Open a new project of type class library and name it sAssembly; under class1 write the following:

```
Public string sayhello()
{
Return "hello from shared assembly";
}
```

To associate a key file we generated with the project, open the project properties and select t "signing" tab on the LHS which displays a CheckBox as "sign the assembly" select it that displa ComboBox below it from it select browse and select key.snk from its physical location then comp the project using build which will generate assembly Assembly.dll that is strong named.

Step 3: copying the assembly into GAC

.Net provides a command line utility to be used as shown in the following:

Gacutil -I | -u <assembly name> I:install u:uninstall

Open a VS command prompt; go to the location where the Assembly.dll is present and write t following:

<drive>:\<folder>\sAssembly\ sAssembly\\bin\Debug>gacutil -I Assembly.dll

Step 4: Testing

Open a new project add a reference to Assembly.dll and write the following code for the button cli event.

```
sAssembly.Classs1 obj=new sAssembly.Class1();
Messagebox.Show(obj.sayhello());
```

1.1.6) METADETA

Metadata in .Net is binary information which describes the characteristics of a resource . This information include Description of the Assembly , Data Types and members with their declarations and implementations, references to other types and members , Security permissions etc. A module's metadata contains everything that needed to interact with another module.

During the compile time Metadata created with Microsoft Intermediate Language (MSIL) and stored in a file called a Manifest . Both Metadata and Microsoft Intermediate Language (MSIL) together wrapped in a Portable Executable (PE) file. During the runtime of a program Just In Time (JIT) compiler of the Common Language Runtime (CLR) uses the Metadata and converts Microsoft Intermediate Language (MSIL) into native code. When code is executed, the runtime loads metadata into memory and references it to discover information about your code's classes, members, inheritance, and so on. Moreover Metadata eliminating the need for Interface Definition Language (IDL) files, header files, or any external method of component reference

The metadata contains the following information:

- Description of assembly
  - Identity (name, version, culture, public key).
  - Types that are exported.
  - Other assemblies where this assembly depends on.
  - The security permissions required to run.
- Description of types
  - Name, visibility, base class and interfaces implemented
  - Members (methods, fields, properties, events and nested types)
- Attributes
  - The additional description of the elements that modify types and members.

Benefits of Metadata

Metadata is a key to simple programming model and it eliminates the need for Interface Definition Language (IDL) files, header files or any other external method of the component reference. The major benefits provided by metadata are:

- Self-describing files: In CLR modules and assemblies are self-describing. The module's metadata consists of everything required to communicate with another module.
- Language interoperability: The metadata provides language interoperability and easier component based design. It also provides all the information required of the compiled code to inherit a class from a PE file which is written in different language.
- Attributes: .Net Framework allows declaring specific kind of metadata which is known as attributes in the compiled file.

Metadata and Portable Executable File Structure

The metadata is stored in one section of .Net Framework PE file while MSIL is stored in another portion of the PE file. The metadata of the file consists of table and heap data structures. The MSIL consists of the MSIL and metadata tokens which refer the metadata portion of the PE file.

Metadata tables and heaps

Each of the metadata table holds the information about the elements of the program. For example one metadata table describes the classes in our code, the second table

will describe the fields and so on. If there are ten classes in our code then class table will have ten rows one for each class.

The metadata table reference other tables and heaps. The metadata also stores the information in four heap structures such as string, blob, user string and GUID.

Metadata Tokens:

Every row of each metadata table is uniquely identified in the MSIL portion of the PE file by metadata token. Metadata tokens concepts are similar to pointers, persisted in MSIL which refer a particular metadata table. The token is four byte number. The first byte refers to denote the metadata table which refer a particular token (method, type, etc.). The other three remaining bytes specify the row in the metadata table.

Metadata within a PE file:

When the program is compiled for the CLR then it is converted into PE file which consists of three parts. Here is the table given below:

PE section	Contents
PE header	This is the index of the PE's file main section and the address of the entry point.
MSIL instructions	MSIL makes up our code and the instructions are accompanied by metadata tokens.
Metadata	This is section used by the runtime to record the information about every type and member in the given code.

Use of Metadata at Runtime

Let us understand metadata better and its role in CLR. Let us understand this through an example that how it affects metadata run time life.

```
using System;
public class Program
{
 public static int Main ()
 {
 int a = 5;
 int b = 2;
 Console.WriteLine ("The Value is: {0}", Mul (a, b));
 return 0;
 }
 public static int Mul (int x, int y)
 {
```

```
 return (x * y);
 }
}
```

1.1.7) GLOBAL ASSEMBLY CACHE [GAC]

Each computer on which the Common Language Runtime is installed has a machine-wide code cache called the 'Global Assembly Cache'. GAC is a folder in Windows directory to store the .NET assemblies that are specifically designated to be shared by all applications executed on a system. Assemblies can be shared among multiple applications on the machine by registering them in global Assembly cache(GAC).

The GAC is automatically installed with the .NET runtime. The global assembly cache is located in 'Windows/WinNT' directory and inherits the directory's access control list that administrators have used to protect the folder.

The approach of having a specially controlled central repository addresses the shared library concept and helps to avoid pitfalls of other solutions that lead to drawbacks like DLL hell.

The Global Assembly Cache Tool (Gacutil.exe), that allows you to view and manipulate the contents of the Global Assembly Cache.

Where is GAC (Global Assembly Cache) located?

GAC is located in %windir%\assembly (for example, C:\WINDOWS\assembly) and it is a shared repository of libraries.

How to view the Contents of the Global Assembly Cache (GAC)?

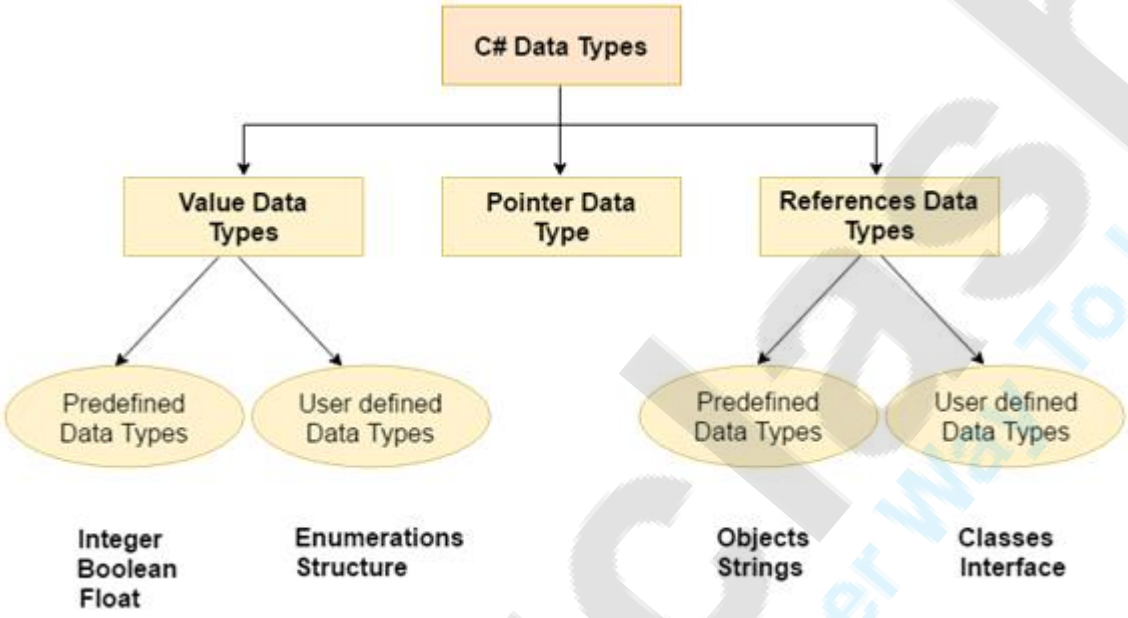
At the Visual Studio command prompt, type the following command:

gacutil -l or gacutil /l

1.2) C# BASICS

1.2.1) C# Data Types

A data type specifies the type of data that a variable can store such as integer, floating, character etc.



There are 3 types of data types in C# language.

Types	Data Types
Value Data Type	short, int, char, float, double etc
Reference Data Type	String, Class, Object and Interface
Pointer Data Type	Pointers

Value Type and Reference Type:

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

1. Value type
2. Reference type

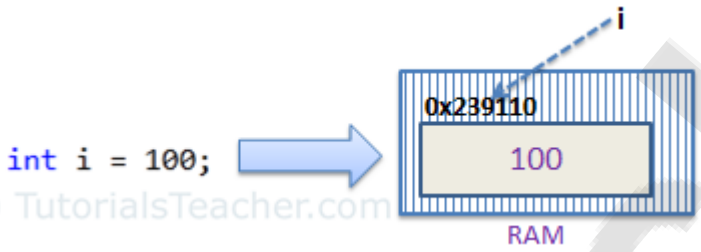
Value Type:

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

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

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

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



Memory allocation for Value Type

The following data types are all of value type:

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

Passing by Value:

When you pass a value type variable from one method to another method, the system creates a separate copy of a variable in another method, so that if value got changed in the one method won't affect on the variable in another method.

Example: Value type passes by value

```
static void ChangeValue(int x)
{
    x =  200;

    Console.WriteLine(x);
}

static void Main(string[] args)
{
    int i = 100;

    Console.WriteLine(i);

    ChangeValue(i);

    Console.WriteLine(i);
}
```

Try it

Output:

```
100
200
100
```

In the above example, variable `i` in Main() method remains unchanged even after we pass it to the ChangeValue() method and change it's value there.

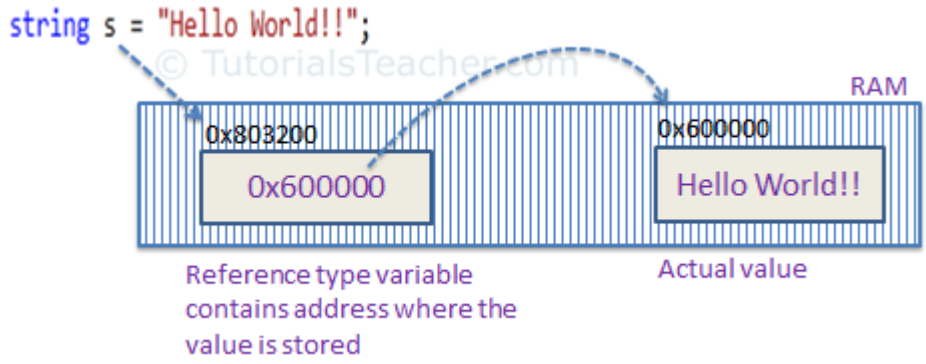
Reference type:

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

For example, consider following string variable:

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

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



Memory allocation for Reference type

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

The following data types are of reference type:

- String
- All arrays, even if their elements are value types
- Class
- Delegates
- In other words, they refer to a memory location. Using multiple variables, the reference types can refer to a memory location. If the data in the memory location is changed by one of the variables, the other variable automatically reflects this change in value.
  Example of **built-in** reference types
  are: **object**, **dynamic,** and **string**.

Object Type

- The **Object Type** is the ultimate base class for all data types in C# Common Type System (CTS). Object is an alias for System.Object class. The object types can be assigned values of any other types, value types, reference types, predefined or user-defined types. However, before assigning values, it needs type conversion.
- When a value type is converted to object type, it is called **boxing** and on the other hand, when an object type is converted to a value type, it is called **unboxing**.

```
object obj;
obj = 100; // this is boxing
```

Dynamic Type

- You can store any type of value in the dynamic data type variable. Type checking for these types of variables takes place at run-time.
- Syntax for declaring a dynamic type is −

```
dynamic <variable_name> = value;
```

- For example,

```
dynamic d = 20;
```

- Dynamic types are similar to object types except that type checking for object type variables takes place at compile time, whereas that for the dynamic type variables takes place at run time.

String Type

- The **String Type** allows you to assign any string values to a variable. The string type is an alias for the System.String class. It is derived from object type. The value for a string type can be assigned using string literals in two forms: quoted and @quoted.
- For example,

```
String str = "Tutorials Point";
```

- A @quoted string literal looks as follows −

```
@"Tutorials Point";
```

- The user-defined reference types are: class, interface, or delegate. We will discuss these types in later chapter.

Pointer Type

- Pointer type variables store the memory address of another type. Pointers in C# have the same capabilities as the pointers in C or C++.
- Syntax for declaring a pointer type is −

```
type* identifier;
```

- For example,

```
char* cptr;
int* iptr;
```

Passing by Reference:

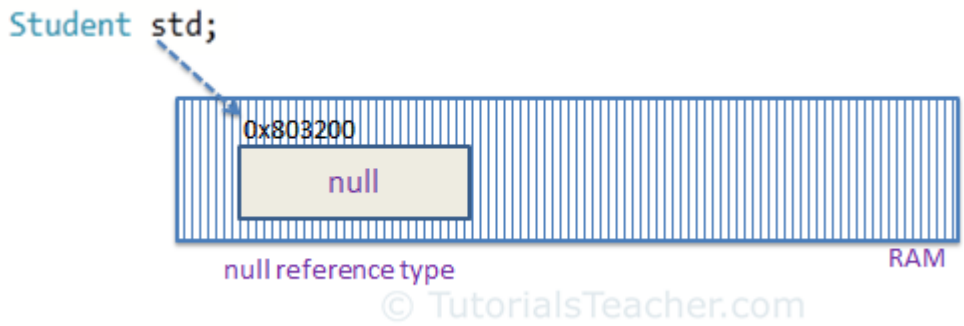
When you pass a reference type variable from one method to another, it doesn't create a new copy; instead, it passes the address of the variable. If we now change the value of the variable in a method, it will also be reflected in the calling method.

Example: Reference type variable passes by reference

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

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

    ChangeReferenceType(std1);

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

null value:

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



Null Reference type

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

Example: Compile time error

```
void someFunction()
{
    int i;

    Console.WriteLine(i);
}
```

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

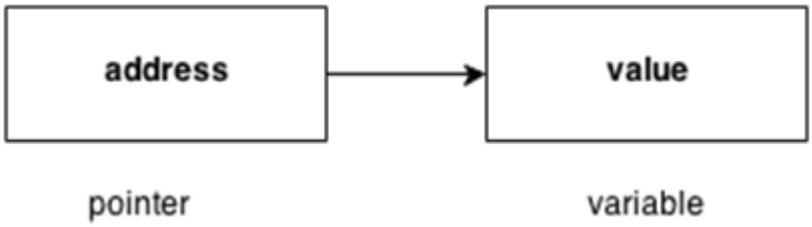
However, value type field in a class can be declared without initialization (field not a local variable in the function) . It will have a default value if not assigned any value, e.g., int will have 0, boolean will have false and so on.

Example: Value type field

```
class myClass
{
    public int i;
}

myClass mcls = new myClass();

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

Pointer Data Type

The pointer in C# language is a variable, it is also known as locator or indicator that points to an address of a value.



Symbols used in pointer

Symbol	Name	Description
& (ampersand sign)	Address operator	Determine the address of a variable.
* (asterisk sign)	Indirection operator	Access the value of an address.

Declaring a pointer

The pointer in C# language can be declared using * (asterisk symbol).

```
int * a;  //pointer to int
char * c; //pointer to char
```

A program executes from top to bottom except when we use control statements, we can control the order of execution of the program, based on logic and values.

In C#, control statements can be divided into the following three categories:

- Selection Statements
- Iteration Statements
- Jump Statements

1.2.2) CONTROL STRUCTURES

Selection Statements

Selection statements allow you to control the flow of program execution on the basis of the outcome of an expression or state of a variable known during runtime.

Selection statements can be divided into the following categories:

- The if and if-else statements
- The if-else statements
- The if-else-if statements
- The switch statements

The if statements

The first contained statement (that can be a block) of an if statement only executes when the specified condition is true. If the condition is false and there is not else keyword then the first contained statement will be skipped and execution continues with the rest of the program. The condition is an expression that returns a boolean value.

Example

```
using System;

namespace Conditional
{
	class IfStatement
	{
		public static void Main(string[] args)
		{
			int number = 2;
			if (number < 5)
			{
				Console.WriteLine("{0} is less than 5", number);
			}

			Console.WriteLine("This statement is always executed.");
		}
	}
}
```

The if-else statements

In if-else statements, if the specified condition in the if statement is false, then the statemet after the else keyword (that can be a block) will execute.

Example

```
sing System;

namespace Conditional
{
	class IfElseStatement
	{
		public static void Main(string[] args)
		{
			int number = 12;

			if (number < 5)
			{
				Console.WriteLine("{0} is less than 5", number);
			}
			else
			{
				Console.WriteLine("{0} is greater than or equal to 5",
number);
			}

			Console.WriteLine("This statement is always executed.");
		}
	}
```

}

The if-else-if statements

This statement following the else keyword can be another if or if-else statement.

That would looks like this:

```
if(condition)
  statements;
else if (condition)
  statements;
else if(condition)
  statement;
else
  statements;
```

Whenever the condition is true, the associated statement will be executed and the remaining conditions will be bypassed. If none of the conditions are true then the else block will execute.

```
using System;

namespace Conditional
{
	class Nested
	{
		public static void Main(string[] args)
		{
			int first = 7, second = -23, third = 13;
			if (first > second)
			{
				if (firstNumber > third)
				{
					Console.WriteLine("{0} is the largest", first);
				}
				else
				{
					Console.WriteLine("{0} is the largest", third);
				}
			}
			else
			{
				if (second > third)
				{
					Console.WriteLine("{0} is the largest", second);
				}
				else
				{
					Console.WriteLine("{0} is the largest", third);
				}
			}
```

```
                }
        }
}
```

The Switch Statements

The switch statement is a multi-way branch statement. The switch statement of C# is another selection statement that defines multiple paths of execution of a program. It provides a better alternative than a large series of if-else-if statements.

Example

```
using System;

namespace Conditional
{
   class SwitchCase
   {
      public static void Main(string[] args)
      {
         char ch;
         Console.WriteLine("Enter an alphabet");
         ch = Convert.ToChar(Console.ReadLine());

         switch(Char.ToLower(ch))
         {
            case 'a':
               Console.WriteLine("Vowel");
               break;
            case 'e':
               Console.WriteLine("Vowel");
               break;
            case 'i':
               Console.WriteLine("Vowel");
               break;
            case 'o':
               Console.WriteLine("Vowel");
               break;
            case 'u':
               Console.WriteLine("Vowel");
               break;
            default:
               Console.WriteLine("Not a vowel");
               break;
         }
      }
   }
}
```

An expression must be of a type of byte, short, int or char. Each of the values specified in the case statement must be of a type compatible with the expression. Duplicate case values are not allowed.

The break statement is used inside the switch to terminate a statement sequence. The break statement is optional in the switch statement.

Iteration Statements

Repeating the same code fragment several times until a specified condition is satisfied is called iteration. Iteration statements execute the same set of instructions until a termination condition is met.

C# provides the following loop for iteration statements:

- The while loop
- The for loop
- The do-while loop
- The for each loop

The while loop

It continually executes a statement (that is usually be a block) while a condition is true. The condition must return a boolean value.

Example

```
using System;

namespace Loop
{
	class WhileLoop
	{
		public static void Main(string[] args)
		{
			int i=1;
			while (i<=5)
			{
				Console.WriteLine("C# For Loop: Iteration {0}", i);
				i++;
			}
		}
	}
}
```

The do-while loop

The only difference between a while and a do-while loop is that do-while evaluates its expression at the bottom of the loop instead of the top. The do-while loop executes at least one time then it

will check the expression prior to the next iteration.

Example

```
using System;

namespace Loop
{
	class DoWhileLoop
	{
		public static void Main(string[] args)
		{
			int i = 1, n = 5, product;

			do
			{
				product = n * i;
				Console.WriteLine("{0} * {1} = {2}", n, i, product);
				i++;
			} while (i <= 10);
		}
	}
}
```

The for loop

A for loop executes a statement (that is usually a block) as long as the boolean condition evaluates to true. A for loop is a combination of the three elements initialization statement, boolean expression and increment or decrement statement.

Syntax:

for(<initialization>;<condition>;<increment or decrement statement>){
<block of code>
}

The initialization block executes first before the loop starts. It is used to initialize the loop variable.

The condition statement evaluates every time prior to when the statement (that is usually be a block) executes, if the condition is true then only the statement (that is usually a block) will execute.

The increment or decrement statement executes every time after the statement (that is usually a block).

Example

```
using System;

namespace Loop
```

```
{
        class ForLoop
        {
                public static void Main(string[] args)
                {
                        for (int i=1; i<=5; i++)
                        {
                                Console.WriteLine("C# For Loop: Iteration {0}", i);
                        }
                }
        }
```

The For each loop

This was introduced in C# 5. This loop is basically used to traverse the array or collection elements.

Example

```
using System;

namespace Loop
{
  class ForEachLoop
  {
    public static void Main(string[] args)
    {
      char[] myArray = {'H','e','l','l','o'};

      foreach(char ch in myArray)
      {
        Console.WriteLine(ch);
      }
    }
  }
}
```

C# switch:

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

Syntax:

```
switch(expression)
{
```

```
    case <value1>
        // code block
    break;
    case <value2>
        // code block
    break;
    case <valueN>
        // code block
    break;
    default
        // code block
    break;
}
```

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

Example: switch statement

```
int x = 10;

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

1.2.3) OPERATORS

- Basic Assignment Operator
- Arithmetic Operators
- Relational Operators
- Logical Operators
- Unary Operators
- Ternary Operator
- Bitwise and Bit Shift Operators
- Compound Assignment Operators

Operators are symbols that are used to perform operations on operands. Operands may be variables and/or constants.

For example, in 2+3, + is an operator that is used to carry out addition operation, while 2 and 3 are operands.

Operators are used to manipulate variables and values in a program. C# supports a number of operators that are classified based on the type of operations they perform.

1. Basic Assignment Operator

Basic assignment operator (=) is used to assign values to variables. For example,

```
double x;
x = 50.05;
```

Here, 50.05 is assigned to x.

Example 1: Basic Assignment Operator

```
using System;

namespace Operator
{
        class AssignmentOperator
        {
                public static void Main(string[] args)
                {
                        int firstNumber, secondNumber;
                        // Assigning a constant to variable
                        firstNumber = 10;
                        Console.WriteLine("First Number = {0}", firstNumber);

                        // Assigning a variable to another variable
                        secondNumber = firstNumber;
                        Console.WriteLine("Second Number = {0}", secondNumber);
                }
        }
}
```

When we run the program, the output will be:

```
First Number = 10

Second Number = 10
```

This is a simple example that demonstrates the use of assignment operator.

You might have noticed the use of curly brackets `{ }` in the example. We will discuss about them in *string formatting*. For now, just keep in mind that `{0}` is replaced by the first variable that follows the string, `{1}` is replaced by the second variable and so on.

2. Arithmetic Operators

Arithmetic operators are used to perform arithmetic operations such as addition, subtraction, multiplication, division, etc.

For example,

```
int x = 5;
int y = 10;
int z = x + y;// z = 15
```

C# Arithmetic Operators

Operator	Operator Name	Example
+	Addition Operator	6 + 3 evaluates to 9
-	Subtraction Operator	10 - 6 evaluates to 4
*	Multiplication Operator	4 * 2 evaluates to 8
/	Division Operator	10 / 5 evaluates to 2
%	Modulo Operator (Remainder)	16 % 3 evaluates to 1

3. Relational Operators

Relational operators are used to check the relationship between two operands. If the relationship is true the result will be true, otherwise it will result in false.

Relational operators are used in decision making and loops.

C# Relational Operators

Operator	Operator Name	Example
==	Equal to	6 == 4 evaluates to false
>	Greater than	3 > -1 evaluates to true
<	Less than	5 < 3 evaluates to false
>=	Greater than or equal to	4 >= 4 evaluates to true
<=	Less than or equal to	5 <= 3 evaluates to false
!=	Not equal to	10 != 2 evaluates to true

4. Logical Operators

Logical operators are used to perform logical operation such as `and`, `or`. Logical operators operates on boolean expressions (`true` and `false`) and returns boolean values. Logical operators are used in decision making and loops.

Here is how the result is evaluated for logical `AND` and `OR` operators.

C# Logical operators

Operand 1	Operand 2	OR (||)	AND (&&)
true	true	true	true
true	false	true	false
false	true	true	false
false	false	false	false

5. Unary Operators

Unlike other operators, the unary operators operates on a single operand.

C# unary operators

Operator	Operator Name	Description
+	Unary Plus	Leaves the sign of operand as it is
-	Unary Minus	Inverts the sign of operand
++	Increment	Increment value by 1
--	Decrement	Decrement value by 1
!	Logical Negation (Not)	Inverts the value of a boolean

6. Ternary Operator
The ternary operator ? : operates on three operands. It is a shorthand for if-then-else statement. Ternary operator can be used as follows:

variable = Condition? Expression1 : Expression2;
The ternary operator works as follows:

If the expression stated by Condition is true, the result of Expression1 is assigned to variable.
If it is false, the result of Expression2 is assigned to variable.

Example 7: Ternary Operator

```
using System;

namespace Operator
{
        class TernaryOperator
        {
                public static void Main(string[] args)
                {
                        int number = 10;
                        string result;

                        result = (number % 2 == 0)? "Even Number" : "Odd Number";
                        Console.WriteLine("{0} is {1}", number, result);
                }
        }
}
```

7. Bitwise and Bit Shift Operators

Bitwise and bit shift operators are used to perform bit manipulation operations.

C# Bitwise and Bit Shift operators

Operator	Operator Name
~	Bitwise Complement
&	Bitwise AND
|	Bitwise OR
^	Bitwise Exclusive OR
<<	Bitwise Left Shift
>>	Bitwise Right Shift

8. Compound Assignment Operators

C# Compound Assignment Operators

Operator	Operator Name	Example	Equivalent To
+=	Addition Assignment	`x += 5`	`x = x + 5`
-=	Subtraction Assignment	`x -= 5`	`x = x - 5`
*=	Multiplication Assignment	`x *= 5`	`x = x * 5`
/=	Division Assignment	`x /= 5`	`x = x / 5`
%=	Modulo Assignment	`x %= 5`	`x = x % 5`
&=	Bitwise AND Assignment	`x &= 5`	`x = x & 5`
|=	Bitwise OR Assignment	`x |= 5`	`x = x | 5`
^=	Bitwise XOR Assignment	`x ^= 5`	`x = x ^ 5`
<<=	Left Shift Assignment	`x <<= 5`	`x = x << 5`
>>=	Right Shift Assignment	`x >>= 5`	`x = x >> 5`
=>	Lambda Operator	`x => x*x`	`Returns x*x`

EXPRESSIONS :

- ASP.NET expressions are a declarative way set control properties based on information that is evaluated at run time. For example, you can use expressions to set a property to values that are based on connection strings, application settings, and other values contained within an application's configuration and resource files. Expressions are evaluated at run time when the declarative elements of the page are parsed, and the value represented by the expression is substituted for the expression syntax.
- A common use of expressions is in data source controls to reference a connection string. Rather than including the connection string directly in the data source control as a property value, you can use an expression that specifies where the connection string is in the configuration file. At run time, the expression is resolved by reading the connection string from the configuration file. You can use expressions for any property setting that you want to resolve at run time rather than set as a static value.

Basic Syntax

The basic syntax of an ASP.NET expression is the following:

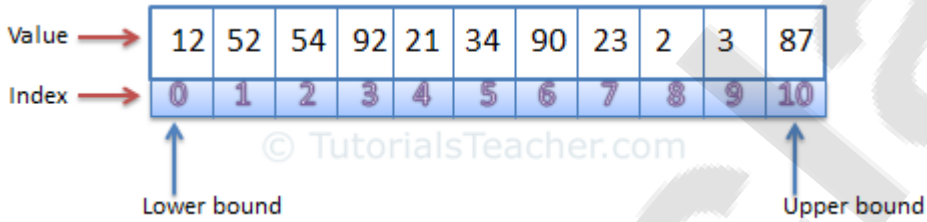
```
<%$ expressionPrefix: expressionValue %>
```

1.2.4) ARRAY:

We have learned that a variable can hold only one literal value, for example `int x = 1;`. Only one literal value can be assigned to a variable x. Suppose, you want to store 100 different values then it will be cumbersome to create 100 different variables. To overcome this problem, C# introduced an array.

An array is a special type of data type which can store fixed number of values sequentially using special syntax.

The following image shows how an array stores values sequentially.



As you can see in the above figure, index is a number starting from 0, which stores the value. You can store a fixed number of values in an array. Array index will be increased by 1 sequentially till the maximum specified array size.

Array Declaration:

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

Example: Array declaration in C#

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

bool[] boolArray; // can store boolean values

string[] stringArray; // can store string values

double[] doubleArray; // can store double values

byte[] byteArray; // can store byte values

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

Initialization:

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

Example: Array Declaration & Initialization

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

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

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

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

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

Example: Wrong way of initializing an array

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

Accessing Array Elements:

As shown above, values can be assigned to an array at the time of initialization. However, value can also be assigned to individual index randomly as shown below.

Example: Assigning values to array index

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

intArray[0] = 10;

intArray[1] = 20;

intArray[2] = 30;

intArray[3] = 40;

intArray[4] = 50;
```

Array properties and methods:

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

Multi-dimensional Array:

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

```
string [,] names;
```

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

```
int [ , , ] m;
```

Two-Dimensional Arrays

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. You can verify it in the above diagram. Let us check the program to handle a two dimensional array –

Live Demo

```
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();
      }
   }
}
```

When the above code is compiled and executed, it produces the following result –

```
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
```

Jagged Array:

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

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

Declaration of Jagged array

Let's see an example to declare jagged array that has two elements.

```
1. int[][] arr = new int[2][];
```

Initialization of Jagged array

Let's see an example to initialize jagged array. The size of elements can be different.

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

Initialization and filling elements in Jagged array

Let's see an example to initialize and fill elements in jagged array.

```
1. arr[0] = new int[4] { 11, 21, 56, 78 };
2. arr[1] = new int[6] { 42, 61, 37, 41, 59, 63 };
```

Here, size of elements in jagged array is optional. So, you can write above code as given below:

1. arr[0] = **new int**[] { 11, 21, 56, 78 };
2. arr[1] = **new int**[] { 42, 61, 37, 41, 59, 63 };

C# Jagged Array Example

Let's see a simple example of jagged array in C# which declares, initializes and traverse jagged arrays.

```
public class JaggedArrayTest
{
  public static void Main()
  {
    int[][] arr = new int[2][];// Declare the array

    arr[0] = new int[] { 11, 21, 56, 78 };// Initialize the array
    arr[1] = new int[] { 42, 61, 37, 41, 59, 63 };

    // Traverse array elements
    for (int i = 0; i < arr.Length; i++)
    {
      for (int j = 0; j < arr[i].Length; j++)
      {
        System.Console.Write(arr[i][j]+" ");
      }
      System.Console.WriteLine();
    }
  }
}
```

Output:

```
11 21 56 78
42 61 37 41 59 63
```

UNIT 2

OOP C#

1)CLASSES AND OBJECTS:

1.1) C# Class:

A class is like a blueprint of specific object. In the real world, every object has some color, shape and functionalities. For example, the luxury car Ferrari. Ferrari is an object of the luxury car type. The luxury car is a class that specify certain characteristic like speed, color, shape, interior etc. So any company that makes a car that meet those requirements is an object of the luxury car type. For example, every single car of BMW, lamborghini, cadillac are an object of the class called 'Luxury Car'. Here, 'Luxury Car' is a class and every single physical car is an object of the luxury car class.

Likewise, in object oriented programming, a class defines certain properties, fields, events, method etc. A class defines the kinds of data and the functionality their objects will have.

A class enables you to create your own custom types by grouping together variables of other types, methods and events.

In C#, a class can be defined by using the class keyword.

```
1. public class Student
2.  {
3.      int id;//field or data member
4.      String name;//field or data member
5.  }
```

1.2) C# Object

In C#, Object is a real world entity, for example, chair, car, pen, mobile, laptop etc.

In other words, object is an entity that has state and behavior. Here, state means data and behavior means functionality.

Object is a runtime entity, it is created at runtime.

Object is an instance of a class. All the members of the class can be accessed through object.

Let's see an example to create object using new keyword.

```
1. Student s1 = new Student();//creating an object of Student
```

In this example, Student is the type and s1 is the reference variable that refers to the instance of Student class. The new keyword allocates memory at runtime.

C# Variable :

A variable is a name of memory location. It is used to store data. Its value can be changed and it can be reused many times.

It is a way to represent memory location through symbol so that it can be easily identified.

The basic variable type available in C# can be categorized as:

Variable Type	Example
Decimal types	decimal
Boolean types	True or false value, as assigned
Integral types	int, char, byte, short, long
Floating point types	float and double
Nullable types	Nullable data types

Let's see the syntax to declare a variable:

```
1. type variable_list;
```

The example of declaring variable is given below:

```
1. int i, j;
2. double d;
3. float f;
4. char ch;
```

Here, i, j, d, f, ch are variables and int, double, float, char are data types.

We can also provide values while declaring the variables as given below:

```
1. int i=2,j=4;  //declaring 2 variable of integer type
2. float f=40.2;
3. char ch='B';
```

Rules for defining variables

- A variable can have alphabets, digits and underscore.
- A variable name can start with alphabet and underscore only. It can't start with digit.
- No white space is allowed within variable name.
- A variable name must not be any reserved word or keyword e.g. char, float etc.

Valid variable names:

1. **int** x;
2. **int** _x;
3. **int** k20;

Invalid variable names:

1. **int** 4;
2. **int** x y;
3. **int double**;

Parse in C#

Used to parse (convert) a string into a primitive data type.

Syntax

```
///standard parse
dataType variableName = dataType.Parse(stringName);

///tryParse takes care of failure by assigning a default value,
no need for try-catch
dataType result;
bool didItWork = dataType.TryParse(stringName, out result);
```

Notes

A standard parse throws an exception if the string cannot be converted to the specified type. It should be used inside a try-catch block.

TryParse returns a boolean to show whether or not the parsing worked. The second argument is used for the output variable, the 'out' keyword assigns the value to the variable.

Example

```
string yearString = "2015";
int year;
bool validYear = int.TryParse(yearString, out year);
```

1.3)Instance Variables:

Instance variables are non-static variables and are declared in a class outside any method, constructor or block.

- As instance variables are declared in a class, these variables are created when an object of the class is created and destroyed when the object is destroyed.
- Unlike local variables, we may use access specifiers for instance variables. If we do not specify any access specifier then the default access specifier will be used.

Instance variables are used by Objects to store their states. Variables which are defined without the ***STATIC keyword*** and are *Outside any method declaration* are Object specific and are known as instance variables. They are called so because their values are instance specific and are ***not*** shared among instances.

Rules for Instance variable

- Instance variables can use any of the four access levels
- They can be marked final
- They can be marked transient
- They cannot be marked abstract
- They cannot be marked synchronized
- They cannot be marked strictfp
- They cannot be marked native
- They cannot be marked static

Instance Variable Type	Default Value
boolean	false
byte	(byte)0
short	(short) 0
int	0
long	0L
char	u0000
float	0.0f
double	0.0d
Object	null

```
1. public class Dog {
2. public String name;
3. }

4. Dog fido = new Dog(); // here is where the blueprint above becomes
   a real
                              i. // structure in memory

5. fido.name = "Scooby Doo"; // Now that instance variable has a
   value.
```

1.4) METHODS

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, you need to –

- Define the method
- Call the method

Defining Methods in C#

When you define a method, you 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**.
- **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

Following code snippet shows a function *FindMax* that takes two integer values and returns the larger of the two. It has public access specifier, so it can be accessed from outside the class using an instance of the class.

```
class NumberManipulator {


   public int FindMax(int num1, int num2) {
      /* local variable declaration */
      int result;


      if (num1 > num2)
         result = num1;
      else
         result = num2;


      return result;
   }
   ...
}
```

Calling Methods in C#

You can call a method using the name of the method. The following example illustrates this –

Live Demo

```
using System;

namespace CalculatorApplication {
   class NumberManipulator {
      public int FindMax(int num1, int num2) {
         /* local variable declaration */
         int result;

         if (num1 > num2)
            result = num1;
         else
```

```
            result = num2;
         return result;
      }

      static void Main(string[] args) {
         /* local variable definition */
         int a = 100;
         int b = 200;
         int ret;
         NumberManipulator n = new NumberManipulator();

         //calling the FindMax method
         ret = n.FindMax(a, b);
         Console.WriteLine("Max value is : {0}", ret );
         Console.ReadLine();
      }
   }
}
```

Passing Parameters to a Method

When method with parameters is called, you need to pass the parameters to the method. There are three ways that parameters can be passed to a method –

Sr.No.	Mechanism & Description
1	**Value parameters** This method copies the actual value of an argument into the formal parameter of the function. In this case, changes made to the parameter inside the function have no effect on the argument.
2	**Reference parameters** This method copies the reference to the memory location of an argument into the formal parameter. This means that changes made to the parameter affect the argument.
3	**Output parameters** This method helps in returning more than one value.

Types Of Methods In C#

Here is the list of Method type in C#.

a. Pure virtual method.
b. Virtual method.
c. Abstract method.
d. Partial method.
e. Extension method.
f. Instance method.
g. Static method.

Pure Virtual Method

Pure virtual method is the term that programmers use in C++. There is a term "abstract" in place of "pure virtual method" in C#.

Example:

```
1.  public void abstract DoSomething();
```

Virtual Method

Virtual method makes some default functionality. In other words, virtual methods are being implemented in the base class and can be overridden in the derived class.

Example:

```
1.  public class A
2.  {
3.      public virtual int Calculate(int a, int b)
4.      {
5.          return a + b;
6.      }
7.  }
8.  public class B: A
9.  {
10.     public override int Calculate(int a, int b)
11.     {
12.         return a + b + 1;
13.     }
14. }
```

Abstract Method

Abstract Method is the method with no implementation and is implicitly virtual. You can make abstract method only in Abstract class.

Example:

```
1.  public void abstract DoSomething(int a);
```

Partial Method

A partial method has its signature defined in one part of a partial type, and its implementation defined in another part of the type.

Example:

```
1.  namespace PM
2.  {
3.      partial class A
4.      {
5.          partial void OnSomethingHappened(string s);
6.      }
7.      // This part can be in a separate file.
8.      partial class A
9.      {
10.         // Comment out this method and the program
```

```
        // will still compile.
        partial void OnSomethingHappened(String s)
        {
            Console.WriteLine("Something happened: {0}", s);
        }
    }
}
```

Extension Method

Extension methods are a special kind of static method, but they are called as if they were instance methods on the extended type.

Extension methods are used to add some functionality to given type.

Note

1. For making Extension methods, you need to have a static class with static method.
2. Do not make Extension method for one or two lines of code, write Extension method for logic.

E.g.

```
public static class ExtensionMethods
{
    public static string UppercaseFirstLetter(this string value)
    {
        //
        // Uppercase the first letter in the string.
        //
        if (value.Length > 0)
        {
            char[] array = value.ToCharArray();
            array[0] = char.ToUpper(array[0]);
            return new string(array);
        }
        return value;
    }
}
class Program
{
    static void Main()
    {
        //
        // Use the string extension method on this value.
        //
        string value = "deeksha sharma";
        value = value.UppercaseFirstLetter();
        Console.WriteLine(value);
    }
}
```

Instance method

An instance method operates on a given instance of a class, and that instance can be accessed as this.

```
public class PropertyUtil
{
    public void DoSomething()
```

```
4.      {
5.          // Do Something.
6.      }
7.      public void DoSomethingElse()
8.      {
9.          this.DoSomething(); // Calling to instance method.
10.     }
11. }
12. public class Program
13. {
14.     static void Main()
15.     {
16.         PropertyUtil util = new PropertyUtil();
17.         util.DoSomething(); // Calling to instance method.
18.     }
19. }
```

Static method

This belongs to the type, it does not belong to instance of the type. You can access static methods by class name.

You can put static method in static or non- static classes.

The only difference is that static methods in a non-static class cannot be extension methods.

Static often improves performance.

```
1.  public static class StaticClass
2.  {
3.      public static int DoSomething()
4.      {
5.          return 2;
6.      }
7.  }
8.  public class Program
9.  {
10.     static void Main()
11.     {
12.         StaticClass.DoSomething();
13.     }
14. }
```

1.5) CONSTRUCTOR

What is constructor?

A special method of the class that will be automatically invoked when an instance of the class is created is called a constructor. The main use of constructors is to initialize private fields of the class while creating an instance for the class. When you have not created a constructor in the class, the compiler will automatically create a default constructor in the class. The default constructor initializes all numeric fields in the class to zero and all string and object fields to null.

Some of the key points regarding the Constructor are:

- A class can have any number of constructors.
- A constructor doesn't have any return type, not even void.
- A static constructor can not be a parametrized constructor.

- Within a class you can create only one static constructor.

Constructors can be divided into 5 types:

1. Default Constructor
2. Parametrized Constructor
3. Copy Constructor
4. Static Constructor
5. Private Constructor

Now let us see each constructor type with example as below

Default Constructor

A constructor without any parameters is called a default constructor; in other words this type of constructor does not take parameters. The drawback of a default constructor is that every instance of the class will be initialized to the same values and it is not possible to initialize each instance of the class to different values. The default constructor initializes:

1. All numeric fields in the class to zero.
2. All string and object fields to null.

Example

```
using System;
namespace ConsoleApplication3
{
class Sample
{
public string param1, param2;
public Sample()      // Default Constructor
{
param1 = "Welcome";
param2 = "Aspdotnet-Suresh";
}
}
class Program
{
static void Main(string[] args)
{
Sample obj=new Sample();    // Once object of class created automatically constructor will be called
Console.WriteLine(obj.param1);
Console.WriteLine(obj.param2);
Console.ReadLine();
}
}
}
```

Parameterized Constructor

A constructor with at least one parameter is called a parametrized constructor. The advantage of a parametrizedconstructor is that you can initialize each instance of the class to different values.

using System;

```
namespace Constructor
{
   class paraconstrctor
   {
     public  int a, b;
     public paraconstrctor(int x, int y)  // decalaring Paremetrized Constructor with ing x,y parameter
       {
          a = x;
          b = y;
       }
  }
   class MainClass
   {
      static void Main()
      {
         paraconstrctor v = new paraconstrctor(100, 175);   // Creating object of Parameterized
Constructor and ing values
         Console.WriteLine("-----------parameterized constructor example by vithal wadje----------
");
         Console.WriteLine("\t");
         Console.WriteLine("value of a=" + v.a );
         Console.WriteLine("value of b=" + v.b);
         Console.Read();
      }
   }
}
```

Copy Constructor

The constructor which creates an object by copying variables from another object is called a copy constructor. The purpose of a copy constructor is to initialize a new instance to the values of an existing instance.

Syntax

```
public employee(employee emp)
{
name=emp.name;
age=emp.age;
}
```

The copy constructor is invoked by instantiating an object of type employee and ing it the object to be copied.

```
using System;
namespace ConsoleApplication3
{
class Sample
{
public string param1, param2;
public Sample(string x, string y)
```

```
{
param1 = x;
param2 = y;
}
public Sample(Sample obj)      // Copy Constructor
{
param1 = obj.param1;
param2 = obj.param2;
}
}
class Program
{
static void Main(string[] args)
{
Sample obj = new Sample("Welcome", "Aspdotnet-Suresh");  // Create instance to class Sample
Sample obj1=new Sample(obj); // Here obj details will copied to obj1
Console.WriteLine(obj1.param1 +" to " + obj1.param2);
Console.ReadLine();
}
}
}
```

Static Constructor

When a constructor is created as static, it will be invoked only once for all of instances of the class and it is invoked during the creation of the first instance of the class or the first reference to a static member in the class. A static constructor is used to initialize static fields of the class and to write the code that needs to be executed only once.

Some key points of a static constructor is:

1. A static constructor does not take access modifiers or have parameters.
2. A static constructor is called automatically to initialize the class before the first instance is created or any static members are referenced.
3. A static constructor cannot be called directly.
4. The user has no control on when the static constructor is executed in the program.
5. A typical use of static constructors is when the class is using a log file and the constructor is used to write entries to this file.

Syntax

```
class employee
 {// Static constructor
  static employee(){}
 }
```

Now let us see it with practically

```
using System;
namespace staticConstractor
{
public class employee
{
```

```
    static employee() // Static constructor declaration{Console.WriteLine("The static constructor ");
}
public static void Salary()
 {
    Console.WriteLine();
    Console.WriteLine("The Salary method");
 }
}
class details
{
    static void Main()
    {
        Console.WriteLine("----------Static constrctor example by vithal wadje----------");
        Console.WriteLine();
        employee.Salary();
        Console.ReadLine();
    }
  }
}
```

Private Constructor

When a constructor is created with a private specifier, it is not possible for other classes to derive from this class,

neither is it possible to create an instance of this class. They are usually used in classes that contain static members
only. Some key points of a private constructor are:

1. One use of a private constructor is when we have only static members.
2. It provides an implementation of a singleton class pattern
3. Once we provide a constructor that is either private or public or any, the compiler will not add the parameter-less public constructor to the class.

Now let us see it practically.

```
using System;
namespace defaultConstractor
{
    public class Counter
    {
        private Counter()   //private constrctor declaration
        {
        }
        public static int currentview;
        public static int visitedCount()
        {
            return ++ currentview;
        }
    }
    class viewCountedetails
    {
        static void Main()
        {
            // Counter aCounter = new Counter();   // Error
```

```
            Console.WriteLine("-------Private constructor example by vithal wadje----------");
            Console.WriteLine();
            Counter.currentview = 500;
            Counter.visitedCount();
            Console.WriteLine("Now the view count is: {0}", Counter.currentview);
            Console.ReadLine();
        }
    }
}
```

1.6) PROPERTIES

Properties are named members of classes, structures, and interfaces. Member variables or methods in a class or structures are called **Fields**. Properties are an extension of fields and are accessed using the same syntax. They use **accessors** through which the values of the private fields can be read, written or manipulated.

Properties do not name the storage locations. Instead, they have **accessors**that read, write, or compute their values.

For example, let us have a class named Student, with private fields for age, name, and code. We cannot directly access these fields from outside the class scope, but we can have properties for accessing these private fields.

Accessors

The **accessor** of a property contains the executable statements that helps in getting (reading or computing) or setting (writing) the property. The accessor declarations can contain a get accessor, a set accessor, or both. For example –

```
// Declare a Code property of type string:
public string Code {
   get {
      return code;
   }
   set {
      code = value;
   }
}

// Declare a Name property of type string:
public string Name {
   get {
      return name;
   }
   set {
```

```
        name = value;
    }
}

// Declare a Age property of type int:
public int Age {
    get {
        return age;
    }
    set {
        age = value;
    }
}
```

Usage of C# Properties

1. C# Properties can be read-only or write-only.
2. We can have logic while setting values in the C# Properties.
3. We make fields of the class private, so that fields can't be accessed from outside the class directly. Now we are forced to use C# properties for setting or getting values.

C# Properties Example

```
1.  using System;
2.    public class Employee
3.     {
4.        private string name;
5.
6.        public string Name
7.        {
8.           get
9.           {
10.             return name;
11.          }
12.          set
13.          {
14.             name = value;
15.          }
16.       }
17.   }
18.   class TestEmployee{
19.      public static void Main(string[] args)
20.       {
```

```
21.         Employee e1 = new Employee();
22.         e1.Name = "Sonoo Jaiswal";
23.         Console.WriteLine("Employee Name: " + e1.Name);
24.
25.     }
26.  }
```

Output:

```
Employee Name: Sonoo Jaiswal
```

C# Properties Example 2: having logic while setting value

```
1.  using System;
2.    public class Employee
3.    {
4.       private string name;
5.
6.       public string Name
7.       {
8.          get
9.          {
10.            return name;
11.         }
12.         set
13.         {
14.            name = value+" JavaTpoint";
15.
16.         }
17.      }
18.  }
19.  class TestEmployee{
20.     public static void Main(string[] args)
21.     {
22.         Employee e1 = new Employee();
23.         e1.Name = "Sonoo";
24.         Console.WriteLine("Employee Name: " + e1.Name);
25.     }
26.  }
```

Output:

```
Employee Name: Sonoo JavaTpoint
```

C# Properties Example 3: read-only property

```
1.  using System;
2.    public class Employee
3.     {
4.        private static int counter;
5.
6.        public Employee()
7.        {
8.           counter++;
9.        }
10.       public static int Counter
11.       {
12.          get
13.          {
14.             return counter;
15.          }
16.       }
17.   }
18.   class TestEmployee{
19.      public static void Main(string[] args)
20.       {
21.          Employee e1 = new Employee();
22.          Employee e2 = new Employee();
23.          Employee e3 = new Employee();
24.          //e1.Counter = 10;//Compile Time Error: Can't set value
25.
26.          Console.WriteLine("No. of Employees: " + Employee.Counter);
27.       }
28.    }
```

Output:

```
No. of Employees: 3
```

1.7) Access Specifiers

Access modifiers and specifiers are keywords (private, public, internal, protected and protected internal) to specify the accessibility of a type and its members.

C# has 5 access specifier or access modifier keywords; those are private, public, internal, protected and protected Internal.

Usage of Access Specifiers

private: limits the accessibility of a member to within the defined type, for example if a variable or a functions is being created in a ClassA and declared as private then another ClassB can't access that.

public: has no limits, any members or types defined as public can be accessed within the class, assembly even outside the assembly. Most DLLs are known to be produced by public class and members written in a .cs file.

internal: internal plays an important role when you want your class members to be accessible within the assembly. An assembly is the produced .dll or .exe from your .NET Language code (C#). Hence, if you have a C# project that has ClassA, ClassB and ClassC then any internal type and members will become accessible across the classes with in the assembly.

protected: plays a role only when inheritance is used. In other words any protected type or member becomes accessible when a child is inherited by the parent. In other cases (when no inheritance) protected members and types are not visible.

Protected internal: is a combination of protected and internal both. A protected internal will be accessible within the assembly due to its internal flavor and also via inheritance due to its protected flavor.

Code Sample

```
1.  public class AccessModifier
2.  {
3.        //Available only to the container Class
4.        private string privateVariable;
5.
6.        // Available in entire assembly across the classes
7.        internal string internalVariable;

8.        //Available in the container class and the derived class
          protected string protectedVariable;

          //Available to the container class, entire assembly and to outside
          public string publicVariable;
9.
10.       //Available to the derived class and entire assembly as well
          protected internal string protectedInternalVariable;
11.
12.       private string PrivateFunction()
13.       {
```

```
14.             return privateVariable;
15.         }
16.
17.         internal string InternalFunction()
18.         {
19.             return internalVariable;
20.         }
21.
22.         protected string ProtectedFunction()
23.         {
24.             return protectedVariable;
25.         }
26.
27.         public string PublicFunction()
28.         {
29.             return publicVariable;
30.         }
31.
32.         protected internal string ProtectedInternalFunction()
33.         {
34.              return protectedInternalVariable;
35.         }
36. }
```

Now to show the behavior of how these class members are exposed to another class depending upon their scope defined via modifier/specifier and when we create an object or inherit from the preceding created class named "AccessModifier".

1.8) STATIC MEMBER AND METHOD

In C#, data members, member functions, properties and events can be declared either as static or non-static.

Only one copy of static fields and events exists, and static methods and properties can only access static fields and static events.

Static members are often used to represent data or calculations that do not change in response to object state.

1. Static classes cannot be instantiated using the new keyword
2. Static items can only access other static items. For example, a static class can only contain static members, e.g., variables, methods, etc. A static method can only contain static variables and can only access other static items.
3. Static items share the resources between multiple users.
4. Static cannot be used with indexers, destructors or types other than classes.
5. A static constructor in a non-static class runs only once when the class is instantiated for the first time.
6. A static constructor in a static class runs only once when any of its static members accessed for the first time.
7. Static members are allocated in high frequency heap area of the memory.

Static can be used in following ways:

1. Static data members
2. Static constructor
3. Static Properties
4. Static methods

Static Data Members

A C# class can contain both static and non-static members. When we declare a member with the help of the keyword static, it becomes a static member.

A static member belongs to the class rather than to the objects of the class. Hence static data members are also known as class members and non-static members are known as instance members.

Static members and preloaded in memory while instance members are post loaded in memory.

You can't use **this** keyword with static, as it will not initialize. Static members are shared level and are not initiated.

Note: Indexers in C# can't declared as static.

Static fields can be declared as follows by using the keyword static.

```
class MyClass
  {
     public static int age;
     public static string name = "George";
  }
```

When we declare static data members inside a class, it can be initialized with a value as shown above. All un-initialized static fields automatically get initialized to their default values when the class is loaded first time.

Static constructor

Static constructor can't be parameterized.

Static constructor doesn't have any access modifier because it doesn't have message passing and used during domain processing.

Static Constructor is used to initialize static data members of the class.

Static Properties

Static properties are used to get or set the value of static data members of a class.

Here is a practical demonstration of static property:

using System;

```
using System.Collections.Generic;
using System.Linq;
using System.Text;

namespace properties_static
{
   class Program
   {
      public class PropertyClass
      {
         static string co_name;
         // Static Property
         public static string _co_name
         {
            get
            {
               return co_name;
            }

            set
            {
               co_name = value;
            }
         }

      }

      static void Main(string[] args)
      {
         PropertyClass._co_name = "George";
         Console.WriteLine(PropertyClass._co_name);
         Console.ReadLine();

      }
   }
}
```

Static methods

Static methods are shared methods. They can be called with the class name and static method name only. No need of instance to call static methods.

Static methods only use static data members to perform calculation or processing.

Practical demonstration of static constructor and static methods.

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;

namespace static_eg
{
   class Program
   {
      public class test
```

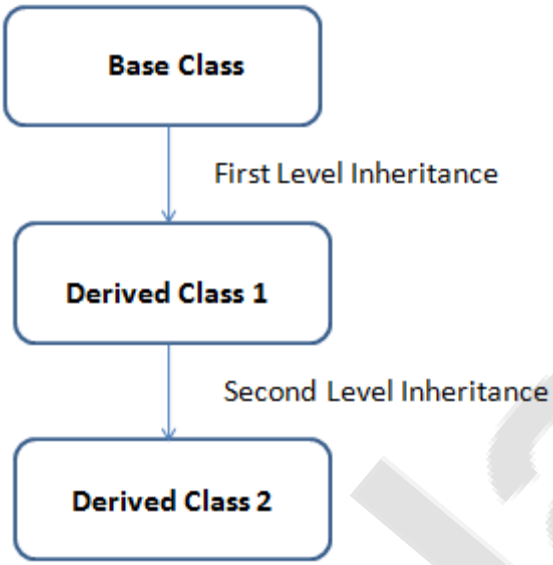
```
      {
         static string name;
         static int age;

         static test()
         {
            Console.WriteLine("Using static constructor to initialize static data
members");
            name = "John Sena";
            age = 23;
         }
         public static void display()
         {

            Console.WriteLine("Using static function");
            Console.WriteLine(name);
            Console.WriteLine(age);
         }

      }
      static void Main(string[] args)
      {
         test.display();

         Console.ReadLine();
      }
   }
}
```

2.1) INHERITANCE

INTRODUCTION

- 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.
- 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.
- In C#, inheritance is a process in which one object acquires all the properties and behaviors of its parent object automatically. In such way, you can reuse, extend or modify the attributes and behaviors which is defined in other class.
- In C#, the class which inherits the members of another class is called **derived class** and the class whose members are inherited is called **base** class. The derived class is the specialized class for the base class.

2.1.1) LEVELS OF INHERITANCE



A class can be derived from more than one class or interface, which means that it can inherit data and functions from multiple base classes or interfaces.

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

```
<acess-specifier> class <base_class> {
   ...
}

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

TYPES OF INHERITANCE

1) SINGLE INHERITANCE

C# Single Level Inheritance Example: Inheriting Fields

When one class inherits another class, it is known as single level inheritance. Let's see the example of single level inheritance which inherits the fields only.

```
using System;
  public class Employee
  {
```

```
    public float salary = 40000;
  }
  public class Programmer: Employee
  {
    public float bonus = 10000;
  }
  class TestInheritance{
    public static void Main(string[] args)
    {
      Programmer p1 = new Programmer();

      Console.WriteLine("Salary: " + p1.salary);
      Console.WriteLine("Bonus: " + p1.bonus);

    }
  }
```

Output:

```
Salary: 40000
Bonus: 10000
```

In the above example, Employee is the **base** class and Programmer is the **derived** class.

C# Single Level Inheritance Example: Inheriting Methods

Let's see another example of inheritance in C# which inherits methods only.

```
using System;
  public class Animal
  {
    public void eat() { Console.WriteLine("Eating..."); }
  }
  public class Dog: Animal
  {
    public void bark() { Console.WriteLine("Barking..."); }
  }
  class TestInheritance2{
    public static void Main(string[] args)
    {
      Dog d1 = new Dog();
      d1.eat();
      d1.bark();
    }
  }
```

Output:

```
Eating...
Barking...
```

2)MULTIPLE INHERITANCE

C# does not support multiple inheritance. However, you can use interfaces to implement multiple inheritance. The following program demonstrates this –

```
using System;

namespace InheritanceApplication {
   class Shape {
      public void setWidth(int w) {
         width = w;
      }
      public void setHeight(int h) {
         height = h;
      }
      protected int width;
      protected int height;
   }

   // Base class PaintCost
   public interface PaintCost {
      int getCost(int area);
   }

   // Derived class
   class Rectangle : Shape, PaintCost {
      public int getArea() {
         return (width * height);
      }
      public int getCost(int area) {
         return area * 70;
      }
   }
   class RectangleTester {
      static void Main(string[] args) {
         Rectangle Rect = new Rectangle();
         int area;

         Rect.setWidth(5);
         Rect.setHeight(7);
         area = Rect.getArea();

         // Print the area of the object.
         Console.WriteLine("Total area: {0}",  Rect.getArea());
         Console.WriteLine("Total paint cost: ${0}" , Rect.getCost(area));
         Console.ReadKey();
      }
   }
```

```
}
```

When the above code is compiled and executed, it produces the following result –

```
Total area: 35
Total paint cost: $2450
```

3)MULTI LEVEL INHERITANCE

When one class inherits another class which is further inherited by another class, it is known as multi level inheritance in C#. Inheritance is transitive so the last derived class acquires all the members of all its base classes.

Let's see the example of multi level inheritance in C#.

```
using System;
  public class Animal
   {
     public void eat() { Console.WriteLine("Eating..."); }
  }
  public class Dog: Animal
  {
     public void bark() { Console.WriteLine("Barking..."); }
  }
  public class BabyDog : Dog
  {
     public void weep() { Console.WriteLine("Weeping..."); }
  }
  class TestInheritance2{
     public static void Main(string[] args)
      {
         BabyDog d1 = new BabyDog();
         d1.eat();
         d1.bark();
         d1.weep();
      }
   }
```

Output:

```
Eating...
Barking...
Weeping...
```

2.1.2) CONSTRUCTORS AND INHERITANCE

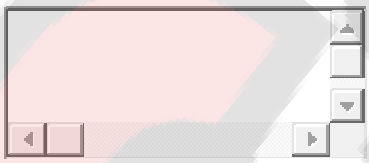
DEFINE BASE CLASS CONSTRUCTOR

There is no special rule for defining base class `constructors`. Rules are same as other class constructors. You can define number of constructor as follows:

```
class baseclass
   {
      public baseclass()
      {

      }
      public baseclass(string message)
      {

      }
   }
```

ACCESS BASE CLASS CONSTRUCTOR IN DERIVED CLASS

If base class has constructor then child class or derived class are required to call the constructor from its base class.



```
class childclass : baseclass
   {
      public childclass()
      {
      }
      public childclass(string message)
         : base(message)
      {
      }
   }
```

EXAMPLE

Following programming example has two constructors in base class. One is default constructor and other has a string parameter message. In child class all these two constructors are called and print message on console.

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

namespace Inheritance_Constructors
{
    class Program
    {
        static void Main(string[] args)
        {
            childclass ch = new childclass();
            childclass ch1 = new childclass("Hello Constructor");
            Console.ReadKey();
        }
    }

    class baseclass
    {
        public baseclass()
        {
            Console.WriteLine("I am Default Constructors");
        }
        public baseclass(string message)
        {
            Console.WriteLine("Constructor Message : " + message);
        }
    }

    class childclass : baseclass
    {
        public childclass()
        {
        }
        public childclass(string message)
            : base(message)
        {
        }
    }
}
```

Output:-

I am Default Constructors
Constructor Message: Hello Constructor

2.1.3) POLYMORPHISM

The word **polymorphism** means having many forms. In object-oriented programming paradigm, polymorphism is often expressed as 'one interface, multiple functions'.

Polymorphism can be static or dynamic. In **static polymorphism**, the response to a function is determined at the compile time. In **dynamic polymorphism**, it is decided at run-time.

There are two types of polymorphism in C#: **compile time polymorphism** and **runtime polymorphism**. Compile time polymorphism is achieved by method overloading and operator overloading in C#. It is also known as static binding or early binding. Runtime polymorphism in achieved by method overriding which is also known as dynamic binding or late binding.

1) Static or Compile time Polymorphism

The mechanism of linking a function with an object during compile time is called early binding. It is also called static binding. C# provides two techniques to implement static polymorphism. They are –

- Function overloading
- Operator overloading

Example:-

```
class SimpleCalculator
{
   int add(int a, int b)
   {
      return a+b;
   }
   int  add(int a, int b, int c)
   {
      return a+b+c;
   }
}
public class Demo
{
  public static void main(String args[])
  {
            SimpleCalculator obj = new SimpleCalculator();
    System.out.println(obj.add(10, 20));
    System.out.println(obj.add(10, 20, 30));
```

```
  }
}
```

Output:

```
30
60
```

2)Runtime or Dynamic Polymorphism:

It is also known as Dynamic Method Dispatch. Dynamic polymorphism is a process in which a call to an overridden method is resolved at runtime, that's why it is called runtime polymorphism. C# provides this type of technique t implement runtime polymorphism.

- Method overriding

Example: -

In this example we have two classes ABC and XYZ. ABC is a parent class and XYZ is a child class. The child class is overriding the method myMethod() of parent class. In this example we have child class object assigned to the parent class reference so in order to determine which method would be called, the type of the object would be determined at run-time. It is the type of object that determines which version of the method would be called (not the type of reference).

```
class ABC
{
  public void myMethod(){
        System.out.println("Overridden Method");
  }
}
public class XYZ extends ABC{

  public void myMethod(){
        System.out.println("Overriding Method");
  }
  public static void main(String args[]){
        ABC obj = new XYZ();
        obj.myMethod();
  }
}
```

Output:

Overriding Method

2.1.4) INTERFACES

- Interface in C# is a blueprint of a class. It is like abstract class because all the methods which are declared inside the interface are abstract methods. It cannot have method body and cannot be instantiated.
- It is used *to achieve multiple inheritance* which can't be achieved by class. It is used *to achieve fully abstraction* because it cannot have method body.
- Its implementation must be provided by class or struct. The class or struct which implements the interface, must provide the implementation of all the methods declared inside the interface.
- Interfaces define properties, methods, and events, which are the members of the interface. Interfaces contain only the declaration of the members. It is the responsibility of the deriving class to define the members. It often helps in providing a standard structure that the deriving classes would follow.
- Abstract classes to some extent serve the same purpose, however, they are mostly used when only few methods are to be declared by the base class and the deriving class implements the functionalities.

Declaring Interfaces

Interfaces are declared using the interface keyword. It is similar to class declaration. Interface statements are public by default. Following is an example of an interface declaration –

```
public interface ITransactions
{
   // interface members
   void showTransaction();
   double getAmount();
}
```

C# interface example

Let's see the example of interface in C# which has draw() method. Its implementation is provided by two classes: Rectangle and Circle.

```
using System;
public interface Drawable
{
  void draw();
}
public class Rectangle : Drawable
{
  public void draw()
  {
    Console.WriteLine("drawing rectangle...");
  }
}
```

```
public class Circle : Drawable
{
   public void draw()
   {
      Console.WriteLine("drawing circle...");
   }
}
public class TestInterface
{
   public static void Main()
   {
      Drawable d;
      d = new Rectangle();
      d.draw();
      d = new Circle();
      d.draw();
   }
}
```

Output:

```
drawing rectangle...
drawing circle...
```

Note: Interface methods are public and abstract by default. You cannot explicitly use public and abstract keywords for an interface method.

```
using System;
public interface Drawable
{
   public abstract void draw();//Compile Time Error
}
```

2.1.5) ABSTRACT CLASSES: -

Abstract classes are the way to achieve abstraction in C#. Abstraction in C# is the process to hide the internal details and showing functionality only. Abstraction can be achieved by two ways:

1. Abstract class
2. Interface

Abstract class and interface both can have abstract methods which are necessary for abstraction.

Abstract Method

A method which is declared abstract and has no body is called abstract method. It can be declared inside the abstract class only. Its implementation must be provided by derived classes. For example:

1. **public abstract void** draw();

An abstract method in C# is internally a virtual method so it can be overridden by the derived class.

You can't use static and virtual modifiers in abstract method declaration.

C# Abstract class

In C#, abstract class is a class which is declared abstract. It can have abstract and non-abstract methods. It cannot be instantiated. Its implementation must be provided by derived classes. Here, derived class is forced to provide the implementation of all the abstract methods.

Let's see an example of abstract class in C# which has one abstract method draw(). Its implementation is provided by derived classes: Rectangle and Circle. Both classes have different implementation.

```
using System;
public abstract class Shape
{
  public abstract void draw();
}
public class Rectangle : Shape
{
  public override void draw()
  {
    Console.WriteLine("drawing rectangle...");
  }
}
public class Circle : Shape
{
  public override void draw()
  {
    Console.WriteLine("drawing circle...");
  }
}
public class TestAbstract
{
  public static void Main()
  {
    Shape s;
    s = new Rectangle();
    s.draw();
    s = new Circle();
```

```
      s.draw();
   }
}
```

Output:

```
drawing rectangle...
drawing circle...
```

2.1.6) DELEGATES

A **delegate** is a reference type variable that holds the reference to a method. The reference can be changed at runtime.

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

Declaring Delegates

Delegate declaration determines the methods that can be referenced by the delegate. A delegate can refer to a method, which has the same signature as that of the delegate.

For example, consider a delegate –

```
public delegate int MyDelegate (string s);
```

The preceding delegate can be used to reference any method that has a single *string* parameter and returns an *int* type variable.

Syntax for delegate declaration is –

```
delegate <return type> <delegate-name> <parameter list>
```

Instantiating Delegates

Once a delegate type is declared, a delegate object must be created with the **new** keyword and be associated with a particular method. When creating a delegate, the argument passed to the **new** expression is written similar to a method call, but without the arguments to the method. For example –

```
public delegate void printString(string s);
...
printString ps1 = new printString(WriteToScreen);
printString ps2 = new printString(WriteToFile);
```

C# Delegate Example

Let's see a simple example of delegate in C# which calls add() and mul() methods.

```
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
```

2.1.7) INDEXERS

An **indexer** allows an object to be indexed such as an array. When you define an indexer for a class, this class behaves similar to a **virtual array**. You can then access the instance of this class using the array access operator ([ ]).

Syntax

A one dimensional indexer has the following syntax –

```
element-type this[int index] {

  // The get accessor.
  get {
    // return the value specified by index
  }

  // The set accessor.
  set {
    // set the value specified by index
  }
}
```

Use of Indexers

Declaration of behavior of an indexer is to some extent similar to a property. similar to the properties, you use **get** and **set** accessors for defining an indexer. However, properties return or set a specific data member, whereas indexers returns or sets a particular value from the object instance. In other words, it breaks the instance data into smaller parts and indexes each part, gets or sets each part.

Defining a property involves providing a property name. Indexers are not defined with names, but with the **this** keyword, which refers to the object instance. The following example demonstrates the concept –

```
using System;

namespace IndexerApplication {

   class IndexedNames {
      private string[] namelist = new string[size];
      static public int size = 10;

      public IndexedNames() {
         for (int i = 0; i < size; i++)
         namelist[i] = "N. A.";
      }
```

```
      public string this[int index] {
         get {
            string tmp;

            if( index >= 0 && index <= size-1 ) {
               tmp = namelist[index];
            } else {
               tmp = "";
            }

            return ( tmp );
         }
         set {
            if( index >= 0 && index <= size-1 ) {
               namelist[index] = value;
            }
         }
      }
      static void Main(string[] args) {
         IndexedNames names = new IndexedNames();
         names[0] = "Zara";
         names[1] = "Riz";
         names[2] = "Nuha";
         names[3] = "Asif";
         names[4] = "Davinder";
         names[5] = "Sunil";
         names[6] = "Rubic";

         for ( int i = 0; i < IndexedNames.size; i++ ) {
            Console.WriteLine(names[i]);
         }
         Console.ReadKey();
      }
   }
}
```

When the above code is compiled and executed, it produces the following result –

```
Zara
Riz
Nuha
Asif
Davinder
Sunil
Rubic
N. A.
N. A.
N. A.
```

Overloaded Indexers

Indexers can be overloaded. Indexers can also be declared with multiple parameters and each parameter may be a different type. It is not

necessary that the indexes have to be integers. C# allows indexes to be of other types, for example, a string.

The following example demonstrates overloaded indexers –

```
using System;

namespace IndexerApplication {
   class IndexedNames {
      private string[] namelist = new string[size];
      static public int size = 10;

      public IndexedNames() {
         for (int i = 0; i < size; i++) {
            namelist[i] = "N. A.";
         }
      }
      public string this[int index] {
         get {
            string tmp;

            if( index >= 0 && index <= size-1 ) {
               tmp = namelist[index];
            } else {
               tmp = "";
            }

            return ( tmp );
         }
         set {
            if( index >= 0 && index <= size-1 ) {
               namelist[index] = value;
            }
         }
      }

      public int this[string name] {
         get {
            int index = 0;

            while(index < size) {
               if (namelist[index] == name) {
                return index;
               }
               index++;
            }
            return index;
         }
      }

      static void Main(string[] args) {
         IndexedNames names = new IndexedNames();
         names[0] = "Zara";
```

```
            names[1] = "Riz";
            names[2] = "Nuha";
            names[3] = "Asif";
            names[4] = "Davinder";
            names[5] = "Sunil";
            names[6] = "Rubic";

            //using the first indexer with int parameter
            for (int i = 0; i < IndexedNames.size; i++) {
               Console.WriteLine(names[i]);
            }

            //using the second indexer with the string parameter
            Console.WriteLine(names["Nuha"]);
            Console.ReadKey();
         }
      }
}
```

When the above code is compiled and executed, it produces the following result –

```
Zara
Riz
Nuha
Asif
Davinder
Sunil
Rubic
N. A.
N. A.
N. A.
2
```

2.1.8) SEALED CLASSES

C# sealed keyword applies restrictions on the class and method. If you create a sealed class, it cannot be derived. If you create a sealed method, it cannot be overridden.

Note: Structs are implicitly sealed therefore they can't be inherited.

C# Sealed class

C# sealed class cannot be derived by any class. Let's see an example of sealed class in C#.

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

```
public class Dog: Animal
{
   public void bark() { Console.WriteLine("barking..."); }
}
public class TestSealed
{
   public static void Main()
   {
      Dog d = new Dog();
      d.eat();
      d.bark();


   }
}
```

Output:

```
Compile Time Error: 'Dog': cannot derive from sealed type 'Animal'
```

C# Sealed method

The sealed method in C# cannot be overridden further. It must be used with override keyword in method.

Let's see an example of sealed method in C#.

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

}
public class Dog: Animal
{
   public override void eat() { Console.WriteLine("eating bread..."); }
   public sealed override void run() {
   Console.WriteLine("running very fast...");
   }
}
public class BabyDog : Dog
{
   public override void eat() { Console.WriteLine("eating biscuits..."); }
   public override void run() { Console.WriteLine("running slowly..."); }
}
public class TestSealed
{
   public static void Main()
   {
      BabyDog d = new BabyDog();
      d.eat();
```

```
    d.run();
  }
}
```

Output:

```
Compile Time Error: 'BabyDog.run()': cannot override inherited member
'Dog.run()' because it is sealed
```

Note: Local variables can't be sealed.

```
using System;
public class TestSealed
{
  public static void Main()
  {
    sealed int x = 10;
    x++;
    Console.WriteLine(x);
  }
}
```

Output:

```
Compile Time Error: Invalid expression term 'sealed'
```

2.1.9) EXCEPTION HANDLING

An exception is a problem that arises during the execution of a program. A C# exception is a response to an exceptional circumstance that arises while a program is running, such as an attempt to divide by zero.

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 identifies a block of code for which particular exceptions is activated. It is followed by one or more catch blocks.
- **catch** – A program catches an exception with an exception handler at the place in a program where you want to handle the problem. The catch keyword indicates the catching of an exception.
- **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 program throws an exception when a problem shows up. This is done using a throw keyword.

Syntax

Assuming a block raises an exception, a method catches an exception using a combination of the try and catch keywords. A try/catch block is placed around the code that might generate an exception. Code within a try/catch block is referred to as protected code, and the syntax for using try/catch looks like the following –

```
try {
   // statements causing exception
} catch( ExceptionName e1 ) {
   // error handling code
} catch( ExceptionName e2 ) {
   // error handling code
} catch( ExceptionName eN ) {
   // error handling code
} finally {
   // statements to be executed
}
```

You can list down multiple catch statements to catch different type of exceptions in case your try block raises more than one exception in different situations.

Exception Classes in C#

- C# exceptions are represented by classes. The exception classes in C# are mainly directly or indirectly derived from the **System.Exception** class. Some of the exception classes derived from the System.Exception class are the **System.ApplicationException** and **System.SystemException** classes.
- The **System.ApplicationException** class supports exceptions generated by application programs. Hence the exceptions defined by the programmers should derive from this class.
- The **System.SystemException** class is the base class for all predefined system exception.

The following table provides some of the predefined exception classes derived from the Sytem.SystemException class –

Sr.No.	Exception Class & Description
1	System.IO.IOException

	Handles I/O errors.
2	System.IndexOutOfRangeException Handles errors generated when a method refers to an array index out of range.
3	System.ArrayTypeMismatchException Handles errors generated when type is mismatched with the array type.
4	System.NullReferenceException Handles errors generated from referencing a null object.
5	System.DivideByZeroException Handles errors generated from dividing a dividend with zero.
6	System.InvalidCastException Handles errors generated during typecasting.
7	System.OutOfMemoryException Handles errors generated from insufficient free memory.
8	System.StackOverflowException Handles errors generated from stack overflow.

Handling Exceptions

C# provides a structured solution to the exception handling in the form of try and catch blocks. Using these blocks the core program statements are separated from the error-handling statements.

These error handling blocks are implemented using the **try**, **catch**, and **finally** keywords. Following is an example of throwing an exception when dividing by zero condition occurs –

```
using System;

namespace ErrorHandlingApplication {
   class DivNumbers {
      int result;

      DivNumbers() {
         result = 0;
      }
      public void division(int num1, int num2) {
         try {
            result = num1 / num2;
         } catch (DivideByZeroException e) {
            Console.WriteLine("Exception caught: {0}", e);
         } finally {
            Console.WriteLine("Result: {0}", result);
         }
      }
```

```
      static void Main(string[] args) {
         DivNumbers d = new DivNumbers();
         d.division(25, 0);
         Console.ReadKey();
      }
   }
}
```

When the above code is compiled and executed, it produces the following result –

```
Exception caught: System.DivideByZeroException: Attempted to divide by zero.
at ...
Result: 0
```

Creating User-Defined Exceptions

You can also define your own exception. User-defined exception classes are derived from the **Exception** class. The following example demonstrates this –

Live Demo

```
using System;

namespace UserDefinedException {
   class TestTemperature {
      static void Main(string[] args) {
         Temperature temp = new Temperature();
         try {
            temp.showTemp();
         } catch(TempIsZeroException e) {
            Console.WriteLine("TempIsZeroException: {0}", e.Message);
         }
         Console.ReadKey();
      }
   }
}
public class TempIsZeroException: Exception {
   public TempIsZeroException(string message): base(message) {
   }
}
public class Temperature {
   int temperature = 0;

   public void showTemp() {

      if(temperature == 0) {
         throw (new TempIsZeroException("Zero Temperature found"));
      } else {
         Console.WriteLine("Temperature: {0}", temperature);
      }
   }
}
```

When the above code is compiled and executed, it produces the following result –

```
TempIsZeroException: Zero Temperature found
```

Throwing Objects

You can throw an object if it is either directly or indirectly derived from the **System.Exception** class. You can use a throw statement in the catch block to throw the present object as –

```
Catch(Exception e) {
   ...
   Throw e
}
```

C# try/catch

In C# programming, exception handling is performed by try/catch statement. The **try block** in C# is used to place the code that may throw exception. The **catch block** is used to handled the exception. The catch block must be preceded by try block.

C# example without try/catch

```
using System;
public class ExExample
{
  public static void Main(string[] args)
  {
    int a = 10;
    int b = 0;
    int x = a/b;
    Console.WriteLine("Rest of the code");
  }
}
```

Output:

```
Unhandled Exception: System.DivideByZeroException: Attempted to divide by zero.
```

C# try/catch example

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

```
    try
    {
      int a = 10;
      int b = 0;
      int x = a / b;
    }
    catch (Exception e) { Console.WriteLine(e); }

    Console.WriteLine("Rest of the code");
  }
}
```

Output:

```
System.DivideByZeroException: Attempted to divide by zero.
Rest of the code
```

C# finally

C# finally block is used to execute important code which is to be executed whether exception is handled or not. It must be preceded by catch or try block.

C# finally example if exception is handled

```
using System;
public class ExExample
{
  public static void Main(string[] args)
  {
    try
    {
      int a = 10;
      int b = 0;
      int x = a / b;
    }
    catch (Exception e) { Console.WriteLine(e); }
    finally { Console.WriteLine("Finally block is executed"); }
    Console.WriteLine("Rest of the code");
  }
}
```

Output:

```
System.DivideByZeroException: Attempted to divide by zero.
Finally block is executed
Rest of the code
```

C# finally example if exception is not handled

```
using System;
public class ExExample
{
  public static void Main(string[] args)
  {
    try
    {
      int a = 10;
      int b = 0;
      int x = a / b;
    }
    catch (NullReferenceException e) { Console.WriteLine(e); }
    finally { Console.WriteLine("Finally block is executed"); }
    Console.WriteLine("Rest of the code");
  }
}
```

Output:

```
Unhandled Exception: System.DivideBy
```

3.1) COLLECTION AND GENERICS: -

3.1.1)GENERICS

Generics allow you to delay the specification of the data type of programming elements in a class or a method, until it is actually used in the program. In other words, generics allow you to write a class or method that can work with any data type.

You write the specifications for the class or the method, with substitute parameters for data types. When the compiler encounters a constructor for the class or a function call for the method, it generates code to handle the specific data type. A simple example would help understanding the concept –

```
using System;
using System.Collections.Generic;

namespace GenericApplication {
   public class MyGenericArray<T> {
      private T[] array;

      public MyGenericArray(int size) {
         array = new T[size + 1];
```

```
      }
      public T getItem(int index) {
         return array[index];
      }
      public void setItem(int index, T value) {
         array[index] = value;
      }
   }
   class Tester {
      static void Main(string[] args) {

         //declaring an int array
         MyGenericArray<int> intArray = new MyGenericArray<int>(5);

         //setting values
         for (int c = 0; c < 5; c++) {
            intArray.setItem(c, c*5);
         }

         //retrieving the values
         for (int c = 0; c < 5; c++) {
            Console.Write(intArray.getItem(c) + " ");
         }

         Console.WriteLine();

         //declaring a character array
         MyGenericArray<char> charArray = new MyGenericArray<char>(5);

         //setting values
         for (int c = 0; c < 5; c++) {
            charArray.setItem(c, (char)(c+97));
         }

         //retrieving the values
         for (int c = 0; c< 5; c++) {
            Console.Write(charArray.getItem(c) + " ");
         }
         Console.WriteLine();

         Console.ReadKey();
      }
   }
}
```

When the above code is compiled and executed, it produces the following result –

```
0 5 10 15 20
a b c d e
```

Features of Generics

Generics is a technique that enriches your programs in the following ways –

- It helps you to maximize code reuse, type safety, and performance.
- You can create generic collection classes. The .NET Framework class library contains several new generic collection classes in the *System.Collections.Generic* namespace. You may use these generic collection classes instead of the collection classes in the *System.Collections* namespace.
- You can create your own generic interfaces, classes, methods, events, and delegates.
- You may create generic classes constrained to enable access to methods on particular data types.
- You may get information on the types used in a generic data type at run-time by means of reflection.

Generic Methods

In the previous example, we have used a generic class; we can declare a generic method with a type parameter. The following program illustrates the concept –

```
using System;
using System.Collections.Generic;

namespace GenericMethodAppl {
   class Program {
      static void Swap<T>(ref T lhs, ref T rhs) {
         T temp;
         temp = lhs;
         lhs = rhs;
         rhs = temp;
      }
      static void Main(string[] args) {
         int a, b;
         char c, d;
         a = 10;
         b = 20;
         c = 'I';
         d = 'V';

         //display values before swap:
         Console.WriteLine("Int values before calling swap:");
         Console.WriteLine("a = {0}, b = {1}", a, b);
```

```
            Console.WriteLine("Char values before calling swap:");
            Console.WriteLine("c = {0}, d = {1}", c, d);

            //call swap
            Swap<int>(ref a, ref b);
            Swap<char>(ref c, ref d);

            //display values after swap:
            Console.WriteLine("Int values after calling swap:");
            Console.WriteLine("a = {0}, b = {1}", a, b);
            Console.WriteLine("Char values after calling swap:");
            Console.WriteLine("c = {0}, d = {1}", c, d);

            Console.ReadKey();
        }
    }
}
```

When the above code is compiled and executed, it produces the following result –

```
Int values before calling swap:
a = 10, b = 20
Char values before calling swap:
c = I, d = V
Int values after calling swap:
a = 20, b = 10
Char values after calling swap:
c = V, d = I
```

Generic Delegates

You can define a generic delegate with type parameters. For example –

```
delegate T NumberChanger<T>(T n);
```

The following example shows use of this delegate –

Live Demo

```
using System;
using System.Collections.Generic;

delegate T NumberChanger<T>(T n);
namespace GenericDelegateAppl {
   class TestDelegate {
      static int num = 10;

      public static int AddNum(int p) {
         num += p;
         return num;
      }
      public static int MultNum(int q) {
         num *= q;
         return num;
      }
```

```
      public static int getNum() {
         return num;
      }
      static void Main(string[] args) {
         //create delegate instances
         NumberChanger<int> nc1 = new NumberChanger<int>(AddNum);
         NumberChanger<int> nc2 = new NumberChanger<int>(MultNum);

         //calling the methods using the delegate objects
         nc1(25);
         Console.WriteLine("Value of Num: {0}", getNum());

         nc2(5);
         Console.WriteLine("Value of Num: {0}", getNum());
         Console.ReadKey();
      }
   }
}
```

When the above code is compiled and executed, it produces the following result –

```
Value of Num: 35
Value of Num: 175
```

Generic simple examples:-

C# Generic class example

```
using System;
namespace CSharpProgram
{
   class GenericClass<T>
   {
      public GenericClass(T msg)
      {
         Console.WriteLine(msg);
      }
   }
   class Program
   {
      static void Main(string[] args)
      {
         GenericClass<string> gen  = new GenericClass<string> ("This is generic class");
         GenericClass<int>   genI  = new GenericClass<int>(101);
         GenericClass<char>  getCh = new GenericClass<char>('I');
      }
   }
```

}

Output:

```
This is generic class
101
I
```

C# allows us to create generic methods also. In the following example, we are creating generic method that can be called by passing any type of argument.

Generic Method Example

```
using System;
namespace CSharpProgram
{
  class GenericClass
  {
    public void Show<T>(T msg)
    {
      Console.WriteLine(msg);
    }
  }
  class Program
  {
    static void Main(string[] args)
    {
      GenericClass genC = new GenericClass();
      genC.Show("This is generic method");
      genC.Show(101);
      genC.Show('I');
    }
  }
}
```

Output:

```
This is generic method
101
I
```

3.1.2) COLLECTION

- Collection classes are specialized classes for data storage and retrieval. These classes provide support for stacks, queues, lists, and hash tables. Most collection classes implement the same interfaces.
- Collection classes serve various purposes, such as allocating memory dynamically to elements and accessing a list of items on the basis of

an index etc. These classes create collections of objects of the Object class, which is the base class for all data types in C#.

In C#, collection represents group of objects. By the help of collections, we can perform various operations on objects such as

- store object
- update object
- delete object
- retrieve object
- search object, and
- sort object

In sort, all the data structure work can be performed by C# collections.

We can store objects in array or collection. Collection has advantage over array. Array has size limit but objects stored in collection can grow or shrink dynamically.

Types of Collections in C#

There are 3 ways to work with collections. The three namespaces are given below:

- **System.Collections.Generic** classes
- **System.Collections** classes (Now deprecated)
- **System.Collections.Concurrent** classes

1) System.Collections.Generic classes

The System.Collections.Generic namespace has following classes:

- List
- Stack
- Queue
- LinkedList
- HashSet
- SortedSet
- Dictionary
- SortedDictionary
- SortedList

2) System.Collections classes

These classes are legacy. It is suggested now to use System.Collections.Generic classes. The System.Collections namespace has following classes:

- ArrayList

- Stack
- Queue
- Hashtable

3) System.Collections.Concurrent classes

The System.Collections.Concurrent namespace provides classes for thread-safe operations. Now multiple threads will not create problem for accessing the collection items.

The System.Collections.Concurrent namespace has following classes:

- BlockingCollection
- ConcurrentBag
- ConcurrentStack
- ConcurrentQueue
- ConcurrentDictionary
- Partitioner
- Partitioner
- OrderablePartitioner
-

Various Collection Classes and Their Usage

The following are the various commonly used classes of the **System.Collection** namespace. Click the following links to check their detail.

Sr.No.	Class & Description and Useage
1	**ArrayList** It represents ordered collection of an object that can be indexedindividually. It is basically an alternative to an array. However, unlike array you can add and remove items from a list at a specified position using an index and the array resizes itself automatically. It also allows dynamic memory allocation, adding, searching and sorting items in the list.
2	**Hashtable** It uses a key to access the elements in the collection. A hash table is used when you need to access elements by using key, and you can identify a useful key value. Each item in the hash table has a key/value pair. The key is used to access the items in the collection.
3	**SortedList** It uses a key as well as an index to access the items in a list. A sorted list is a combination of an array and a hash table. It contains a list of items that can be accessed using a key or an index. If you access items using an index, it is an ArrayList, and if you access items using a key , it is a Hashtable. The collection of items is always sorted by the key value.
4	**Stack** It represents a last-in, first out collection of object.

	It is used when you need a last-in, first-out access of items. When you add an item in the list, it is called pushing the item and when you remove it, it is called popping the item.
5	**Queue** It represents a first-in, first out collection of object. It is used when you need a first-in, first-out access of items. When you add an item in the list, it is called enqueue and when you remove an item, it is called deque.
6	**BitArray** It represents an array of the binary representation using the values 1 and 0. It is used when you need to store the bits but do not know the number of bits in advance. You can access items from the BitArray collection by using an integer index, which starts from zero.

3.1.3)Constraints on Generic Programming

Generics support the ability to define constraints on type parameters. These constraints enforce the types to conform to various rules. Take, for example, the `BinaryTree<T>` class shown in Listing 11.18.

Example 11.18. Declaring a `BinaryTree<T>` Class with No Constraints

```
public  class BinaryTree<T>
{
   public BinaryTree ( T item)
   {
      Item = item;

   }

   public T Item
   {
      get{ return _Item; }
      set{ _Item = value; }
   }
   private T _Item;

   public Pair<BinaryTree<T>> SubItems
   {
      get{ return _SubItems; }
      set{ _SubItems = value; }
    }
    private Pair<BinaryTree<T>> _SubItems;
}
```

(An interesting side note is that `BinaryTree<T>` uses `Pair<T>` internally, which is possible because `Pair<T>` is simply another type.)

UNIT 3
DATABASES AND C#

FILE HANDLING
Text Files, Binary Files, String Processing, Serialization and Deserialization

---

1.1)File Handling

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).

1.1.1)TEXTFILES

C# TextWriter

C# TextWriter class is an abstract class. It is used to write text or sequential series of characters into file. It is found in System.IO namespace.

C# TextWriter Example

Let's see the simple example of TextWriter class to write two lines data.

```
using System;
using System.IO;
namespace TextWriterExample
{
   class Program
   {
      static void Main(string[] args)
      {
         using (TextWriter writer = File.CreateText("e:\\f.txt"))
         {
```

```
            writer.WriteLine("Hello C#");
            writer.WriteLine("C# File Handling by JavaTpoint");
        }
        Console.WriteLine("Data written successfully...");
    }
  }
}
```

C# TextReader

C# TextReader class is found in System.IO namespace. It represents a reader that can be used to read text or sequential series of characters.

C# TextReader Example: Read All Data

Let's see the simple example of TextReader class that reads data till the end of file.

```
using System;
using System.IO;
namespace TextReaderExample
{
  class Program
  {
    static void Main(string[] args)
    {
      using (TextReader tr = File.OpenText("e:\\f.txt"))
      {
        Console.WriteLine(tr.ReadToEnd());
      }
    }
  }
}
```

1.1.2) BINARY FILES

C# BinaryWriter

C# BinaryWriter class is used to write binary information into stream. It is found in System.IO namespace. It also supports writing string in specific encoding.

C# BinaryWriter Example

Let's see the simple example of BinaryWriter class which writes data into dat file.

```
using System;
using System.IO;
namespace BinaryWriterExample
{
```

```
   class Program
   {
      static void Main(string[] args)
      {
         string fileName = "e:\\binaryfile.dat";
         using (BinaryWriter writer = new BinaryWriter(File.Open(fileName, FileMode.Create)))
         {
            writer.Write(2.5);
            writer.Write("this is string data");
            writer.Write(true);
         }
         Console.WriteLine("Data written successfully...");
      }
   }
}
```

C# BinaryReader

C# BinaryReader class is used to read binary information from stream. It is found in System.IO namespace. It also supports reading string in specific encoding.

C# BinaryReader Example

Let's see the simple example of BinaryReader class which reads data from dat file.

```
using System;
using System.IO;
namespace BinaryWriterExample
{
   class Program
   {
      static void Main(string[] args)
      {
         WriteBinaryFile();
         ReadBinaryFile();
         Console.ReadKey();
      }
      static void WriteBinaryFile()
      {
         using (BinaryWriter writer = new BinaryWriter(File.Open("e:\\binaryfile.dat", FileMode.Creat
e)))
         {

            writer.Write(12.5);
            writer.Write("this is string data");
            writer.Write(true);
         }
      }
      static void ReadBinaryFile()
```

```
    {
      using (BinaryReader reader = new BinaryReader(File.Open("e:\\binaryfile.dat", FileMode.Ope
n)))
      {
        Console.WriteLine("Double Value : " + reader.ReadDouble());
        Console.WriteLine("String Value : " + reader.ReadString());
        Console.WriteLine("Boolean Value : " + reader.ReadBoolean());
      }
    }
  }
}
```

1.1.3)STRING PROCESSING

C# StringWriter Class

This class is used to write and deal with string data rather than files. It is derived class of TextWriter class. The string data written by StringWriter class is stored into StringBuilder.

The purpose of this class is to manipulate string and save result into the StringBuilder.

StringWriter Class Signature

1. [SerializableAttribute]
2. [ComVisibleAttribute(**true**)]
3. **public class** StringWriter : TextWriter

C# StringWriter Constructors

C# StringWriter Constructors

Constructors	Description
StringWriter()	It is used to initialize a new instance of the StringWriter class.
StringWriter(IFormatProvider)	It is used to initialize a new instance of the StringWriter class with the specified format control.
StringWriter(StringBuilder)	It is used to initialize a new instance of the StringWriter class that writes to the specified StringBuilder.
StringWriter(StringBuilder,? IFormatProvider)	It is used to initialize a new instance of the StringWriter class that writes to the specified StringBuilder and has the specified format provider.

C# StringWriter Properties

Property	Description
Encoding	It is used to get the Encoding in which the output is written.
FormatProvider	It is used to get an object that controls formatting.
NewLine	It is used to get or set the line terminator string used by the current **TextWriter.**

C# StringWriter Methods

Methods	Description
Close()	It is used to close the current StringWriter and the underlying stream.
Dispose()	It is used to release all resources used by the TextWriter object.
Equals(Object)	It is used to determine whether the specified object is equal to the current object or not.
Finalize()	It allows an object to try to free resources and perform other cleanup operations.
GetHashCode()	It is used to serve as the default hash function.
GetStringBuilder()	It returns the underlying StringBuilder.
ToString()	It returns a string containing the characters written to the current StringWriter.
WriteAsync(String)	It is used to write a string to the current string asynchronously.
Write(Boolean)	It is used to write the text representation of a Boolean value to the string.
Write(String)	It is used to write a string to the current string.
WriteLine(String)	It is used to write a string followed by a line terminator to the string or stream.

C# StringWriter Example

In the following program, we are using StringWriter class to write string information to the StringBuilder class. The StringReader class is used to read written information to the StringBuilder.

```
using System;
using System.IO;
using System.Text;
namespace CSharpProgram
{
   class Program
   {
      static void Main(string[] args)
      {
```

```
        string text = "Hello, Welcome to the javatpoint \n" +
           "It is nice site. \n" +
           "It provides technical tutorials";
        // Creating StringBuilder instance
        StringBuilder sb = new StringBuilder();
        // Passing StringBuilder instance into StringWriter
        StringWriter writer = new StringWriter(sb);
        // Writing data using StringWriter
        writer.WriteLine(text);
        writer.Flush();
        // Closing writer connection
        writer.Close();
        // Creating StringReader instance and passing StringBuilder
        StringReader reader = new StringReader(sb.ToString());
        // Reading data
        while (reader.Peek() > -1)
        {
           Console.WriteLine(reader.ReadLine());
        }
     }
  }
}
```

C# StringReader Class

StringReader class is used to read data written by the StringWriter class. It is subclass of TextReader class. It enables us to read a string synchronously or asynchronously. It provides constructors and methods to perform read operations.

C# StringReader Signature

1. [SerializableAttribute]
2. [ComVisibleAttribute(**true**)]
3. **public class** StringReader : TextReader

C# StringReader Constructors

StringReader has the following constructors.

Constructors	Description
StringReader(String)	Initializes a new instance of the StringReader class that reads from the specified string.

C# StringReader Methods

Following are the methods of StringReader class.

C# StringReader Methods

Following are the methods of StringReader class.

Method	Description
Close()	It is used to close the StringReader.
Dispose()	It is used to release all resources used by the TextReader object.
Equals(Object)	It determines whether the specified object is equal to the current object or not.
Finalize()	It allows an object to try to free resources and perform other cleanup operations.
GetHashCode()	It serves as the default hash function.
GetType()	It is used to get the type of the current instance.
Peek()	It is used to return the next available character but does not consume it.
Read()	It is used to read the next character from the input string.
ReadLine()	It is used to read a line of characters from the current string.
ReadLineAsync()	It is used to read a line of characters asynchronously from the current string.
ReadToEnd()	It is used to read all the characters from the current position to the end of the string.

C# StringReader Example

In the following example, StringWriter class is used to write the string information and StringReader class is used to read the string, written by the StringWriter class.

```
using System;
using System.IO;
namespace CSharpProgram
{
   class Program
   {
      static void Main(string[] args)
      {
         StringWriter str = new StringWriter();
         str.WriteLine("Hello, this message is read by StringReader class");
         str.Close();
         // Creating StringReader instance and passing StringWriter
         StringReader reader = new StringReader(str.ToString());
         // Reading data
         while (reader.Peek() > -1)
         {
            Console.WriteLine(reader.ReadLine());
         }
```

```
    }
  }
}
```

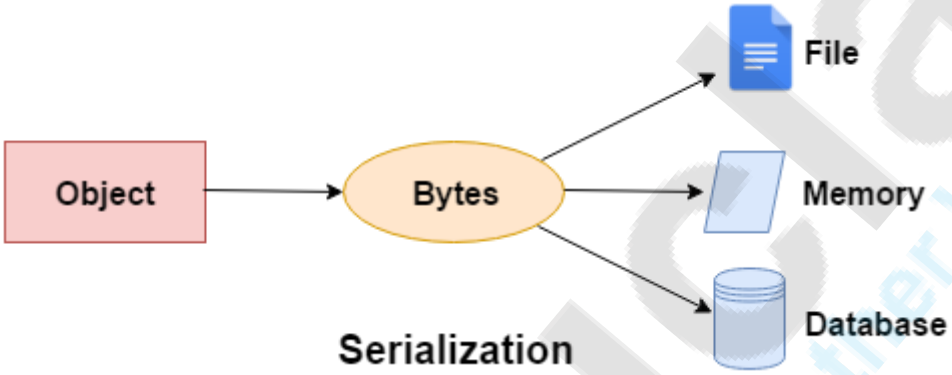
Output:

```
Hello, this message is read by StringReader class
```

1.1.4) C# Serialization

In C#, serialization is the process of converting object into byte stream so that it can be saved to memory, file or database. The reverse process of serialization is called deserialization.

Serialization is internally used in remote applications.



C# SerializableAttribute

To serialize the object, you need to apply *SerializableAttribute* attribute to the type. If you don't apply *SerializableAttribute* attribute to the type, *SerializationException* exception is thrown at runtime.

C# Serialization example

Let's see the simple example of serialization in C# where we are serializing the object of Student class. Here, we are going to use **BinaryFormatter.Serialize(stream, reference)** method to serialize the object.

```
using System;
using System.IO;
using System.Runtime.Serialization.Formatters.Binary;
[Serializable]
class Student
{
   int rollno;
   string name;
   public Student(int rollno, string name)
   {
```

```
        this.rollno = rollno;
        this.name = name;
    }
}
public class SerializeExample
{
    public static void Main(string[] args)
    {
        FileStream stream = new FileStream("e:\\sss.txt", FileMode.OpenOrCreate);
        BinaryFormatter formatter=new BinaryFormatter();

        Student s = new Student(101, "sonoo");
        formatter.Serialize(stream, s);

        stream.Close();
    }
}
```

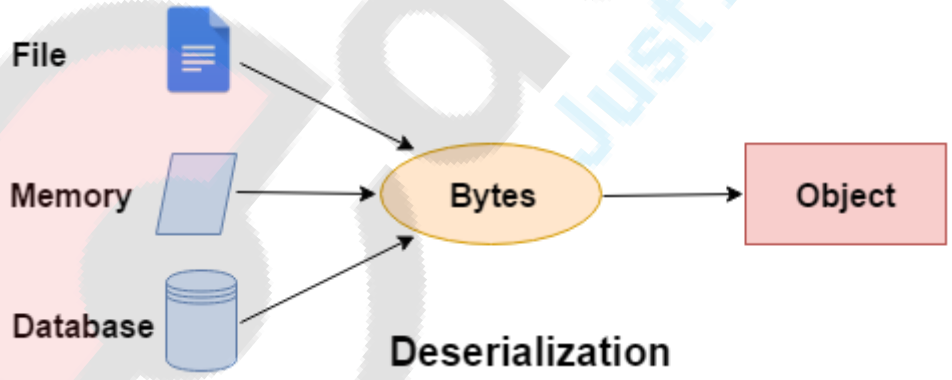
sss.txt:

```
  JConsoleApplication1,          Version=1.0.0.0,          Culture=neutral,
PublicKeyToken=null Student rollnoname e sonoo
```

As you can see, the serialized data is stored in the file. To get the data, you need to perform deserialization.

1.1.4) C# Deserialization

In C# programming, deserialization is the reverse process of serialization. It means you can read the object from byte stream. Here, we are going to use **BinaryFormatter.Deserialize(stream)** method to deserialize the stream.



C# Deserialization Example

Let's see the simple example of deserialization in C#.

```
using System;
using System.IO;
using System.Runtime.Serialization.Formatters.Binary;
```

```
[Serializable]
class Student
{
   public int rollno;
   public string name;
   public Student(int rollno, string name)
   {
      this.rollno = rollno;
      this.name = name;
   }
}
public class DeserializeExample
{
   public static void Main(string[] args)
   {
      FileStream stream = new FileStream("e:\\sss.txt", FileMode.OpenOrCreate);
      BinaryFormatter formatter=new BinaryFormatter();

      Student s=(Student)formatter.Deserialize(stream);
      Console.WriteLine("Rollno: " + s.rollno);
      Console.WriteLine("Name: " + s.name);

      stream.Close();
   }
}
```

Output:

```
Rollno: 101
Name: sonoo
```

C# StreamWriter

C# StreamWriter class is used to write characters to a stream in specific encoding. It inherits TextWriter class. It provides overloaded write() and writeln() methods to write data into file.

C# StreamWriter example

Let's see a simple example of StreamWriter class which writes a single line of data into the file.

```
using System;
using System.IO;
public class StreamWriterExample
{
   public static void Main(string[] args)
   {
      FileStream f = new FileStream("e:\\output.txt", FileMode.Create);
      StreamWriter s = new StreamWriter(f);
```

```
      s.WriteLine("hello c#");
      s.Close();
      f.Close();
    Console.WriteLine("File created successfully...");
   }
}
```

Output:

```
File created successfully...
```

Now open the file, you will see the text "hello c#" in output.txt file.

output.txt:

```
hello c#
```

C# StreamReader

C# StreamReader class is used to read string from the stream. It inherits TextReader class. It provides Read() and ReadLine() methods to read data from the stream.

C# StreamReader example to read one line

Let's see the simple example of StreamReader class that reads a single line of data from the file.

```
using System;
using System.IO;
public class StreamReaderExample
{
   public static void Main(string[] args)
   {
      FileStream f = new FileStream("e:\\output.txt", FileMode.OpenOrCreate);
      StreamReader s = new StreamReader(f);

      string line=s.ReadLine();
      Console.WriteLine(line);

      s.Close();
      f.Close();
   }
}
```

Output:

```
Hello C#
```

2.1)ADO.Net

Connected and Disconnected,Architecture of ADO.Net,Commands,Datasets,Data Readers, Data Adapters,Working with Stored Procedures

---

2.1)ADO.Net

Introduction

ADO.NET is an object-oriented set of libraries that allows you to interact with data sources. Commonly, the data source is a database, but it could also be a text file, an Excel spreadsheet, or an XML file. For the purposes of this tutorial, we will look at ADO.NET as a way to interact with a data base.

As you are probably aware, there are many different types of databases available. For example, there is Microsoft SQL Server, Microsoft Access, Oracle, Borland Interbase, and IBM DB2, just to name a few. To further refine the scope of this tutorial, all of the examples will use SQL Server.

Data Providers

We know that ADO.NET allows us to interact with different types of data sources and different types of databases. However, there isn't a single set of classes that allow you to accomplish this universally. Since different data sources expose different protocols, we need a way to communicate with the right data source using the right protocol Some older data sources use the ODBC protocol, many newer data sources use the OleDb protocol, and there are more data sources every day that allow you to communicate with them directly through .NET ADO.NET class libraries.

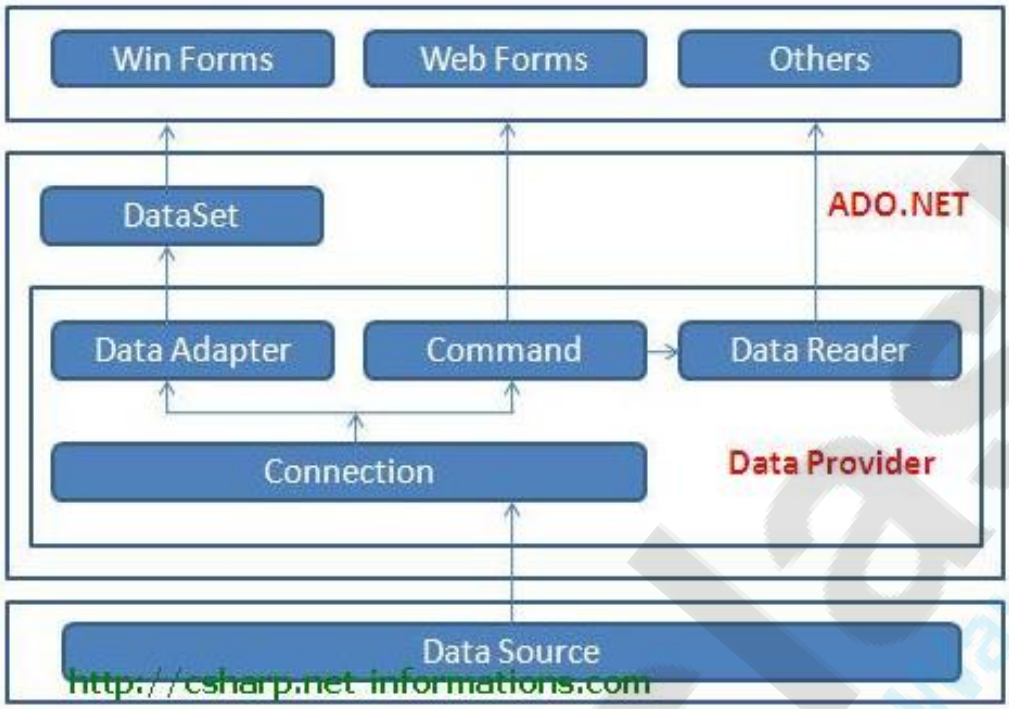
ADO.NET provides a relatively common way to interact with data sources, but comes in different sets of libraries for each way you can talk to a data source. These libraries are called Data Providers and are usually named for the protocol or data source type they allow you to interact with. Table 1 lists some well known data providers, the API prefix they use, and the type of data source they allow you to interact with.

Table 1. ADO.NET Data Providers are class libraries that allow a common way to interact with specific data sources or protocols. The library APIs have prefixes that indicate which provider they support.

Provider Name	API prefix	Data Source Description
ODBC Data Provider	Odbc	Data Sources with an ODBC interface. Normally older data bases.
OleDb Data Provider	OleDb	Data Sources that expose an OleDb interface, i.e. Access or Excel.
Oracle Data Provider	Oracle	For Oracle Databases.
SQL Data Provider	Sql	For interacting with Microsoft SQL Server.
Borland Data Provider	Bdp	Generic access to many databases such as Interbase, SQL Server, IBM DB2, and Oracle.

An example may help you to understand the meaning of the API prefix. One of the first ADO.NET objects you'll learn about is the connection object, which allows you to establish a connection to a data source. If we were using the OleDb Data Provider to connect to a data source that exposes an OleDb interface, we would use a connection object named OleDbConnection. Similarly, the connection object name would be prefixed with Odbc or Sql for an OdbcConnection object on an Odbc data source or a SqlConnection object on a SQL Server database, respectively. Since we are using MSDE in this tutorial (a scaled down version of SQL Server) all the API objects will have the Sql prefix. i.e. SqlConnection.

2.1.1)Architecture of ADO.Net



ADO.NET is a data access technology from Microsoft .Net Framework , which provides communication between relational and non-relational systems through a common set of components .**ADO.NET** consist of a set of Objects that expose data access services to the .NET environment. ADO.NET is designed to be easy to use, and Visual Studio provides several wizards and other features that you can use to generate **ADO.NET** data access code.

The architecture of ADO.net, in which connection must be opened to access the data retrieved from database is called as connected architecture. Connected architecture was built on the classes connection, command, datareader and transaction. Connected architecture is when you constantly make trips to the database for any CRUD (Create, Read, Update and Delete) operation you wish to do. This creates more traffic to the database but is normally much faster as you should be doing smaller transactions.

The architecture of ADO.net in which data retrieved from database can be accessed even when connection to database was closed is called as

disconnected architecture.Disconnected architecture of ADO.net was built on classes connection, dataadapter,commandbuilder and dataset and dataview.
Disconnected architecture is a method of retrieving a record set from the database and storing it giving you the ability to do many CRUD (Create, Read, Update and Delete) operations on the data in memory, then it can be re-synchronized with the database when reconnecting. A method of using disconnected architecture is using a Dataset

2.1.2) Connected and disconnected architecture in ADO.Net

Connected Architecture of ADO.NET

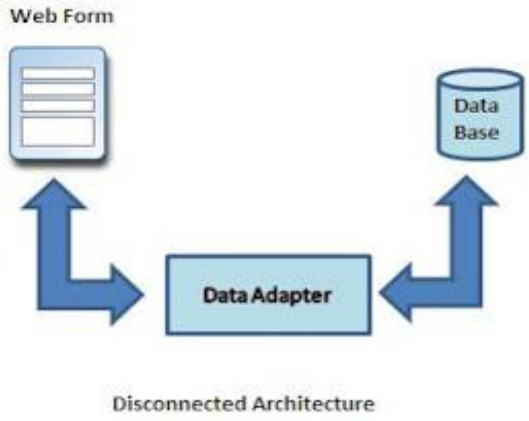
The architecture of ADO.net, in which connection must be opened to access the data retrieved from database is called as connected architecture. Connected architecture was built on the classes connection, command, data reader and transaction.

Connected architecture is when you constantly make trips to the database for any CRUD (Create, Read, Update and Delete) operation you wish to do. This creates more traffic to the database but is normally much faster as you should be doing smaller transactions.

Disconnected Architecture in ADO.NET

The architecture of ADO.net in which data retrieved from database can be accessed even when connection to database was closed is called as disconnected architecture. Disconnected architecture of ADO.net was built on classes connection, dataadapter, commandbuilder and dataset and dataview.
Disconnected architecture is a method of retrieving a record set from the database and storing it giving you the ability to do many CRUD (Create, Read, Update and Delete) operations on the data in memory, then it can

be re-synchronized with the database when reconnecting. A method of using disconnected architecture is using a Dataset.



Disconnected Architecture

DataReader is Connected Architecture since it keeps the connection open until all rows are fetched one by one
DataSet is **DisConnected** Architecture since all the records are brought at once and there is no need to keep the connection alive
Difference between Connected and disconnected architecture

Connected	Disconnected
It is connection oriented.	It is dis_connection oriented.
Datareader	DataSet
Connected methods gives faster performance	Disconnected get low in speed and performance.
connected can hold the data of single table	disconnected can hold multiple tables of data
connected you need to use a read only forward only data reader	disconnected you cannot
Data Reader can't persist the data	Data Set can persist the data
It is Read only, we can't update the data.	We can update data

Example

Create Database "Student"

```
CREATE TABLE [dbo].[Student]
(
   [ID] [int] PRIMARY KEY IDENTITY(1,1) NOT NULL,
   [Name] [varchar](255) NULL,
   [Age] [int] NULL,
   [Address] [varchar](255) NULL
)
```

```
INSERT INTO Student([Name],[Age],[Address])VALUES('NAME
1','22','PUNE')
INSERT INTO Student([Name],[Age],[Address])VALUES('NAME
2','25','MUMBAI')
INSERT INTO Student([Name],[Age],[Address])VALUES('NAME
3','23','PUNE')
INSERT INTO Student([Name],[Age],[Address])VALUES('NAME
4','21','DELHI')
INSERT INTO Student([Name],[Age],[Address])VALUES('NAME
5','22','PUNE')
```

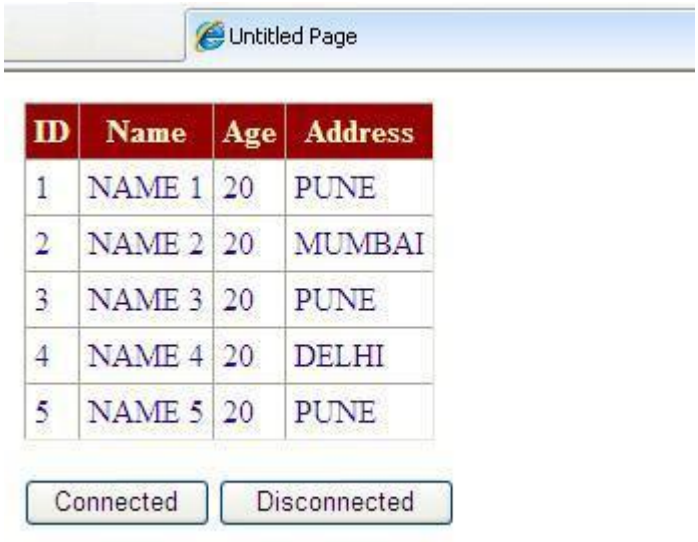
HTML

```
<html xmlns="http://www.w3.org/1999/xhtml">
<head id="Head1" runat="server">
   <title>Untitled Pagetitle>
</head>
<body>
   <form id="form1" runat="server">
   <div>
      <asp:GridView ID="GridView1" runat="server" BackColor="White"B
orderColor="#CC9966"
         BorderStyle="None" BorderWidth="1px" CellPadding="4">
         <FooterStyle BackColor="#FFFFCC" ForeColor="#330099" />
         <RowStyle BackColor="White" ForeColor="#330099" />
         <PagerStyle BackColor="#FFFFCC" ForeColor="#330099"Horizon
talAlign="Center" />
         <SelectedRowStyle BackColor="#FFCC66" Font-
Bold="True"ForeColor="#663399" />
         <HeaderStyle BackColor="#990000" Font-
Bold="True" ForeColor="#FFFFCC" />
      </asp:GridView>
      <br />
      <asp:Button ID="Connected" runat="server" OnClick="Connected_
Click"Text="Connected" />
      <asp:Button ID="Disconnected" runat="server" EnableTheming="Fa
lse"OnClick="Disconnected_Click"
         Text="Disconnected" />
   </div>
   </form>
</body>
</html>
```

Code Behind

```
String StrSQL = "", StrConnection = "";
      protected void Page_Load(object sender, EventArgs e)
      {
```

```
            StrSQL = "SELECT * FROM Student";
            StrConnection = "Data Source=ServerName;Initial
Catalog=Database;User ID=Username;Password=password";
        }

        protected void Connected_Click(object sender, EventArgs e)
        {
            using (SqlConnection objConn
= new SqlConnection(StrConnection))
            {
                SqlCommand objCmd = new SqlCommand(StrSQL, objConn);
                objCmd.CommandType = CommandType.Text;
                objConn.Open();
                SqlDataReader objDr = objCmd.ExecuteReader();
                GridView1.DataSource = objDr;
                GridView1.DataBind();
                objConn.Close();
            }
        }

        protected void Disconnected_Click(object sender, EventArgs e)
        {
            SqlDataAdapter objDa = new SqlDataAdapter();
            DataSet objDs = new DataSet();
            using (SqlConnection objConn
= new SqlConnection(StrConnection))
            {
                SqlCommand objCmd = new SqlCommand(StrSQL, objConn);
                objCmd.CommandType = CommandType.Text;
                objDa.SelectCommand = objCmd;
                objDa.Fill(objDs, "Student");
                GridView1.DataSource = objDs.Tables[0];
                GridView1.DataBind();
            }
        }
```



ADO.NET Objects

ADO.NET includes many objects you can use to work with data. This section introduces some of the primary objects you will use. Over the course of this tutorial, you'll be exposed to many more ADO.NET objects from the perspective of how they are used in a particular lesson. The objects below are the ones you must know. Learning about them will give you an idea of the types of things you can do with data when using ADO.NET.

1. Each Data Provider provides the following core objects:

   A. Connection

   B. Command

   C. Data Reader

   D. Data Adapter

   A. Connection:

   - This is the object that allows you to establish a connection with the data source.
   - Depending on the actual .NET Data Provider involved, connection objects automatically pool physical databases connections for you.
   - Examples of connection objects are OleDbConnection, SqlConnection, OracleConnection and so on.

Connection class used to establish the connection between front end and back end:

```
01. SqlConnection con=new SqlConnection("Integrated security=true;initial catalog=Student;Data s
```

Or

```
01. SqlConnection Con=new SqlConnection("User id=sa;Password=sa123;Database=Student;Server=.");
```

B. Command:

- This object represents an executable command on the underlying data source. The command may or may not return any results. These commands can be used to manipulate existing data.
- In addition, the commands can be used to manipulate underlying table structures.
- Examples of command objects are SqlCommand, OracleCommand and so on. A Command needs to be able to accept parameters. The Parameter object of ADO.NET allows commands to be more flexible and accept input values and act accordingly.

```
01. SqlCommand cmd=new SqlCommand("Query which has to perform",Connection Object);
```

C. Data Reader:

- This object is designed to help you retrieve and examine the rows returned by the query as quickly as possible.
- DataReader object examines the results of a query one row at a time. When you move forward to the next row, the contents of the previous row are discarded.
- The DataReader doesn't support updating. The data returned by the DataReader is read-only. Because the DataReader object supports such a minimal set of features, it is extremely fast and lightweight.
- The disadvantage of using a DataReader object is that it requires an open database connection and increases network activity.

```
SqlCommand cmd = new SqlCommand(SQL, conn);
// Call ExecuteReader to return a DataReader
SqlDataReader reader = cmd.ExecuteReader();
```

Once you're done with the data reader, call the Close method to close a data reader:

```
reader.Close();
```

D. DataAdapter:

- This object acts a gateway between the disconnected and connected flavours of ADO.NET. The architecture of ADO.NET, in which connection must be opened to access the data retrieved from the database is called as connected architecture whereas as in a disconnected architecture data retrieved from database can be accessed by holding it in a memory with the help of DataSet object even when the connection is closed.
- Examples of DataAdapters are SqlDataAdapter, OracleDataAdapter and so on. It has commands like Select, Insert, Update and Delete .
- Select command is used to retrieve data from the database and insert, update and delete commands are used to send changes to the data in dataset to database.

2.
```
// Create an OleDbDataAdapter object
OleDbDataAdapter adapter = new OleDbDataAdapter();
adapter.SelectCommand = new OleDbCommand (SQL, conn);
```

DataSet :

- It is an in-memory cache of data retrieved from the data source. Data is organized into multiple tables using DataTable objects, tables can be related using DataRelation objects and data integrity can be enforced using the constraint objects like UnqueConstraint and ForeignKeyConstraint.
- DataSet are also fully XML-featured. They contain methods such as GetXML and WriteXML that respectively produce and comsume XML data easily.
- The DataSet object provides a consistent programming model that works with all the current models of data storage: flat, relational and hierarchial.
- Another feature of DataSet is that it tracks changes that are made to the data it holds before updating the source data.

1. **DataTable** : A DataSet object is made up of a collection of tables, relationships, and constraints. In ADO.NET, DataTable objects are used to represent the tables in a DataSet object.
2. **DataColumn** : Each DataTable object has a Columns collection, which is a container for DataColumn objects. A DataColumn object corresponds to a column in a table.

3. **DataRow** : To access the actual values stored in a DataTable object, use the object's Rows collection, which contains a series of DataRow objects.
4. **DataView** : Once we have retrieved the results of a query into a DataTable object, we can use a DataView object to view the data in different ways. In order to sort the content of the DataTable object based on a column, simply set the DataView object's Sort property to the name of that column. Similarly the Filter property of the DataView can be used so that only the rows that match certain criteria are visible.
5. **DataRelation** : A DataSet, like a database, might contain various interrelated tables. A DataRelation object lets you specify relations between various tables that allow you to both validate data across tables and browse parent and child rows in various Data Tables.

```
6. //Create a DataSet
7. DataSet dset = new DataSet();
```

3.1)STORED PROCEDURE

- Stored Procedures are a set of sql commands which are compiled and are stored inside the database. Every time you execute a sql command, the command is parsed, optimization is done and then the command is executed. Parsing and optimization the command each time you run the query is very expensive. To solve this we have a set of commands collectively called as stored procedure, which are already parsed and optimized and are executed when ever we call them. This article describes about how to call the stored procedures through Ado.net and how to handle the output parameters of the called stored procedures.

- Initially create a object of SqlConnection class which is available in System.Data.SqlClient namespace. You has to provide the connection string as a parameter which includes the Data Source name, the database name and the authentication credentials. Open the connection using the Open() method.

Hide Copy Code

```
            SqlConnection con = new SqlConnection("Data Source=  ;
initial                catalog= Northwind ; User Id=  ; Password=  '");

            con.open();
```

Create the following stored procedure on the Region table in the Northwind database which accepts two parameters and does not have any output parameters.

Hide Copy Code

```
CREATE PROCEDURE RegionUpdate (@RegionID INTEGER,
@RegionDescription NCHAR(50)) AS
SET NOCOUNT OFF
UPDATE Region
SET RegionDescription = @RegionDescription
```

Create a SqlCommand object with the parameters as the name of the stored procedure that is to be executed and the connection object con to which the command is to be sent for execution.

Hide Copy Code

```
SqlCommand command = new SqlCommand("RegionUpdate",con);
```

Change the command objects CommandType property to stored procedure.

Hide Copy Code

```
command.CommandType = CommandType.StoredProcedure;
```

Add the parameters to the command object using the Parameters collection and the SqlParameter class.

Hide Copy Code

```
command.Parameters.Add(new
SqlParameter("@RegionID",SqlDbType.Int,0,"RegionID"));

command.Parameters.Add(new
SqlParameter("@RegionDescription",SqlDbType.NChar,50,"RegionDescription"));
```

Specify the values of the parameters using the Value property of the parameters

Hide Copy Code

```
command.Parameters[0].Value=4;

command.Parameters[1].Value="SouthEast";
```

Excecute the stored procedure using the ExecuteNonQuery method which returns the number of rows effected by the stored procedure.

Hide Copy Code

```
int i=command.ExecuteNonQuery();
```

Now let us see how to execute stored procedures which has output parameters and how to access the results using the output parameters.

Create the following stored procedure which has one output parameter.

Hide Copy Code

```
ALTER PROCEDURE RegionFind(@RegionDescription NCHAR(50) OUTPUT,
@RegionID INTEGER )AS
```

```
            SELECT @RegionDescription =RegionDescription from Region where <A
href="mailto:RegionID=@RegionID">RegionID=@RegionID</A>
```

The above stored procedure accepts regionID as input parameter and finds the RegionDescription for the RegionID input and results it as the output parameter.

Hide Copy Code

```
            SqlCommand command1 = new SqlCommand("RegionFind",con);
            command1.CommandType = CommandType.StoredProcedure;
```

Add the paremeters to the command1

Hide Copy Code

```
            command1.Parameters.Add(new
SqlParameter     ("@RegionDescription",SqlDbType.NChar
,50,ParameterDirection.Output,false,0,50,"RegionDescription",DataRowVersion.Default,null));
            command1.Parameters.Add(new SqlParameter("@RegionID" ,
            SqlDbType.Int,
            0 ,
            "RegionID" ));
```

Observe that the parameter RegionDescription is added with the ParameterDirection as Ouput.

specify the value for the input parameter RegionID.

Hide Copy Code

```
            command1.Parameters["@RegionID"].Value = 4;
```

Assign the UpdatedRowSource property of the SqlCommand object to UpdateRowSource.OutputParameters to indicate that data will be returned from this stored procedure via output parameters.

Hide Copy Code

```
            command1.UpdatedRowSource = UpdateRowSource.OutputParameters;
```

Call the stored procedure and access the RegionDescription for the RegionID 4 using the value property of the parameter.

Hide Copy Code

```
          command1.ExecuteNonQuery();
          string newRegionDescription =(string)
command1.Parameters["@RegionDescription"].Value;
```

Close the sql connection.

Hide Copy Code

```
            con.Close();
```

In the same way you can call the stored procedure that returns a set of rows by defining the parameters as appropriate and executing the command using ExecuteReader() that is used to traverse the records returned by the command.

4.1)LINQ and the ADO.NET EntityFramework: -

4.1.1)LINQ Introduction

Developers across the world have always encountered problems in querying data because of the lack of a defined path and need to master a multiple of technologies like SQL, Web Services, XQuery, etc.

Introduced in Visual Studio 2008 and designed by Anders Hejlsberg, LINQ (Language Integrated Query) allows writing queries even without the knowledge of query languages like SQL, XML etc. LINQ queries can be written for diverse data types.

Example of a LINQ query

C#

```
using System;
using System.Linq;

class Program {
   static void Main() {
      string[] words = {"hello", "wonderful", "LINQ", "beautiful",
"world"};

      //Get only short words
      var shortWords = from word in words where word.Length <= 5 select
word;

      //Print each word out
      foreach (var word in shortWords) {
         Console.WriteLine(word);
      }

      Console.ReadLine();
   }
}
```

VB

```
Module Module1
   Sub Main()
      Dim words As String() = {"hello", "wonderful", "LINQ", "beautiful",
"world"}

      ' Get only short words
```

```
      Dim shortWords = From word In words _ Where word.Length <= 5 _
Select word

      ' Print each word out.

      For Each word In shortWords
         Console.WriteLine(word)
      Next

      Console.ReadLine()
   End Sub
End Module
```

When the above code of C# or VB is compiled and executed, it produces the following result –

```
hello
LINQ
world
```

Syntax of LINQ

There are two syntaxes of LINQ. These are the following ones.

Lamda (Method) Syntax

```
var longWords = words.Where( w ⇒ w.length > 10);
Dim longWords = words.Where(Function(w) w.length > 10)
```

Query (Comprehension) Syntax

```
var longwords = from w in words where w.length > 10;
Dim longwords = from w in words where w.length > 10
```

Types of LINQ

The types of LINQ are mentioned below in brief.

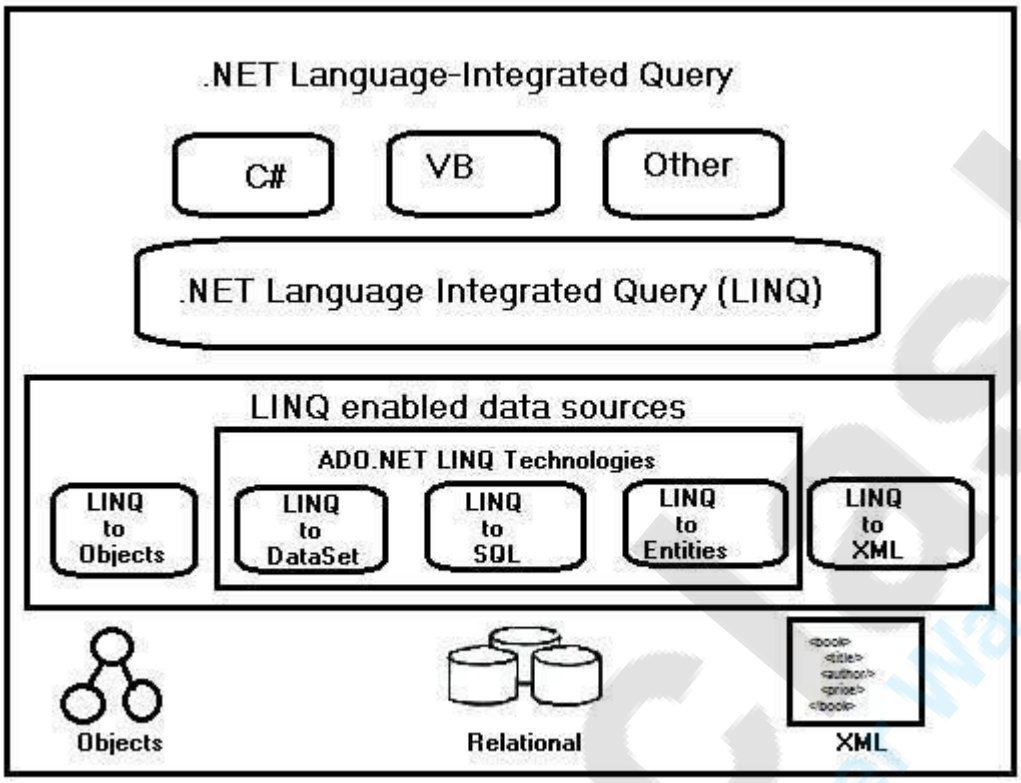
- LINQ to Objects
- LINQ to XML(XLINQ)
- LINQ to DataSet
- LINQ to SQL (DLINQ)
- LINQ to Entities

Apart from the above, there is also a LINQ type named PLINQ which is Microsoft's parallel LINQ.

LINQ Architecture in .NET

LINQ has a 3-layered architecture in which the uppermost layer consists of the language extensions and the bottom layer consists of data sources

that are typically objects implementing IEnumerable <T> or IQueryable <T> generic interfaces. The architecture is shown below.



Query Expressions

Query expression is nothing but a LINQ query, expressed in a form similar to that of SQL with query operators like Select, Where and OrderBy. Query expressions usually start with the keyword "From".

To access standard LINQ query operators, the namespace System.Query should be imported by default. These expressions are written within a declarative query syntax which was C# 3.0.

Below is an example to show a complete query operation which consists of data source creation, query expression definition and query execution.

C#

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;

namespace Operators {
   class LINQQueryExpressions {
      static void Main() {
```

```
         // Specify the data source.
         int[] scores = new int[] { 97, 92, 81, 60 };

         // Define the query expression.
         IEnumerable<int> scoreQuery = from score in scores where score >
80 select score;

         // Execute the query.

         foreach (int i in scoreQuery) {
            Console.Write(i + " ");
         }

         Console.ReadLine();
      }
   }
}
```

When the above code is compiled and executed, it produces the following result –

```
97 92 81
```

Extension Methods

Introduced with .NET 3.5, Extension methods are declared in static classes only and allow inclusion of custom methods to objects to perform some precise query operations to extend a class without being an actual member of that class. These can be overloaded also.

In a nutshell, extension methods are used to translate query expressions into traditional method calls (object-oriented).

Difference between LINQ and Stored Procedure

There is an array of differences existing between LINQ and Stored procedures. These differences are mentioned below.

- Stored procedures are much faster than a LINQ query as they follow an expected execution plan.
- It is easy to avoid run-time errors while executing a LINQ query than in comparison to a stored procedure as the former has Visual Studio's Intellisense support as well as full-type checking during compile-time.
- LINQ allows debugging by making use of .NET debugger which is not in case of stored procedures.

- LINQ offers support for multiple databases in contrast to stored procedures, where it is essential to re-write the code for diverse types of databases.
- Deployment of LINQ based solution is easy and simple in comparison to deployment of a set of stored procedures.

Need For LINQ

Prior to LINQ, it was essential to learn C#, SQL, and various APIs that bind together the both to form a complete application. Since, these data sources and programming languages face an impedance mismatch; a need of short coding is felt.

Below is an example of how many diverse techniques were used by the developers while querying a data before the advent of LINQ.

```
SqlConnection sqlConnection = new SqlConnection(connectString);
SqlConnection.Open();

System.Data.SqlClient.SqlCommand sqlCommand = new SqlCommand();
sqlCommand.Connection = sqlConnection;

sqlCommand.CommandText = "Select * from Customer";
return sqlCommand.ExecuteReader (CommandBehavior.CloseConnection)
```

Interestingly, out of the featured code lines, query gets defined only by the last two. Using LINQ, the same data query can be written in a readable color-coded form like the following one mentioned below that too in a very less time.

```
Northwind db = new Northwind(@"C:\Data\Northwnd.mdf");

var query = from c in db.Customers select c;
```

Advantages of LINQ

LINQ offers a host of advantages and among them the foremost is its powerful expressiveness which enables developers to express declaratively. Some of the other advantages of LINQ are given below.

- LINQ offers syntax highlighting that proves helpful to find out mistakes during design time.
- LINQ offers IntelliSense which means writing more accurate queries easily.
- Writing codes is quite faster in LINQ and thus development time also gets reduced significantly.
- LINQ makes easy debugging due to its integration in the C# language.

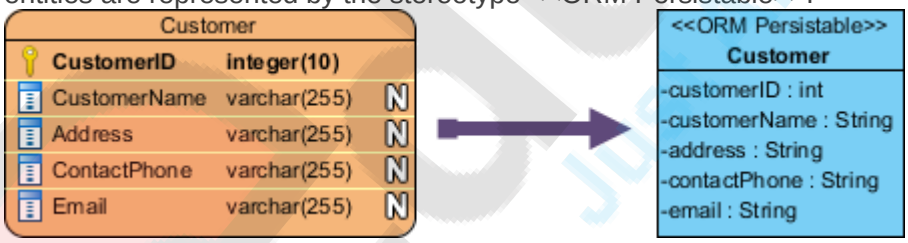
- Viewing relationship between two tables is easy with LINQ due to its hierarchical feature and this enables composing queries joining multiple tables in less time.
- LINQ allows usage of a single LINQ syntax while querying many diverse data sources and this is mainly because of its unitive foundation.
- LINQ is extensible that means it is possible to use knowledge of LINQ to querying new data source types.
- LINQ offers the facility of joining several data sources in a single query as well as breaking complex problems into a set of short queries easy to debug.
- LINQ offers easy transformation for conversion of one data type to another like transforming SQL data to XML data.

4.1.2)Mapping your Data Model to an Object Model

Visual Paradigm supports Object Relational Mapping (ORM) which maps data model to object model and vice versa. Visual Paradigm helps mapping entities to classes. Source files can then be generated from classes for use in software development. The mapping to object model preserves the data, but also the state, foreign/primary key mapping, difference in data type and business logic. Thus, you are not required to handle those tedious tasks during software development.

Mapping entities to classes

All entities map one-to-one to persistent classes in an object model. Classes that map with entities are represented by the stereotype <<ORM Persistable>>.

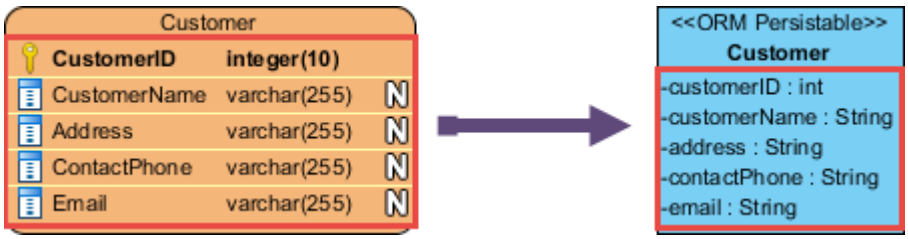


Mapping class

In the above example, the *Customer* entity map one-to-one the *Customer* class as the *Customer* instance can store the customer information from the *Customer* Entity.

Mapping columns to attributes

Since all entities map one-to-one to persistent classes in an object model, columns in turn map to attributes in a one-to-one mapping. Visual Paradigm ignores all specialty columns such as computed columns and foreign key columns.



Mapping columns

Mapping data type

Column types are automatically mapped to appropriate attribute types. The following table lists out the typical mapping between data model and object model.

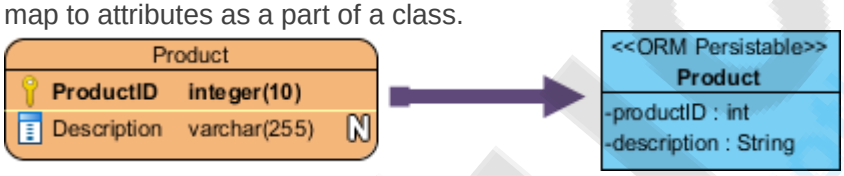
Data Model	Object Model
bigint	Long
binary	Byte []
bit	Boolean
blob	Blob
varchar	String
char	Character
char (1)	Character
clob	String
date	Date
decimal	BigDecimal
double	Double
float	Float
integer	Integer

numeric	BigDecimal
real	Float
time	Time
timestamp	Timestamp
tinyint	Byte
smallint	Short
varbinary	Byte []

Mapping of data types between data and object model

Mapping primary key

As the columns map to attributes in a one-to-one mapping, primary key columns in the entity map to attributes as a part of a class.



Mapping primary key

In the example, the primary key of entity *Product*, *ProductID* maps to an attribute *ProductID* of the class *Product*.

Mapping relationship

Relationship represents the correlation among entities. Each entity of a relationship has a role, called Phrase describing how the entity acts in it. The phrase is attached at the end of a relationship connection line. Visual Paradigm maps the phrase to role name of association in the object model.

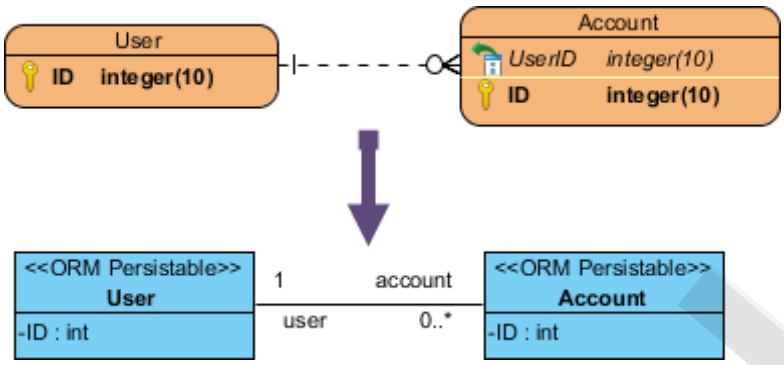
Mapping cardinality

Cardinality refers to the number of possible instances of an entity relate to one instance of another entity. The following table shows the syntax to express the Cardinality.

Type of cardinality	Description
	Zero or one intance

	Zero or more instances
	Exactly one instance
	One or more instances

Description of cardinalities

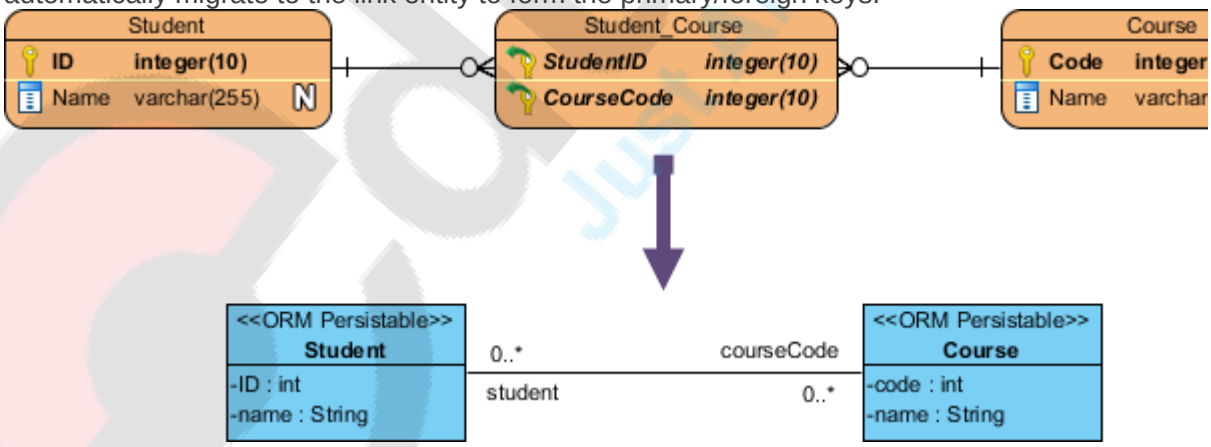


Mapping cardinality

In the above example, it shows that a user contains multiple accounts.

Mapping many-to-many relationship

For each many-to-many relationship between entities, Visual Paradigm generates a link entity to form two one-to-many relationships in between. The primary keys of the two entities will automatically migrate to the link entity to form the primary/foreign keys.



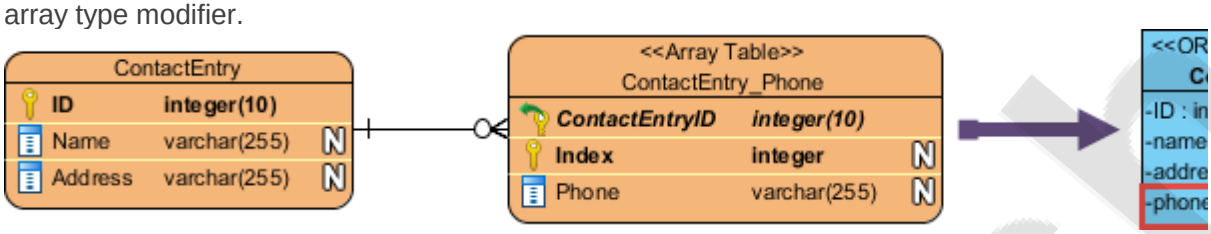
Mapping many-to-many relationship

In the above example, Visual Paradigm generates the link entity once a many-to-many relationship is setup between two entities. To transform the many-to-many relationship, Visual Paradigm maps the many-to-many relationship to many-to-many association.

Mapping Array Table to collection of objects

Visual Paradigm promotes the use of Array Table to allow users retrieve objects in the form of primitive array.

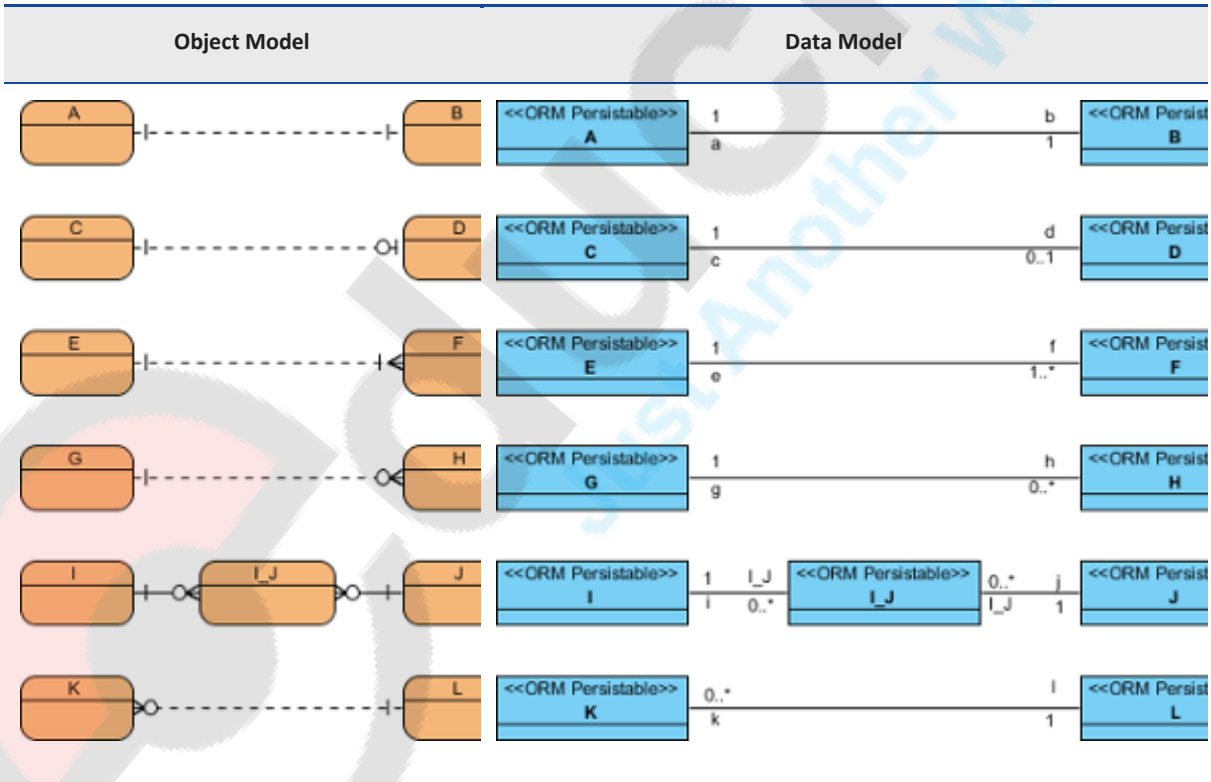
When Visual Paradigm transforms an array table, the array table will map to an attribute with array type modifier.



Mapping Array Table to collection of objects

In the above example, each *Contact* may have more than one phone numbers, stored in *ContactEntry_Phone*. The array table of *ContactEntry_Phone*maps into the phone attribute with array type modifier in the *ContactEntry* class.

Typical mapping between data model and object model



Typical mapping between data model and object model

UNIT 4

ASP.NET WEB APPLICATIONS

INTRODUCTION: -

- ASP.NET is a web development platform, which provides a programming model, a comprehensive software infrastructure and various services required to build up robust web applications for PC, as well as mobile devices.
- ASP.NET works on top of the HTTP protocol, and uses the HTTP commands and policies to set a browser-to-server bilateral communication and cooperation.
- ASP.NET is a part of Microsoft .Net platform. ASP.NET applications are compiled codes, written using the extensible and reusable components or objects present in .Net framework. These codes can use the entire hierarchy of classes in .Net framework.

The ASP.NET application codes can be written in any of the following languages:

- C#
- Visual Basic.Net
- Jscript
- J#

ASP.NET is used to produce interactive, data-driven web applications over the internet. It consists of a large number of controls such as text boxes, buttons, and labels for assembling, configuring, and manipulating code to create HTML pages.

.NET is language independent, which means you can use any .NET supported language to make .NET applications. The most common languages for writing ASP.NET applications are C# and VB.NET. While VB.NET is directly based VB (Visual Basic), C# was introduced together with the .NET framework, and is therefore a somewhat new language.

ASP.NET Web Forms Model

ASP.NET web forms extend the event-driven model of interaction to the web applications. The browser submits a web form to the web server and the server returns a full markup page or HTML page in response.

All client side user activities are forwarded to the server for stateful processing. The server processes the output of the client actions and triggers the reactions.

Now, HTTP is a stateless protocol. ASP.NET framework helps in storing the information regarding the state of the application, which consists of:

- Page state
- Session state

The page state is the state of the client, i.e., the content of various input fields in the web form. The session state is the collective information obtained from various pages the user visited and worked with, i.e., the overall session state. To clear the concept, let us take an example of a shopping cart.

User adds items to a shopping cart. Items are selected from a page, say the items page, and the total collected items and price are shown on a different page, say the cart page. Only HTTP cannot keep track of all the information coming from various pages. ASP.NET session state and server side infrastructure keeps track of the information collected globally over a session.

The ASP.NET runtime carries the page state to and from the server across page requests while generating ASP.NET runtime codes, and incorporates the state of the server side components in hidden fields.

This way, the server becomes aware of the overall application state and operates in a two-tiered connected way.

The ASP.NET Component Model

The ASP.NET component model provides various building blocks of ASP.NET pages. Basically it is an object model, which describes:

- Server side counterparts of almost all HTML elements or tags, such as <form> and <input>.

- Server controls, which help in developing complex user-interface. For example, the Calendar control or the Gridview control.

ASP.NET is a technology, which works on the .Net framework that contains all web-related functionalities. The .Net framework is made of an object-oriented hierarchy. An ASP.NET web application is made of pages. When a user requests an ASP.NET page, the IIS delegates the processing of the page to the ASP.NET runtime system.

The ASP.NET runtime transforms the .aspx page into an instance of a class, which inherits from the base class page of the .Net framework. Therefore, each ASP.NET page is an object and all its components i.e., the server-side controls are also objects.

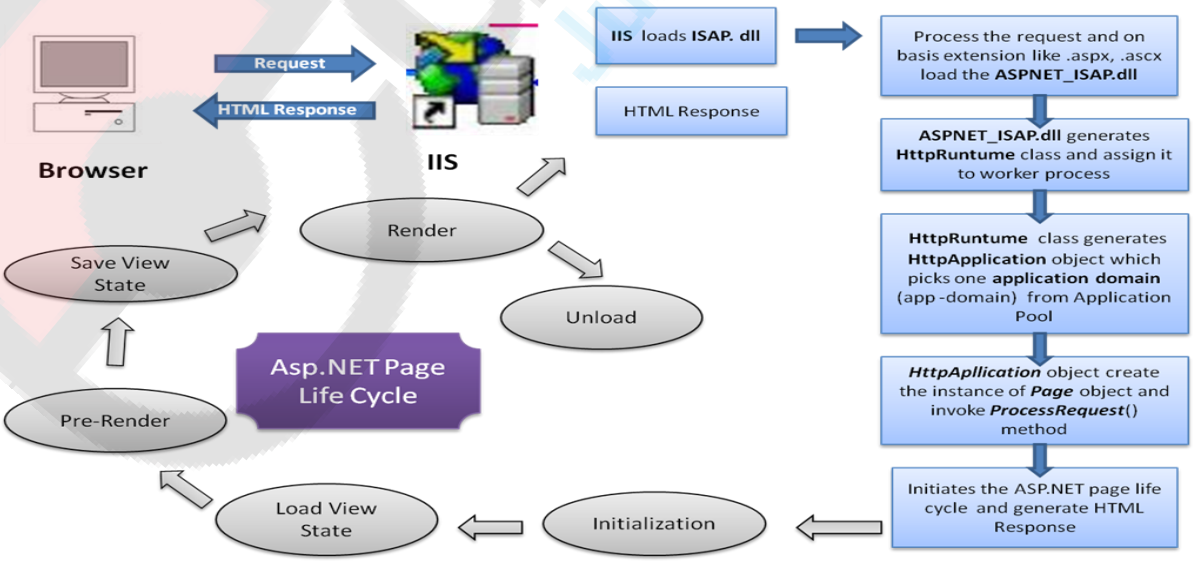
4.1) LIFECYCLE OF ASP.NET WEB PAGES

In ASP.NET, a web page has execution lifecycle that includes various phases. These phases include initialization, instantiation, restoring and maintaining state etc. it is required to understand the page lifecycle so that we can put custom code at any stage to perform our business logic.

The ASP.NET life cycle could be divided into two groups:

- **Application Life Cycle**
- **Page Life Cycle**

FIGURE:-



4.1.1) ASP.NET APPLICATION LIFECYCLE

The application life cycle has the following stages:

- User makes a request for accessing application resource, a page. Browser sends this request to the web server.
- A unified pipeline receives the first request and the following events take place:
    - An object of the class ApplicationManager is created.
    - An object of the class HostingEnvironment is created to provide information regarding the resources.
    - Top level items in the application are compiled.
- Response objects are created. The application objects such as HttpContext, HttpRequest and HttpResponse are created and initialized.
- An instance of the HttpApplication object is created and assigned to the request.
- The request is processed by the HttpApplication class. Different events are raised by this class for processing the request.

4.1.2) ASP.NET PAGE LIFECYCLE

When a page is requested, it is loaded into the server memory, processed, and sent to the browser. Then it is unloaded from the memory. At each of these steps, methods and events are available, which could be overridden according to the need of the application. In other words, you can write your own code to override the default code.

The Page class creates a hierarchical tree of all the controls on the page. All the components on the page, except the directives, are part of this control tree. You can see the control tree by adding trace= "true" to the page directive. We will cover page directives and tracing under 'directives' and 'event handling'.

The page life cycle phases are:

- Initialization

- Instantiation of the controls on the page
- Restoration and maintenance of the state
- Execution of the event handler codes
- Page rendering

Understanding the page cycle helps in writing codes for making some specific thing happen at any stage of the page life cycle. It also helps in writing custom controls and initializing them at right time, populate their properties with view-state data and run control behavior code.

Following are the different stages of an ASP.NET page:

- **Page request** - When ASP.NET gets a page request, it decides whether to parse and compile the page, or there would be a cached version of the page; accordingly the response is sent.
- **Starting of page life cycle** - At this stage, the Request and Response objects are set. If the request is an old request or post back, the IsPostBack property of the page is set to true. The UICulture property of the page is also set.
- **Page initialization** - At this stage, the controls on the page are assigned unique ID by setting the UniqueID property and the themes are applied. For a new request, postback data is loaded and the control properties are restored to the view-state values.
- **Page load** - At this stage, control properties are set using the view state and control state values.
- **Validation** - Validate method of the validation control is called and on its successful execution, the IsValid property of the page is set to true.
- **Postback event handling** - If the request is a postback (old request), the related event handler is invoked.
- **Page rendering** - At this stage, view state for the page and all controls are saved. The page calls the Render method for each control and the output of rendering is written to the OutputStream class of the Response property of page.
- **Unload** - The rendered page is sent to the client and page properties, such as Response and Request, are unloaded and all cleanup done.

ASP.NET PAGE LIFECYCLE EVENTS

At each stage of the page life cycle, the page raises some events, which could be coded. An event handler is basically a function or subroutine, bound to the event, using declarative attributes such as Onclick or handle.

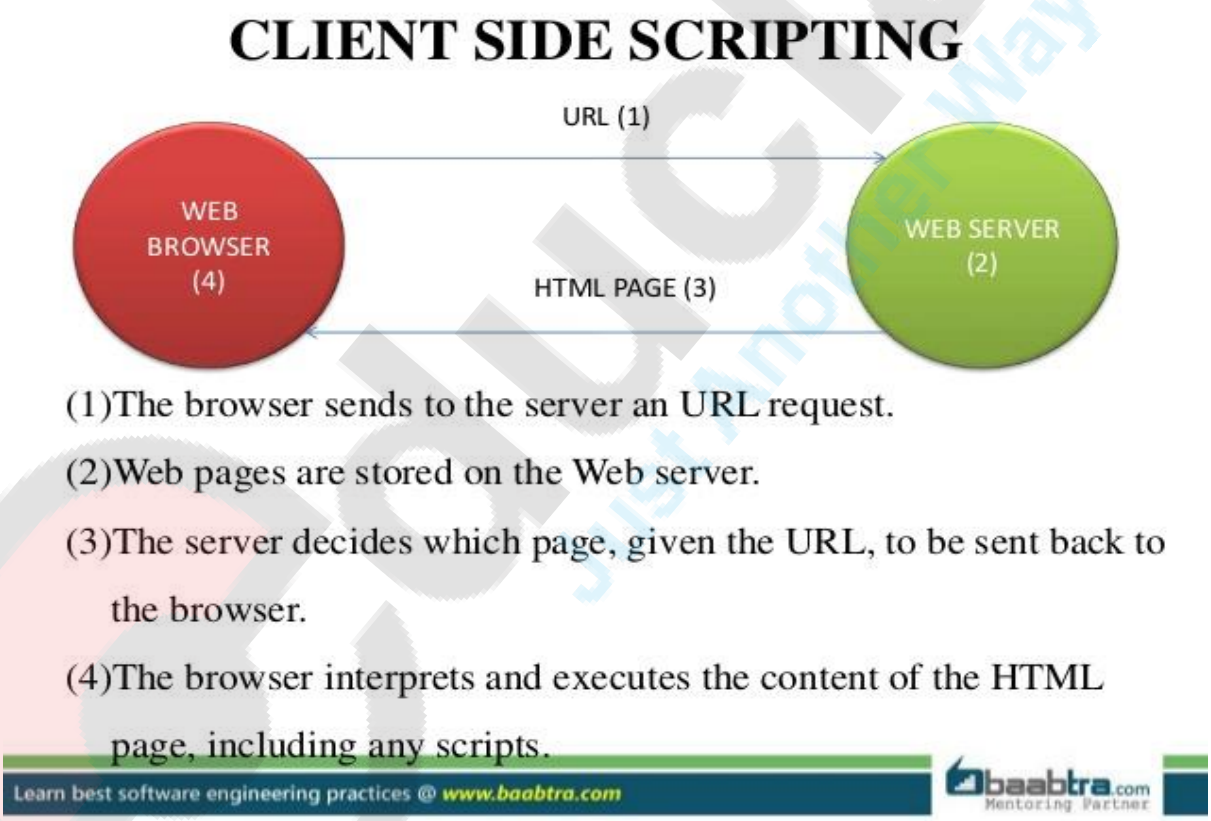
Following are the page life cycle events:

- **PreInit** - PreInit is the first event in page life cycle. It checks the IsPostBack property and determines whether the page is a postback. It sets the themes and master pages, creates dynamic controls, and gets and sets profile property values. This event can be handled by overloading the OnPreInit method or creating a Page_PreInit handler.
- **Init** - Init event initializes the control property and the control tree is built. This event can be handled by overloading the OnInit method or creating a Page_Init handler.
- **InitComplete** - InitComplete event allows tracking of view state. All the controls turn on view-state tracking.
- **LoadViewState** - LoadViewState event allows loading view state information into the controls.
- **LoadPostData** - During this phase, the contents of all the input fields are defined with the <form> tag are processed.
- **PreLoad** - PreLoad occurs before the post back data is loaded in the controls. This event can be handled by overloading the OnPreLoad method or creating a Page_PreLoad handler.
- **Load** - The Load event is raised for the page first and then recursively for all child controls. The controls in the control tree are created. This event can be handled by overloading the OnLoad method or creating a Page_Load handler.
- **LoadComplete** - The loading process is completed, control event handlers are run, and page validation takes place. This event can be handled by overloading the OnLoadComplete method or creating a Page_LoadComplete handler
- **PreRender** - The PreRender event occurs just before the output is rendered. By handling this event, pages and controls can perform any updates before the output is rendered.
- **PreRenderComplete** - As the PreRender event is recursively fired for all child controls, this event ensures the completion of the pre-rendering phase.

- **SaveStateComplete** - State of control on the page is saved. Personalization, control state and view state information is saved. The HTML markup is generated. This stage can be handled by overriding the Render method or creating a Page_Render handler.
- **UnLoad** - The UnLoad phase is the last phase of the page life cycle. It raises the UnLoad event for all controls recursively and lastly for the page itself. Final cleanup is done and all resources and references, such as database connections, are freed. This event can be handled by modifying the OnUnLoad method or creating a Page_UnLoad handler.

4.2) ROLE OF CLIENT SIDE SCRIPTING

FIGURE:-



ASP.NET client side coding has two aspects:

- **Client side scripts** : It runs on the browser and in turn speeds up the execution of page. For example, client side data validation which can catch invalid data and warn the user accordingly without making a round trip to the server.
- **Client side source code** : ASP.NET pages generate this. For example, the HTML source code of an ASP.NET page contains a number of hidden fields and automatically injected blocks of JavaScript code, which keeps information like view state or does other jobs to make the page work.

4.2.1)CLIENT SIDE SCRIPTS

All ASP.NET server controls allow calling client-side code written using JavaScript or VBScript. Some ASP.NET server controls use client-side scripting to provide response to the users without posting back to the server. For example, the validation controls.

Apart from these scripts, the Button control has a property OnClientClick, which allows executing client-side script, when the button is clicked.

The traditional and server HTML controls have the following events that can execute a script when they are raised:

Event	Description
onblur	When the control loses focus
onfocus	When the control receives focus
onclick	When the control is clicked
onchange	When the value of the control changes
onkeydown	When the user presses a key
onkeypress	When the user presses an alphanumeric key
onkeyup	When the user releases a key
onmouseover	When the user moves the mouse pointer over the control
onserverclick	It raises the ServerClick event of the control, when the control is clicked

4.2.2)CLIENT SIDE SOURCE CODE

We have already discussed that, ASP.NET pages are generally written in two files:

- The content file or the markup file (.aspx)
- The code-behind file

The content file contains the HTML or ASP.NET control tags and literals to form the structure of the page. The code behind file contains the class definition. At run-time, the content file is parsed and transformed into a page class.

This class, along with the class definition in the code file, and system generated code, together make the executable code (assembly) that processes all posted data, generates response, and sends it back to the client.

Consider the simple page:

```
<%@ Page Language="C#" AutoEventWireup="true" CodeBehind="Default.aspx.cs"
  Inherits="clientside._Default" %>

<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
  "http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">

<html xmlns="http://www.w3.org/1999/xhtml" >

  <head runat="server">
    <title>
      Untitled Page
    </title>
  </head>

  <body>
    <form id="form1" runat="server">

      <div>
        <asp:TextBox ID="TextBox1" runat="server"></asp:TextBox>
        <asp:Button ID="Button1" runat="server" OnClick="Button1_Click" Text="Click" />
      </div>

      <hr />

      <h3> <asp:Label ID="Msg" runat="server" Text=""> </asp:Label> </h3>
    </form>
  </body>

</html>
```

When this page is run on the browser, the View Source option shows the HTML page sent to the browser by the ASP.Net runtime:

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
  "http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">

<html xmlns="http://www.w3.org/1999/xhtml" >

  <head>
    <title>
      Untitled Page
    </title>
  </head>

  <body>
    <form name="form1" method="post" action="Default.aspx" id="form1">

      <div>
        <input type="hidden" name="__VIEWSTATE" id="__VIEWSTATE"
          value="/wEPDwUKMTU5MTA2ODYwOWRk31NudGDgvhhA7joJum9Qn5RxU2M=" />
      </div>

      <div>
        <input type="hidden" name="__EVENTVALIDATION"  id="__EVENTVALIDATION"
          value="/wEWAwKpjZj0DALs0bLrBgKM54rGBhHsyM61rraxE+KnBTCS8cd1QDJ/"/>
      </div>

      <div>
        <input name="TextBox1" type="text" id="TextBox1" />
        <input type="submit" name="Button1" value="Click" id="Button1" />
      </div>

      <hr />
      <h3><span id="Msg"></span></h3>

    </form>
  </body>
</html>
```

If you go through the code properly, you can see that first two tags contain the hidden fields which store the view state and validation information.

4.3) POSTBACK POSTING AND CROSS PAGE POSTING IN ASP.NET

4.3.1) POSTBACK POSTING

All the web applications are running on Web Servers. Whenever a user made a request to the web server, the web server has to return the response to the user. PostBack is the name given to the process of submitting all the information that the user is currently working on and send it all back to the server. Postback is actually sending all the information from client to web server, then web server process all those contents and returns back to client.

Example:

```
if (!IsPostback)
   // generate dynamic form
else
   process submitted data;
```

A postback originates from the client side browser. When the web page and its contents are sent to the web server for processing some information and then, the web server posts the same page back to the client browser. Normally one of the controls on the page will be manipulated by the user (e.g. button click), and this control will initiate a postback. Then the state of this control and all other controls on the page is Posted Back to the web server.

ASP.NET also adds two additional hidden input fields that are used to exchange information back to the server. This information consists of ID of the Control that raised the event and any additional information if needed. These fields will empty initially like the following:

```
<input type="hidden" name="__EVENTTARGET" id="__EVENTTARGET" value="" />
<input type="hidden" name="__EVENTARGUMENT" id="__EVENTARGUMENT" value=""
/>
```

The _doPostBack() function has the responsibility for setting these values with the appropriate information about the event and the submitting the form. The _doPostBack() function is shown below:

```
<script language="text/javascript">
        function __doPostBack(eventTarget, eventArgument) {
                if (!theForm.onsubmit  (theForm.onsubmit() != false)) {
                theForm.__EVENTTARGET.value = eventTarget;
                theForm.__EVENTARGUMENT.value = eventArgument;
                theForm.submit();
        }
</script>
```

ASP.NET generates the _doPostBack() function automatically, provided at least one control on the page uses automatic postbacks.

You can also pass an event argument along with the target in case you need to pass something to your code behind:

Example:

```
__doPostBack( "Button1', '<event argument here>' )
```

This would be captured in the code behind as Request.Form["__EVENTARGUEMENT"]

So in this way you can use __doPostBack.

4.4) ASP.NET COMPILATION MODEL

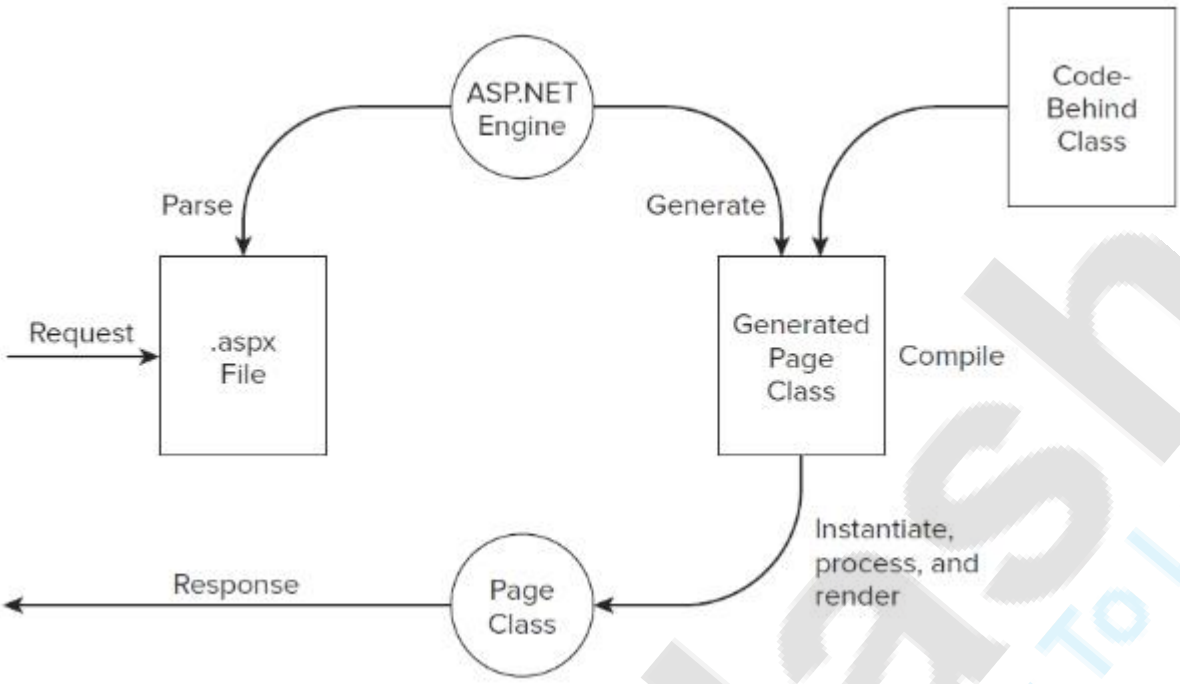
- One of the most important improvements Microsoft has made to the ASP development environment is to build the Web request handling framework out of classes. Pushing request processing into a class-based architecture allows for a Web-handling framework that's compiled. When ASP.NET pages are first accessed, they are compiled into assemblies.

- This is advantageous because subsequent access loads the page directly from the assembly. Whereas classic ASP interpreted the same script code over and over, ASP.NET applications are compiled into .NET assemblies and ultimately perform better and are safer.

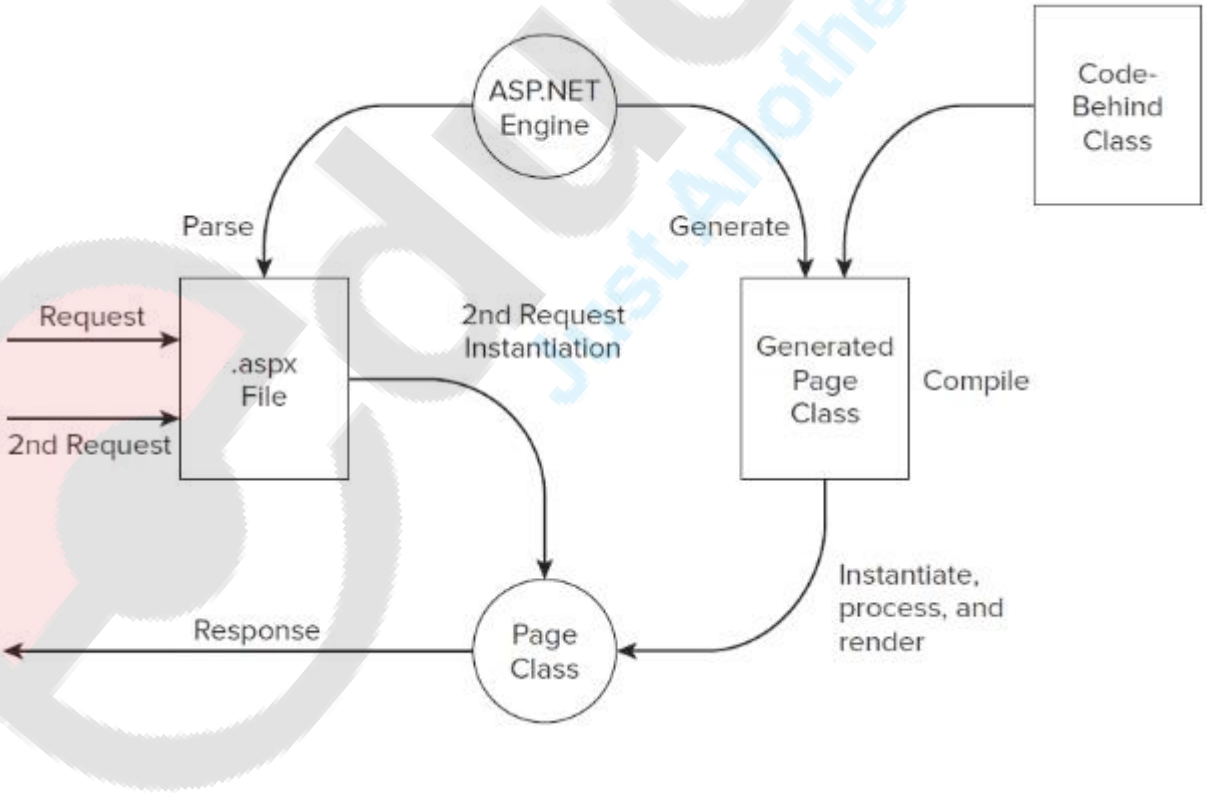
- In addition, compiling the Web request framework allows for more robust and consistent debugging. Whenever you run an ASP.NET application from Visual Studio, you can debug it as though it were a normal desktop application.

- ASP.NET compiles .aspx files automatically. To get an .aspx page to compile, you simply need to surf to the .aspx file containing the code. When you do so, ASP.NET compiles the page into a class. However, you won't see that class anywhere near your virtual directory. ASP.NET copies the resulting assemblies to a temporary directory.

When an ASP.NET page is referenced in the browser for the first time, the request is passed to the ASP.NET parser that creates the class file in the language of the page. It is passed to the ASP.NET parser based on the file's extension (.aspx) because ASP.NET realizes that this file extension type is meant for its handling and processing. After the class file has been created, the class file is compiled into a DLL and then written to the disk of the Web server. At this point, the DLL is instantiated and processed, and an output is generated for the initial requester of the ASP.NET page. This is detailed in this Figure.



On the next request, great things happen. Instead of going through the entire process again for the second and respective requests, the request simply causes an instantiation of the already-created DLL, which sends out a response to the requester. This is illustrated in this Figure.



Because of the mechanics of this process, if you made changes to your .aspx code-behind pages, you found it necessary to recompile your application. This was quite a pain if you had a larger site and did not want your end users to experience the extreme lag that occurs when an .aspx page is referenced for the first time after compilation. Many developers, consequently, began to develop their own tools that automatically go out and hit every single page within their application to remove this first-time lag hit from the end user's browsing experience.

ASP.NET provides a few ways to precompile your entire application with a single command that you can issue through a command line. One type of compilation is referred to as in-place precompilation. To precompile your entire ASP.NET application, you must use the aspnet_compiler.exe tool that comes with ASP.NET. You navigate to the tool using the Command window. Open the Command window and navigate to *C:WindowsMicrosoft.NETFrameworkv4.0.xxxxx*. When you are there, you can work with the aspnet_compiler tool. You can also get to this tool directly from the Visual Studio 2010 Command Prompt. Choose Start ➪ All Programs ➪ Microsoft Visual Studio 2010 ➪ Visual Studio Tools ➪ Visual Studio Command Prompt (2010).

After you get the command prompt, you use the aspnet_compiler.exe tool to perform an in-place precompilation using the following command:

```
1
```

```
aspnet_compiler -p "C:Inetpubwwwrootcode" -v none
```

You then get a message stating that the precompilation is successful. The other great thing about this precompilation capability is that you can also use it to find errors on any of the ASP.NET pages in your application. Because it hits each and every page, if one of the pages contains an error that won't be triggered until runtime, you get notification of the error immediately as you employ this precompilation method. The next precompilation option is commonly referred to as precompilation for deployment. This outstanding capability of ASP.NET enables you to compile your application down to some DLLs, which can then be deployed to customers, partners, or elsewhere for your own use. Not only are minimal steps required to do this, but also after your application is compiled, you simply have to move around the DLL and some placeholder files for the site to work. This means that your Web site code is completely removed and placed in the DLL when deployed.

However, before you take these precompilation steps, create a folder in your root drive called, for example, code. This folder is the one to which you will direct the compiler output. When it is in place, you can return to the compiler tool and give the following command:

```
1  aspnet_compiler -v [Application Name] -p [Physical Location][Target]
```

Therefore, if you have an application called ThomsonReuters located at C:Websitescompile, you use the following commands:

1

```
aspnet_compiler -v /compile -p C:Websitescompile C:code
```

Press the Enter key, and the compiler either tells you that it has a problem with one of the command parameters or that it was successful. If it was successful, you can see the output placed in the target directory.

In the example just shown, -v is a command for the virtual path of the application, which is provided by using /compile. The next command is –p, which is pointing to the physical path of the application. In this case, it is C:Websitescompile. Finally, the last bit, C:code, is the location of the compiler output. Table describes some of the possible commands for the aspnet_compiler.exe tool.

Command	Description
-m	Specifies the full IIS metabase path of the application. If you use the -m command, you cannot use the -v or -p command.
-v	Specifies the virtual path of the application to be compiled. If you also use the -p command, the physical path is used to find the location of the application.
-p	Specifies the physical path of the application to be compiled. If this is not specified, the IIS metabase is used to find the application.
-u	If this command is utilized, it specifies that the application is updatable.
-f	Specifies to overwrite the target directory if it already exists.
-d	Specifies that the debug information should be excluded from the compilation process.
[targetDir]	Specifies the target directory where the compiled files should be placed. If this is not specified, the output files are placed in the application directory.

After compiling the application, you can go to C:Wrox to see the output. Here you see all the files and the file structures that were in the original application. However, if you look at the content of one of the files, notice that the file is simply a placeholder. In the actual file, you find the following comment:

This is a marker file generated by the precompilation tool and should not be deleted!

In fact, you find a Code.dll file in the bin folder where all the page code is located. Because it is in a DLL file, it provides great code obfuscation as well. From here on, all you do is move these files to another server using FTP or Windows Explorer, and you can run the entire Web

application from these files. When you have an update to the application, you simply provide a new set of compiled files.

Note that this compilation process does not compile every type of Web file. In fact, it compiles only the ASP.

NET-specific file types and leaves out of the compilation process the following types of files:

- HTML files
- XML files
- XSD files
- web.config files
- Text files

You cannot do much to get around this, except in the case of the HTML files and the text files. For these file types, just change the file extensions of these file types to .aspx; they are then compiled into the Code.dll like all the other ASP.NET files.

4.5) ASP.NET HTML CONTROLS

The HTML server controls are basically the standard HTML controls enhanced to enable server side processing. The HTML controls such as the header tags, anchor tags, and input elements are not processed by the server but are sent to the browser for display.

They are specifically converted to a server control by adding the attribute runat="server" and adding an id attribute to make them available for server-side processing.

For example, consider the HTML input control:

```
<input type="text" size="40">
```

It could be converted to a server control, by adding the runat and id attribute:

```
<input type="text" id="testtext" size="40" runat="server">
```

Advantages of using HTML Server Controls

Although ASP.NET server controls can perform every job accomplished by the HTML server controls, the later controls are useful in the following cases:

- Using static tables for layout purposes.
- Converting a HTML page to run under ASP.NET

The following table describes the HTML server controls:

Control Name	HTML tag

HtmlHead	<head>element
HtmlInputButton	<input type=button|submit|reset>
HtmlInputCheckbox	<input type=checkbox>
HtmlInputFile	<input type = file>
HtmlInputHidden	<input type = hidden>
HtmlInputImage	<input type = image>
HtmlInputPassword	<input type = password>
HtmlInputRadioButton	<input type = radio>
HtmlInputReset	<input type = reset>
HtmlText	<input type = text|password>
HtmlImage	<img> element
HtmlLink	<link> element
HtmlAnchor	<a> element
HtmlButton	<button> element
HtmlButton	<button> element
HtmlForm	<form> element
HtmlTable	<table> element
HtmlTableCell	<td> and <th>
HtmlTableRow	<tr> element
HtmlTitle	<title> element
HtmlSelect	<select&t; element
HtmlGenericControl	All HTML controls not listed

HTML Components

The following table contains commonly used HTML components.

Controls Name	Description

Button	It is used to create HTML button.
Reset Button	It is used to reset all HTML form elements.
Submit Button	It is used to submit form data to the server.
Text Field	It is used to create text input.
Text Area	It is used to create a text area in the html form.
File	It is used to create a input type = "file" component which is used to upload file to the server.
Password	It is a password field which is used to get password from the user.
CheckBox	It creates a check box that user can select or clear.
Radio Button	A radio field which is used to get user choice.
Table	It allows us to present information in a tabular format.
Image	It displays an image on an HTML form
ListBox	It displays a list of items to the user. You can set the size from two or more to specify how many items you wish to show.
Dropdown	It displays a list of items to the user in a dropdown list.
Horizontal Rule	It displays a horizontal line across the HTML page.

Example 1

Here, we are implementing an HTML server control in the form.

// htmlcontrolsexample.aspx

```
<%@ Page Language="C#" AutoEventWireup="true" CodeBehind="htmlcontrolsexample.aspx.cs"
```

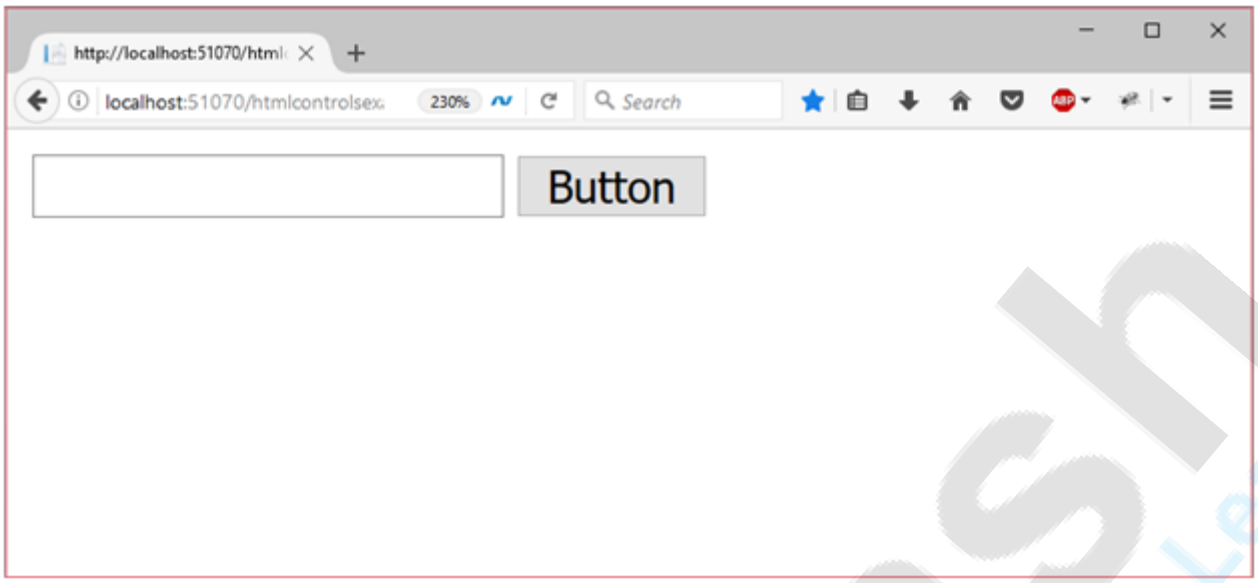
```
Inherits="asp.netexample.htmlcontrolsexample" %>
<!DOCTYPE html>
<html xmlns="http://www.w3.org/1999/xhtml">
<head runat="server">
<title></title>
</head>
<body>
<form id="form1" runat="server">
<div>
<input id="Text1" type="text" runat="server"/>
<asp:Button ID="Button1" runat="server" Text="Button" OnClick="Button1_Click"/>
</div>
</form>
</body>
</html>
```

This application contains a code behind file.

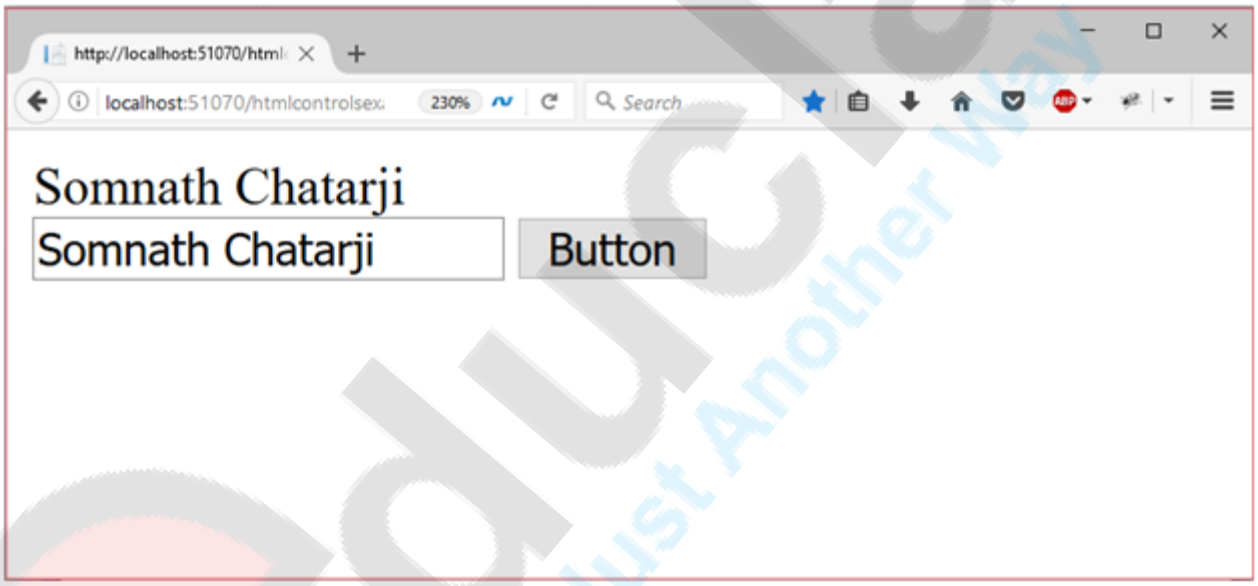
// htmlcontrolsexample.aspx.cs

```
using System;
namespace asp.netexample
{
public partial class htmlcontrolsexample : System.Web.UI.Page
   {
protected void Page_Load(object sender, EventArgs e)
      {
      }
protected void Button1_Click(object sender, EventArgs e)
      {
string a = Request.Form["Text1"];
         Response.Write(a);
      }
   }
}
```

Output:



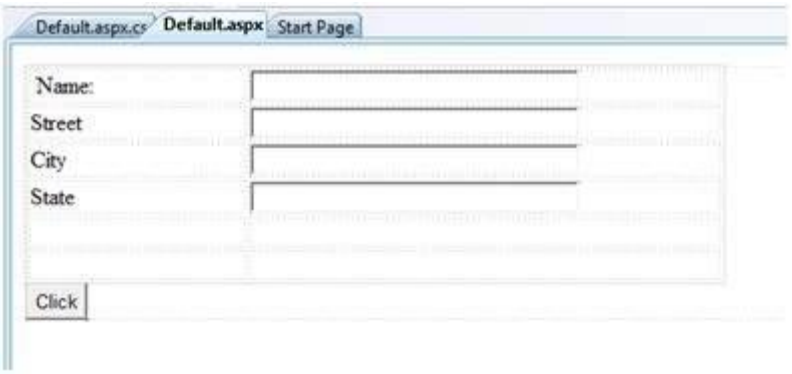
When we click the button after entering text, it responses back to client.



Example 2

The following example uses a basic HTML table for layout. It uses some boxes for getting input from the users such as name, address, city, state etc. It also has a button control, which is clicked to get the user data displayed in the last row of the table.

The page should look like this in the design view:



The code for the content page shows the use of the HTML table element for layout.

```
<%@ Page Language="C#" AutoEventWireup="true" CodeBehind="Default.aspx.cs"
Inherits="htmlserver._Default" %>

<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
"http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">

<html xmlns="http://www.w3.org/1999/xhtml" >

   <head runat="server">
      <title>Untitled Page</title>

      <style type="text/css">
         .style1
         {
            width: 156px;
         }
         .style2
         {
            width: 332px;
         }
      </style>

   </head>

   <body>
      <form id="form1" runat="server">
         <div>
            <table style="width: 54%;">
               <tr>
                  <td class="style1">Name:</td>
                  <td class="style2">
                     <asp:TextBox ID="txtname" runat="server"  style="width:230px">
                     </asp:TextBox>
                  </td>
               </tr>

               <tr>
                  <td class="style1">Street</td>
                  <td class="style2">
                     <asp:TextBox ID="txtstreet" runat="server"  style="width:230px">
                     </asp:TextBox>
                  </td>
               </tr>

               <tr>
                  <td class="style1">City</td>
                  <td class="style2">
                     <asp:TextBox ID="txtcity" runat="server"  style="width:230px">
```

```
                    </asp:TextBox>
                </td>
            </tr>

            <tr>
                <td class="style1">State</td>
                <td class="style2">
                    <asp:TextBox ID="txtstate" runat="server" style="width:230px">
                    </asp:TextBox>
                </td>
            </tr>

        <tr>
                <td class="style1"> </td>
                <td class="style2"></td>
            </tr>

            <tr>
                <td class="style1"></td>
                <td ID="displayrow" runat ="server" class="style2">
                </td>
            </tr>
        </table>

      </div>
      <asp:Button ID="Button1" runat="server" onclick="Button1_Click" Text="Click" />
    </form>
  </body>
</html>
```

The code behind the button control:

```
protected void Button1_Click(object sender, EventArgs e)
{
  string str = "";
  str += txtname.Text + "<br />";
  str += txtstreet.Text + "<br />";
  str += txtcity.Text + "<br />";
  str += txtstate.Text + "<br />";
  displayrow.InnerHtml = str;
}
```

Observe the following:

- The standard HTML tags have been used for the page layout.
- The last row of the HTML table is used for data display. It needed server side processing, so an ID attribute and the runat attribute has been added to it.

4.6) ASP.NET SERVER CONTROLS

Controls are small building blocks of the graphical user interface, which include text boxes, buttons, check boxes, list boxes, labels, and numerous other tools. Using these tools, the users can enter data, make selections and indicate their preferences.

Controls are also used for structural jobs, like validation, data access, security, creating master pages, and data manipulation.

ASP.NET uses five types of web controls, which are:

- HTML controls
- HTML Server controls
- ASP.NET Server controls
- ASP.NET Ajax Server controls
- User controls and custom controls

ASP.NET server controls are the primary controls used in ASP.NET. These controls can be grouped into the following categories:

- **Validation controls** - These are used to validate user input and they work by running client-side script.
- **Data source controls** - These controls provides data binding to different data sources.
- **Data view controls** - These are various lists and tables, which can bind to data from data sources for displaying.
- **Personalization controls** - These are used for personalization of a page according to the user preferences, based on user information.
- **Login and security controls** - These controls provide user authentication.
- **Master pages** - These controls provide consistent layout and interface throughout the application.
- **Navigation controls** - These controls help in navigation. For example, menus, tree view etc.
- **Rich controls** - These controls implement special features. For example, AdRotator, FileUpload, and Calendar control.

The syntax for using server controls is:

```
<asp:controlType  ID ="ControlID" runat="server" Property1=value1
[Property2=value2] />
```

In addition, visual studio has the following features, to help produce in error-free coding:

- Dragging and dropping of controls in design view

- IntelliSense feature that displays and auto-completes the properties
- The properties window to set the property values directly

Properties of the Server Controls

ASP.NET server controls with a visual aspect are derived from the WebControl class and inherit all the properties, events, and methods of this class.

The WebControl class itself and some other server controls that are not visually rendered are derived from the System.Web.UI.Control class. For example, PlaceHolder control or XML control.

ASP.Net server controls inherit all properties, events, and methods of the WebControl and System.Web.UI.Control class.

The following table shows the inherited properties, common to all server controls:

Property	Description
AccessKey	Pressing this key with the Alt key moves focus to the control.
Attributes	It is the collection of arbitrary attributes (for rendering only) that do not correspond to properties on the control.
BackColor	Background color.
BindingContainer	The control that contains this control's data binding.
BorderColor	Border color.
BorderStyle	Border style.
BorderWidth	Border width.
CausesValidation	Indicates if it causes validation.
ChildControlCreated	It indicates whether the server control's child controls have been created.
ClientID	Control ID for HTML markup.
Context	The HttpContext object associated with the server control.
Controls	Collection of all controls contained within the control.

ControlStyle	The style of the Web server control.
CssClass	CSS class
DataItemContainer	Gets a reference to the naming container if the naming container implements IDataItemContainer.
DataKeysContainer	Gets a reference to the naming container if the naming container implements IDataKeysControl.
DesignMode	It indicates whether the control is being used on a design surface.
DisabledCssClass	Gets or sets the CSS class to apply to the rendered HTML element when the control is disabled.
Enabled	Indicates whether the control is grayed out.
EnableTheming	Indicates whether theming applies to the control.
EnableViewState	Indicates whether the view state of the control is maintained.
Events	Gets a list of event handler delegates for the control.
Font	Font.
Forecolor	Foreground color.
HasAttributes	Indicates whether the control has attributes set.
HasChildViewState	Indicates whether the current server control's child controls have any saved view-state settings.
Height	Height in pixels or %.
ID	Identifier for the control.
IsChildControlStateCleared	Indicates whether controls contained within this control have control state.
IsEnabled	Gets a value indicating whether the control is enabled.
IsTrackingViewState	It indicates whether the server control is saving changes to its view state.
IsViewStateEnabled	It indicates whether view state is enabled for this control.
LoadViewStateById	It indicates whether the control participates in loading its view state by ID instead of index.
Page	Page containing the control.

Parent	Parent control.
RenderingCompatibility	It specifies the ASP.NET version that the rendered HTML will be compatible with.
Site	The container that hosts the current control when rendered on a design surface.
SkinID	Gets or sets the skin to apply to the control.
Style	Gets a collection of text attributes that will be rendered as a style attribute on the outer tag of the Web server control.
TabIndex	Gets or sets the tab index of the Web server control.
TagKey	Gets the HtmlTextWriterTag value that corresponds to this Web server control.
TagName	Gets the name of the control tag.
TemplateControl	The template that contains this control.
TemplateSourceDirectory	Gets the virtual directory of the page or control containing this control.
ToolTip	Gets or sets the text displayed when the mouse pointer hovers over the web server control.
UniqueID	Unique identifier.
ViewState	Gets a dictionary of state information that saves and restores the view state of a server control across multiple requests for the same page.
ViewStateIgnoreCase	It indicates whether the StateBag object is case-insensitive.
ViewStateMode	Gets or sets the view-state mode of this control.
Visible	It indicates whether a server control is visible.
Width	Gets or sets the width of the Web server control.

4.6.1) BASIC CONTROLS

In this chapter, we will discuss the basic controls available in ASP.NET.

Button Controls

ASP.NET provides three types of button control:

- **Button** : It displays text within a rectangular area.
- **Link Button** : It displays text that looks like a hyperlink.
- **Image Button** : It displays an image.

When a user clicks a button, two events are raised: Click and Command.

Basic syntax of button control:

```
<asp:Button ID="Button1" runat="server" onclick="Button1_Click" Text="Click" / >
```

Common properties of the button control:

Property	Description
Text	The text displayed on the button. This is for button and link button controls only.
ImageUrl	For image button control only. The image to be displayed for the button.
AlternateText	For image button control only. The text to be displayed if the browser cannot display the image.
CausesValidation	Determines whether page validation occurs when a user clicks the button. The default is true.
CommandName	A string value that is passed to the command event when a user clicks the button.
CommandArgument	A string value that is passed to the command event when a user clicks the button.
PostBackUrl	The URL of the page that is requested when the user clicks the button.

Text Boxes and Labels

Text box controls are typically used to accept input from the user. A text box control can accept one or more lines of text depending upon the settings of the TextMode attribute.

Label controls provide an easy way to display text which can be changed from one execution of a page to the next. If you want to display text that does not change, you use the literal text.

Basic syntax of text control:

```
<asp:TextBox ID="txtstate" runat="server" ></asp:TextBox>
```

Common Properties of the Text Box and Labels:

Property	Description
TextMode	Specifies the type of text box. SingleLine creates a standard text box, MultiLIne creates a text box that accepts more than one line of text and the Password causes the characters that are entered to be masked. The default is SingleLine.
Text	The text content of the text box.
MaxLength	The maximum number of characters that can be entered into the text box.
Wrap	It determines whether or not text wraps automatically for multi-line text box; default is true.
ReadOnly	Determines whether the user can change the text in the box; default is false, i.e., the user can not change the text.
Columns	The width of the text box in characters. The actual width is determined based on the font that is used for the text entry.
Rows	The height of a multi-line text box in lines. The default value is 0, means a single line text box.

The mostly used attribute for a label control is 'Text', which implies the text displayed on the label.

Check Boxes and Radio Buttons

A check box displays a single option that the user can either check or uncheck and radio buttons present a group of options from which the user can select just one option.

To create a group of radio buttons, you specify the same name for the GroupName attribute of each radio button in the group. If more than one group is required in a single form, then specify a different group name for each group.

If you want check box or radio button to be selected when the form is initially displayed, set its Checked attribute to true. If the Checked attribute is set to true for multiple radio buttons in a group, then only the last one is considered as true.

Basic syntax of check box:

```
<asp:CheckBox ID= "chkoption" runat= "Server">
```

```
</asp:CheckBox>
```

Basic syntax of radio button:

```
<asp:RadioButton ID= "rdboption" runat= "Server">

</asp: RadioButton>
```

Common properties of check boxes and radio buttons:

Property	Description
Text	The text displayed next to the check box or radio button.
Checked	Specifies whether it is selected or not, default is false.
GroupName	Name of the group the control belongs to.

List Controls

ASP.NET provides the following controls

- Drop-down list,
- List box,
- Radio button list,
- Check box list,
- Bulleted list.

These control let a user choose from one or more items from the list. List boxes and drop-down lists contain one or more list items. These lists can be loaded either by code or by the ListItemCollection editor.

Basic syntax of list box control:

```
<asp:ListBox ID="ListBox1" runat="server" AutoPostBack="True"
OnSelectedIndexChanged="ListBox1_SelectedIndexChanged">

</asp:ListBox>
```

Basic syntax of drop-down list control:

```
<asp:DropDownList ID="DropDownList1" runat="server" AutoPostBack="True"
OnSelectedIndexChanged="DropDownList1_SelectedIndexChanged">

</asp:DropDownList>
```

Common properties of list box and drop-down Lists:

Property	Description
Items	The collection of ListItem objects that represents the items in the control. This property returns an object of type ListItemCollection.
Rows	Specifies the number of items displayed in the box. If actual list contains more rows than displayed then a scroll bar is added.
SelectedIndex	The index of the currently selected item. If more than one item is selected, then the index of the first selected item. If no item is selected, the value of this property is -1.
SelectedValue	The value of the currently selected item. If more than one item is selected, then the value of the first selected item. If no item is selected, the value of this property is an empty string ("").
SelectionMode	Indicates whether a list box allows single selections or multiple selections.

Common properties of each list item objects:

Property	Description
Text	The text displayed for the item.
Selected	Indicates whether the item is selected.
Value	A string value associated with the item.

It is important to notes that:

- To work with the items in a drop-down list or list box, you use the Items property of the control. This property returns a ListItemCollection object which contains all the items of the list.
- The SelectedIndexChanged event is raised when the user selects a different item from a drop-down list or list box.

The ListItemCollection

The ListItemCollection object is a collection of ListItem objects. Each ListItem object represents one item in the list. Items in a ListItemCollection are numbered from 0.

When the items into a list box are loaded using strings like: lstcolor.Items.Add("Blue"), then both the Text and Value properties of the list item are set to the string value you specify. To set it differently you must create a list item object and then add that item to the collection.

The ListItemCollection Editor is used to add item to a drop-down list or list box. This is used to create a static list of items. To display the collection editor, select edit item from the smart tag menu, or select the control and then click the ellipsis button from the Item property in the properties window.

Common properties of ListItemCollection:

Property	Description
Item(integer)	A ListItem object that represents the item at the specified index.
Count	The number of items in the collection.

Common methods of ListItemCollection:

Methods	Description
Add(string)	Adds a new item at the end of the collection and assigns the string parameter to the Text property of the item.
Add(ListItem)	Adds a new item at the end of the collection.
Insert(integer, string)	Inserts an item at the specified index location in the collection, and assigns string parameter to the text property of the item.
Insert(integer, ListItem)	Inserts the item at the specified index location in the collection.
Remove(string)	Removes the item with the text value same as the string.
Remove(ListItem)	Removes the specified item.
RemoveAt(integer)	Removes the item at the specified index as the integer.
Clear	Removes all the items of the collection.
FindByValue(string)	Returns the item whose value is same as the string.
FindByValue(Text)	Returns the item whose text is same as the string.

Radio Button list and Check Box list

A radio button list presents a list of mutually exclusive options. A check box list presents a list of independent options. These controls contain a collection of ListItem objects that could be referred to through the Items property of the control.

Basic syntax of radio button list:

```
<asp:RadioButtonList ID="RadioButtonList1" runat="server" AutoPostBack="True"
```

```
   OnSelectedIndexChanged="RadioButtonList1_SelectedIndexChanged">
</asp:RadioButtonList>
```

Basic syntax of check box list:

```
<asp:CheckBoxList ID="CheckBoxList1" runat="server" AutoPostBack="True"
   OnSelectedIndexChanged="CheckBoxList1_SelectedIndexChanged">
</asp:CheckBoxList>
```

Common properties of check box and radio button lists:

Property	Description
RepeatLayout	This attribute specifies whether the table tags or the normal html flow to use while formatting the list when it is rendered. The default is Table.
RepeatDirection	It specifies the direction in which the controls to be repeated. The values available are Horizontal and Vertical. Default is Vertical.
RepeatColumns	It specifies the number of columns to use when repeating the controls; default is 0.

Bulleted lists and Numbered lists

The bulleted list control creates bulleted lists or numbered lists. These controls contain a collection of ListItem objects that could be referred to through the Items property of the control.

Basic syntax of a bulleted list:

```
<asp:BulletedList ID="BulletedList1" runat="server">
</asp:BulletedList>
```

Common properties of the bulleted list:

Property	Description
BulletStyle	This property specifies the style and looks of the bullets, or numbers.
RepeatDirection	It specifies the direction in which the controls to be repeated. The values available are Horizontal and Vertical. Default is Vertical.
RepeatColumns	It specifies the number of columns to use when repeating the controls; default is 0.

HyperLink Control

The HyperLink control is like the HTML <a> element.

Basic syntax for a hyperlink control:

```
<asp:HyperLink ID="HyperLink1" runat="server">
   HyperLink
</asp:HyperLink>
```

It has the following important properties:

Property	Description
ImageUrl	Path of the image to be displayed by the control.
NavigateUrl	Target link URL.
Text	The text to be displayed as the link.
Target	The window or frame which loads the linked page.

Image Control

The image control is used for displaying images on the web page, or some alternative text, if the image is not available.

Basic syntax for an image control:

```
<asp:Image ID="Image1" runat="server">
```

It has the following important properties:

Property	Description
AlternateText	Alternate text to be displayed in absence of the image.
ImageAlign	Alignment options for the control.
ImageUrl	Path of the image to be displayed by the control.

4.6.2) CALENDAR

The calendar control is a functionally rich web control, which provides the following capabilities:

- Displaying one month at a time
- Selecting a day, a week or a month
- Selecting a range of days
- Moving from month to month

- Controlling the display of the days programmatically

It is used to display selectable date in a calendar. It also shows data associated with specific date. This control displays a calendar through which users can move to any day in any year.

We can also set Selected Date property that shows specified date in the calendar.

To create **Calendar** we can drag it from the toolbox of visual studio.

The basic syntax of a calendar control is:

```
<asp:Calender ID = "Calendar1" runat = "server">
</asp:Calender>
```

This is a server side control and ASP.NET provides own tag to create it. The example is given below.

```
< asp:CalendarID="Calendar1" runat="server" SelectedDate="2017-06-15" ></asp:Calendar>
```

Server renders it as the HTML control and produces the following code to the browser:

```
<table id="Calendar1" cellspacing="0" cellpadding="2" title="Calendar"
style="border-width:1px;border-style:solid;border-collapse:collapse;">
<tr><td colspan="7" style="background-color:Silver;">
<table cellspacing="0" style="width:100%;border-collapse:collapse;">
<tr><td style="width:15%;">
<a href="javascript:__doPostBack('Calendar1','V6330')"
style="color:Black" title="Go to the previous month"></a> ...
```

Properties and Events of the Calendar Control

The calendar control has many properties and events, using which you can customize the actions and display of the control. The following table provides some important properties of the Calendar control:

Properties	Description
Caption	Gets or sets the caption for the calendar control.
CaptionAlign	Gets or sets the alignment for the caption.
CellPadding	Gets or sets the number of spaces between the data and the cell border.
CellSpacing	Gets or sets the space between cells.
DayHeaderStyle	Gets the style properties for the section that displays the day of the week.

DayNameFormat	Gets or sets format of days of the week.
DayStyle	Gets the style properties for the days in the displayed month.
FirstDayOfWeek	Gets or sets the day of week to display in the first column.
NextMonthText	Gets or sets the text for next month navigation control. The default value is >.
NextPrevFormat	Gets or sets the format of the next and previous month navigation control.
OtherMonthDayStyle	Gets the style properties for the days on the Calendar control that are not in the displayed month.
PrevMonthText	Gets or sets the text for previous month navigation control. The default value is <.
SelectedDate	Gets or sets the selected date.
SelectedDates	Gets a collection of DateTime objects representing the selected dates.
SelectedDayStyle	Gets the style properties for the selected dates.
SelectionMode	Gets or sets the selection mode that specifies whether the user can select a single day, a week or an entire month.
SelectMonthText	Gets or sets the text for the month selection element in the selector column.
SelectorStyle	Gets the style properties for the week and month selector column.
SelectWeekText	Gets or sets the text displayed for the week selection element in the selector column.
ShowDayHeader	Gets or sets the value indicating whether the heading for the days of the week is displayed.
ShowGridLines	Gets or sets the value indicating whether the gridlines would be shown.
ShowNextPrevMonth	Gets or sets a value indicating whether next and previous month navigation elements are shown in the title section.
ShowTitle	Gets or sets a value indicating whether the title section is displayed.
TitleFormat	Gets or sets the format for the title section.
Titlestyle	Get the style properties of the title heading for the Calendar control.

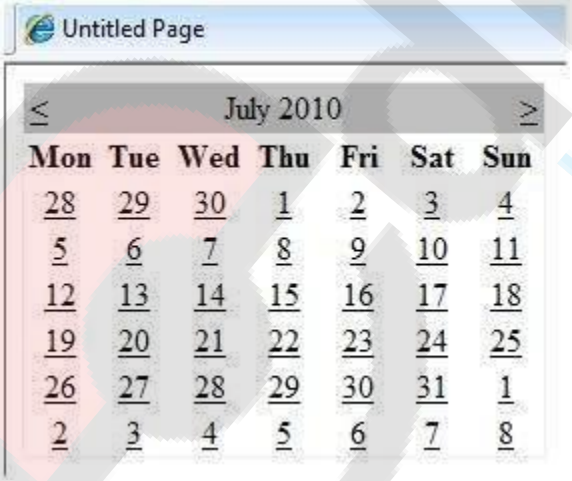
TodayDayStyle	Gets the style properties for today's date on the Calendar control.
TodaysDate	Gets or sets the value for today's date.
UseAccessibleHeader	Gets or sets a value that indicates whether to render the table header <th> HTML element for the day headers instead of the table data <td> HTML element.
VisibleDate	Gets or sets the date that specifies the month to display.
WeekendDayStyle	Gets the style properties for the weekend dates on the Calendar control.

The Calendar control has the following three most important events that allow the developers to program the calendar control. They are:

Events	Description
SelectionChanged	It is raised when a day, a week or an entire month is selected.
DayRender	It is raised when each data cell of the calendar control is rendered.
VisibleMonthChanged	It is raised when user changes a month.

Working with the Calendar Control

Putting a bare-bone calendar control without any code behind file provides a workable calendar to a site, which shows the months and days of the year. It also allows navigation to next and previous months.



Calendar controls allow the users to select a single day, a week, or an entire month. This is done by using the SelectionMode property. This property has the following values:

Properties	Description

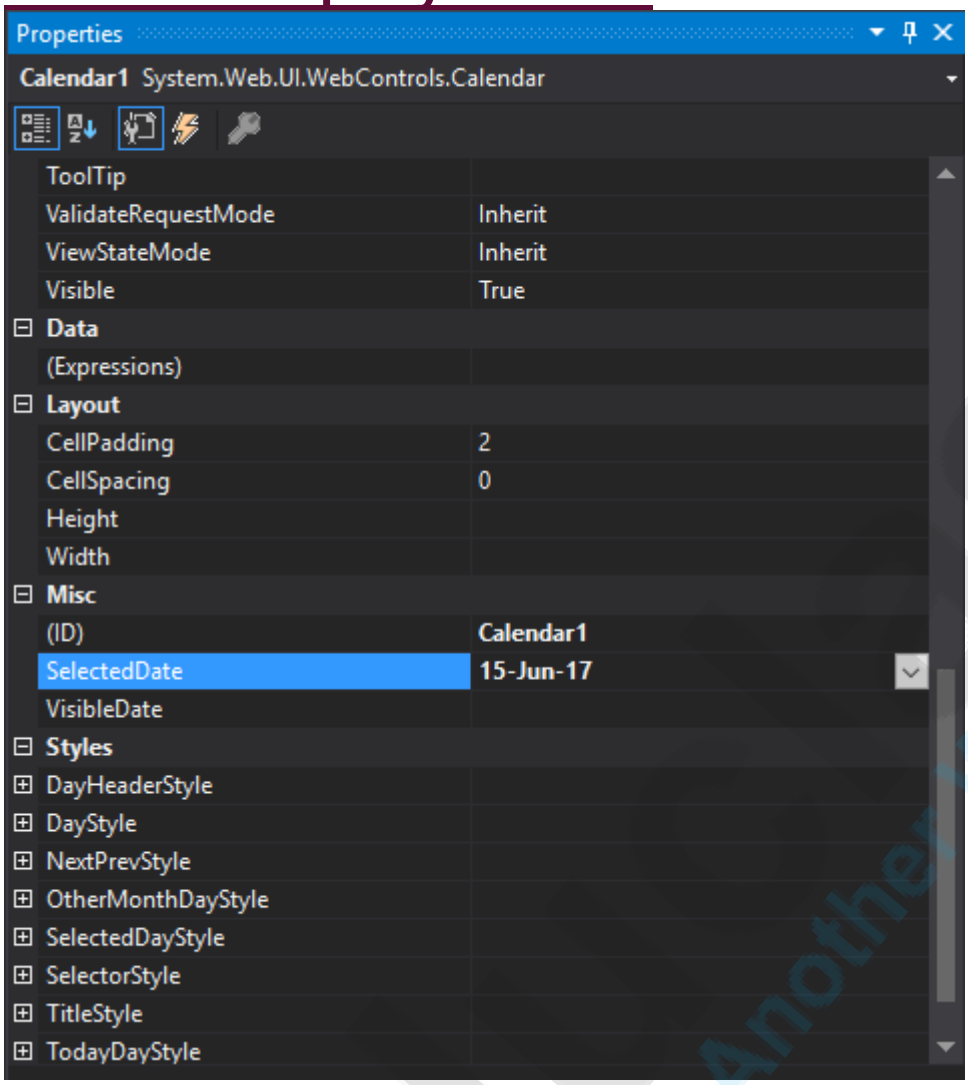
Day	To select a single day.
DayWeek	To select a single day or an entire week.
DayWeekMonth	To select a single day, a week, or an entire month.
None	Nothing can be selected.

The syntax for selecting days:

```
<asp:Calender ID = "Calendar1" runat = "server" SelectionMode="DayWeekMonth">

</asp:Calender>
```

When the selection mode is set to the value DayWeekMonth, an extra column with the > symbol appears for selecting the week, and a >> symbol appears to the left of the days name for selecting the month.

Calendar Property Window



Example

In this example, we are implementing calendar and displaying user selected date to the web page.

// WebControls.aspx

```
<%@ Page Language="C#" AutoEventWireup="true" CodeBehind="WebControls.aspx.cs
"
Inherits="WebFormsControlls.WebControls" %>
<!DOCTYPE html>
<html xmlns="http://www.w3.org/1999/xhtml">
<head runat="server">
  <title></title>
</head>
<body>
  <form id="form1" runat="server">
    <h2>Select Date from the Calender</h2>
```

```
    <div>
      <asp:Calendar ID="Calendar1" runat="server"
      OnSelectionChanged="Calendar1_SelectionChanged"></asp:Calendar>
    </div>
  </form>
  <p>
    <asp:Label runat="server" ID="ShowDate" ></asp:Label>
  </p>
</body>
</html>
```

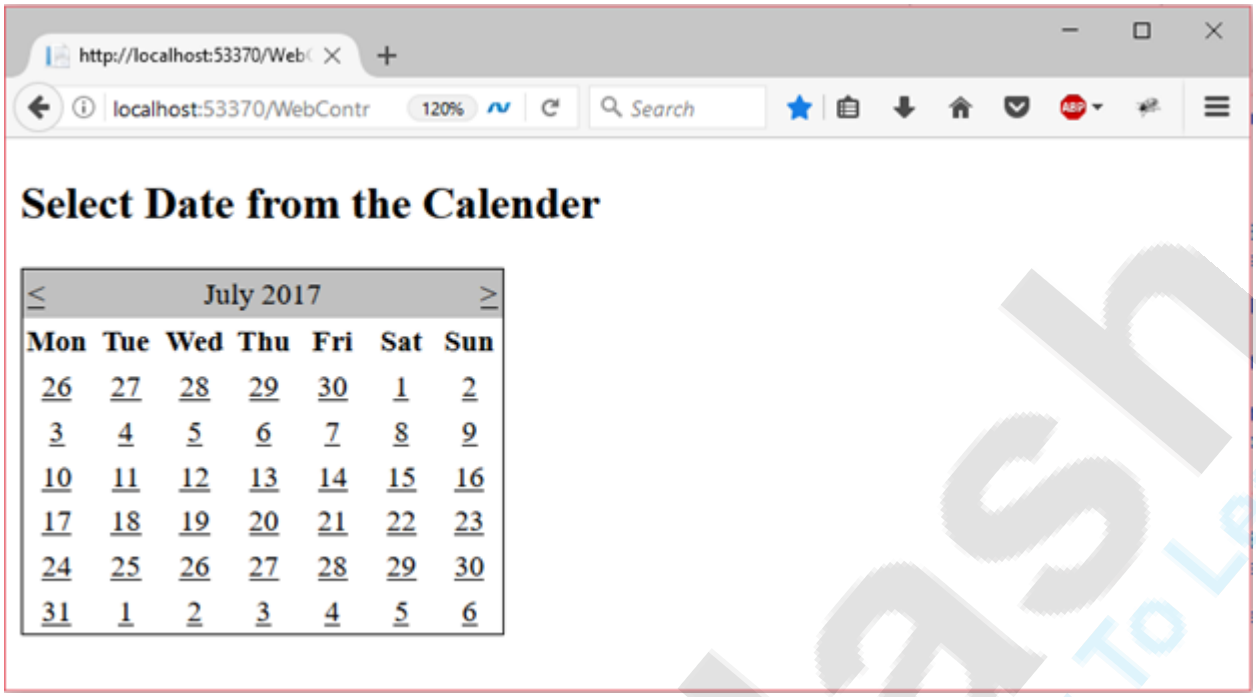
Code Behind

// WebControls.aspx.cs

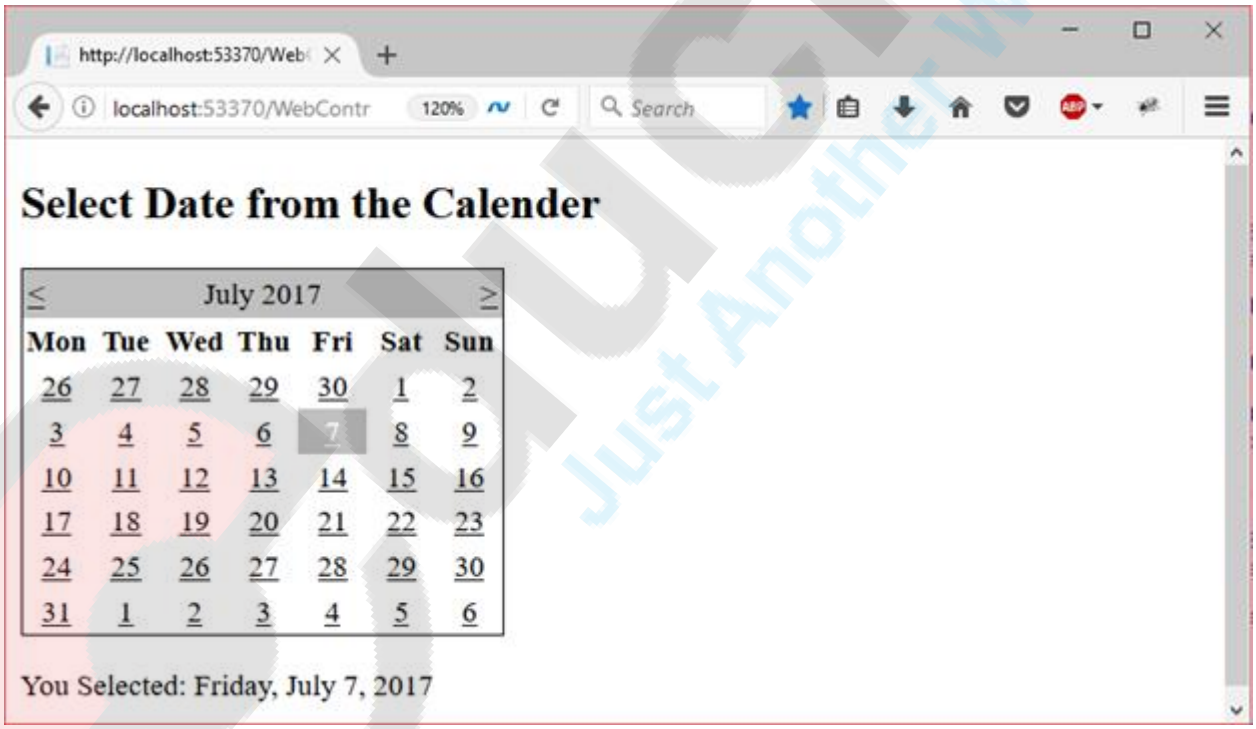
```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Web;
using System.Web.UI;
using System.Web.UI.WebControls;
namespace WebFormsControlls
{
  public partial class WebControls : System.Web.UI.Page
  {
    public void Calendar1_SelectionChanged(object sender, EventArgs e)
    {
      ShowDate.Text = "You Selected: "+Calendar1.SelectedDate.ToString("D");
    }
  }
}
```

Output:

This view shows calendar to the browser.



It shows date selected by the user at the web page. A screenshot is attached below.



4.6.3) AD ROTATOR

- The AdRotator control randomly selects banner graphics from a list, which is specified in an external XML schedule file. This external XML schedule file is called the advertisement file.
- The AdRotator control allows you to specify the advertisement file and the type of window that the link should follow in the AdvertisementFile and the Target property respectively.

The basic syntax of adding an AdRotator is as follows:

```
<asp:AdRotator  runat = "server" AdvertisementFile = "adfile.xml"  Target =
"_blank" />
```

Before going into the details of the AdRotator control and its properties, let us look into the construction of the advertisement file.

The Advertisement File

The advertisement file is an XML file, which contains the information about the advertisements to be displayed.

Extensible Markup Language (XML) is a W3C standard for text document markup. It is a text-based markup language that enables you to store data in a structured format by using meaningful tags. The term 'extensible' implies that you can extend your ability to describe a document by defining meaningful tags for the application.

XML is not a language in itself, like HTML, but a set of rules for creating new markup languages. It is a meta-markup language. It allows developers to create custom tag sets for special uses. It structures, stores, and transports the information.

Following is an example of XML file:

```
<BOOK>
   <NAME> Learn XML </NAME>
   <AUTHOR> Samuel Peterson </AUTHOR>
   <PUBLISHER> NSS Publications </PUBLISHER>
   <PRICE> $30.00</PRICE>
</BOOK>
```

Like all XML files, the advertisement file needs to be a structured text file with well-defined tags delineating the data. There are the following standard XML elements that are commonly used in the advertisement file:

Element	Description
Advertisements	Encloses the advertisement file.
Ad	Delineates separate ad.
ImageUrl	The path of image that will be displayed.
NavigateUrl	The link that will be followed when the user clicks the ad.
AlternateText	The text that will be displayed instead of the picture if it cannot be displayed.
Keyword	Keyword identifying a group of advertisements. This is used for filtering.
Impressions	The number indicating how often an advertisement will appear.
Height	Height of the image to be displayed.
Width	Width of the image to be displayed.

Apart from these tags, customs tags with custom attributes could also be included. The following code illustrates an advertisement file ads.xml:

```
<Advertisements>
   <Ad>
      <ImageUrl>rose1.jpg</ImageUrl>
      <NavigateUrl>http://www.1800flowers.com</NavigateUrl>
      <AlternateText>
         Order flowers, roses, gifts and more
      </AlternateText>
      <Impressions>20</Impressions>
      <Keyword>flowers</Keyword>
   </Ad>

   <Ad>
      <ImageUrl>rose2.jpg</ImageUrl>
      <NavigateUrl>http://www.babybouquets.com.au</NavigateUrl>
      <AlternateText>Order roses and flowers</AlternateText>
      <Impressions>20</Impressions>
      <Keyword>gifts</Keyword>
   </Ad>

   <Ad>
      <ImageUrl>rose3.jpg</ImageUrl>
      <NavigateUrl>http://www.flowers2moscow.com</NavigateUrl>
      <AlternateText>Send flowers to Russia</AlternateText>
      <Impressions>20</Impressions>
      <Keyword>russia</Keyword>
   </Ad>
```

```
   <Ad>
      <ImageUrl>rose4.jpg</ImageUrl>
      <NavigateUrl>http://www.ediblebloom s.com</NavigateUrl>
      <AlternateText>Edible Blooms</AlternateText>
      <Impressions>20</Impressions>
      <Keyword>gifts</Keyword>
   </Ad>
</Advertisements>
```

Properties and Events of the AdRotator Class

The AdRotator class is derived from the WebControl class and inherits its properties. Apart from those, the AdRotator class has the following properties:

Properties	Description
AdvertisementFile	The path to the advertisement file.
AlternateTextFeild	The element name of the field where alternate text is provided. The default value is AlternateText.
DataMember	The name of the specific list of data to be bound when advertisement file is not used.
DataSource	Control from where it would retrieve data.
DataSourceID	Id of the control from where it would retrieve data.
Font	Specifies the font properties associated with the advertisement banner control.
ImageUrlField	The element name of the field where the URL for the image is provided. The default value is ImageUrl.
KeywordFilter	For displaying the keyword based ads only.
NavigateUrlField	The element name of the field where the URL to navigate to is provided. The default value is NavigateUrl.
Target	The browser window or frame that displays the content of the page linked.
UniqueID	Obtains the unique, hierarchically qualified identifier for the AdRotator control.

Following are the important events of the AdRotator class:

Events	Description

AdCreated	It is raised once per round trip to the server after creation of the control, but before the page is rendered
DataBinding	Occurs when the server control binds to a data source.
DataBound	Occurs after the server control binds to a data source.
Disposed	Occurs when a server control is released from memory, which is the last stage of the server control lifecycle when an ASP.NET page is requested
Init	Occurs when the server control is initialized, which is the first step in its lifecycle.
Load	Occurs when the server control is loaded into the Page object.
PreRender	Occurs after the Control object is loaded but prior to rendering.
Unload	Occurs when the server control is unloaded from memory.

Working with AdRotator Control

Create a new web page and place an AdRotator control on it.

```
<form id="form1" runat="server">

   <div>

      <asp:AdRotator ID="AdRotator1" runat="server" AdvertisementFile
="~/ads.xml" onadcreated="AdRotator1_AdCreated" />

   </div>

</form>
```

The ads.xml file and the image files should be located in the root directory of the web site.

Try to execute the above application and observe that each time the page is reloaded, the ad is changed.

4.6.4) FILEUPLOAD

ASP.NET has two controls that allow users to upload files to the web server. Once the server receives the posted file data, the application can save it, check it, or ignore it. The following controls allow the file uploading:

- **HtmlInputFile** - an HTML server control
- **FileUpload** - and ASP.NET web control

Both controls allow file uploading, but the FileUpload control automatically sets the encoding of the form, whereas the HtmlInputFile does not do so.

It is an input controller which is used to upload file to the server. It creates a browse button on the form that pop up a window to select the file from the local machine.

The FileUpload control allows the user to browse for and select the file to be uploaded, providing a browse button and a text box for entering the filename.

Once, the user has entered the filename in the text box by typing the name or browsing, the SaveAs method of the FileUpload control can be called to save the file to the disk.

The basic syntax of FileUpload is:

```
<asp:FileUpload ID= "Uploader" runat = "server" />
```

The FileUpload class is derived from the WebControl class, and inherits all its members. Apart from those, the FileUpload class has the following read-only properties:

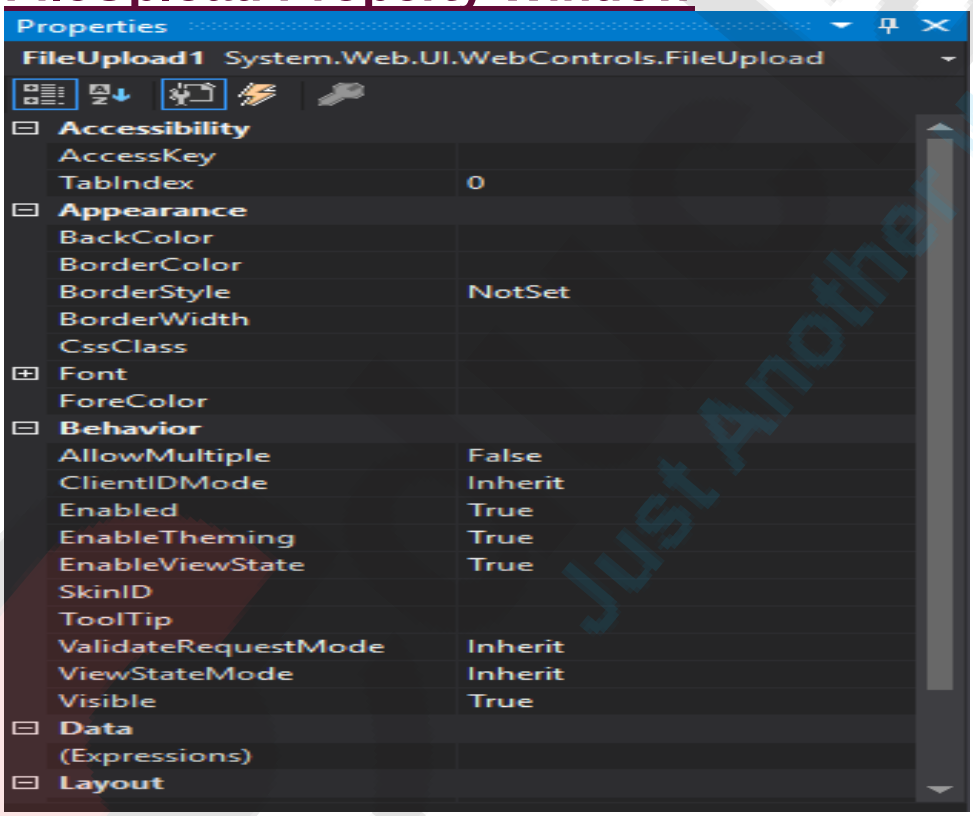
Properties	Description
FileBytes	Returns an array of the bytes in a file to be uploaded.
FileContent	Returns the stream object pointing to the file to be uploaded.
FileName	Returns the name of the file to be uploaded.
HasFile	Specifies whether the control has a file to upload.
PostedFile	Returns a reference to the uploaded file.

The posted file is encapsulated in an object of type HttpPostedFile, which could be accessed through the PostedFile property of the FileUpload class.

The HttpPostedFile class has the following frequently used properties:

Properties	Description
ContentLength	Returns the size of the uploaded file in bytes.
ContentType	Returns the MIME type of the uploaded file.
FileName	Returns the full filename.
InputStream	Returns a stream object pointing to the uploaded file.

FileUpload Property Window



Example

Here, we are implementing file upload control in web form.

// WebControls.aspx

<%@ Page Language="C#" AutoEventWireup="true" CodeBehind="WebControls.aspx.cs
"

```
Inherits="WebFormsControlls.WebControls" %>
<!DOCTYPE html>
<html xmlns="http://www.w3.org/1999/xhtml">
<head runat="server">
  <title></title>
</head>
<body>
  <form id="form1" runat="server">
    <div>
      <p>Browse to Upload File</p>
      <asp:FileUpload ID="FileUpload1" runat="server" />
    </div>
    <p>
      <asp:Button ID="Button1" runat="server" Text="Upload File" OnClick="Button1
_Click" />
    </p>
  </form>
  <p>
    <asp:Label runat="server" ID="FileUploadStatus"></asp:Label>
  </p>
</body>
</html>
```

Code

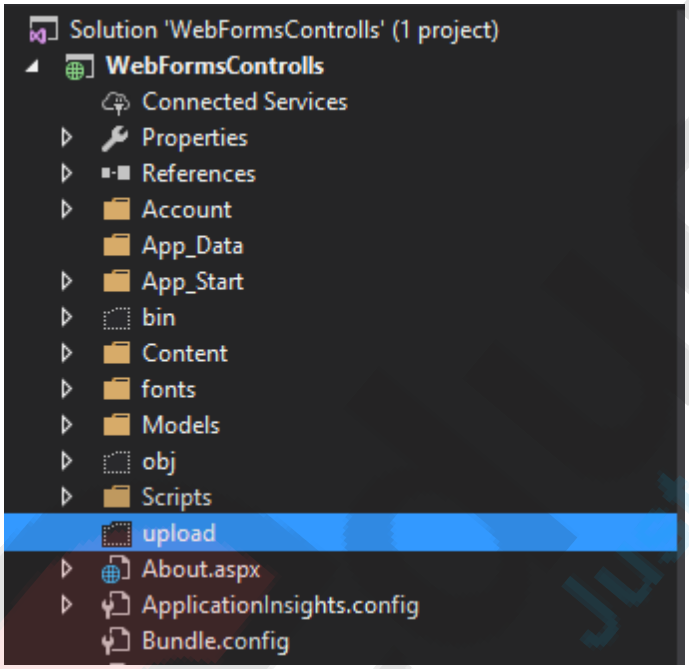
// WebControls.aspx.cs

```
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 System.Web.UI.HtmlControls.HtmlInputFile File1;
    protected System.Web.UI.HtmlControls.HtmlInputButton Submit1;
    protected void Page_Load(object sender, EventArgs e)
    {
    }
    protected void Button1_Click(object sender, EventArgs e)
    {
      if ((FileUpload1.PostedFile != null) && (FileUpload1.PostedFile.ContentLength > 0))
      {
        string fn = System.IO.Path.GetFileName(FileUpload1.PostedFile.FileName);
        string SaveLocation = Server.MapPath("upload") + "\\" + fn;
        try
        {
          FileUpload1.PostedFile.SaveAs(SaveLocation);
```

```
            FileUploadStatus.Text = "The file has been uploaded.";
          }
          catch (Exception ex)
          {
            FileUploadStatus.Text = "Error: " + ex.Message;
          }
        }
        else
        {
          FileUploadStatus.Text = "Please select a file to upload.";
        }
      }
    }
}
```

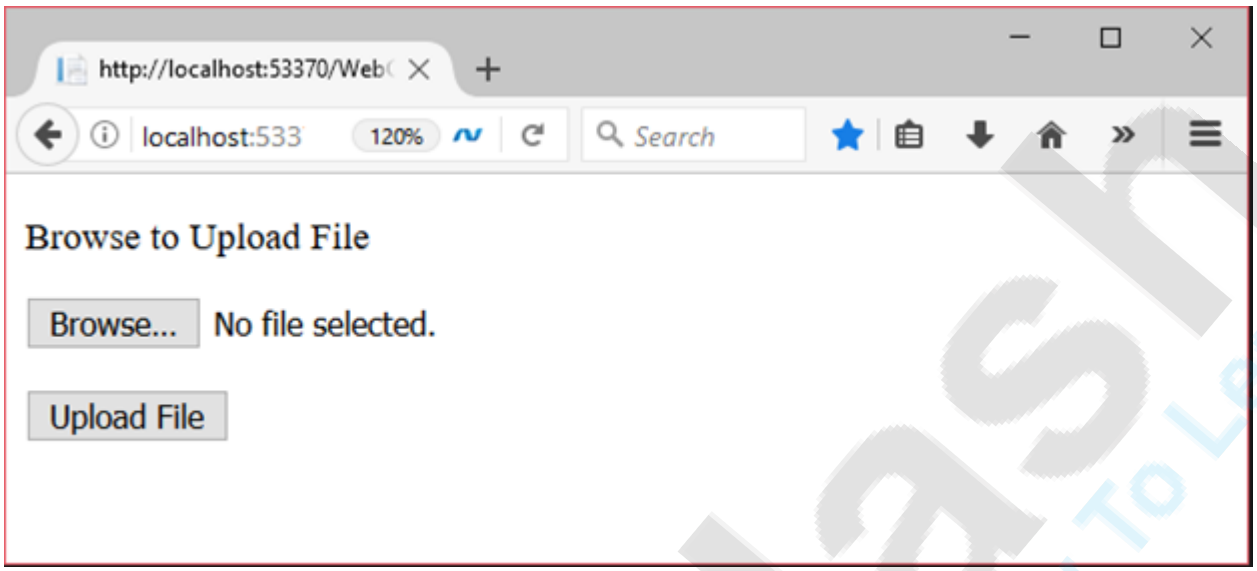
Create a directory into the project to store uploaded files as we did in below screen shoot.

Output:

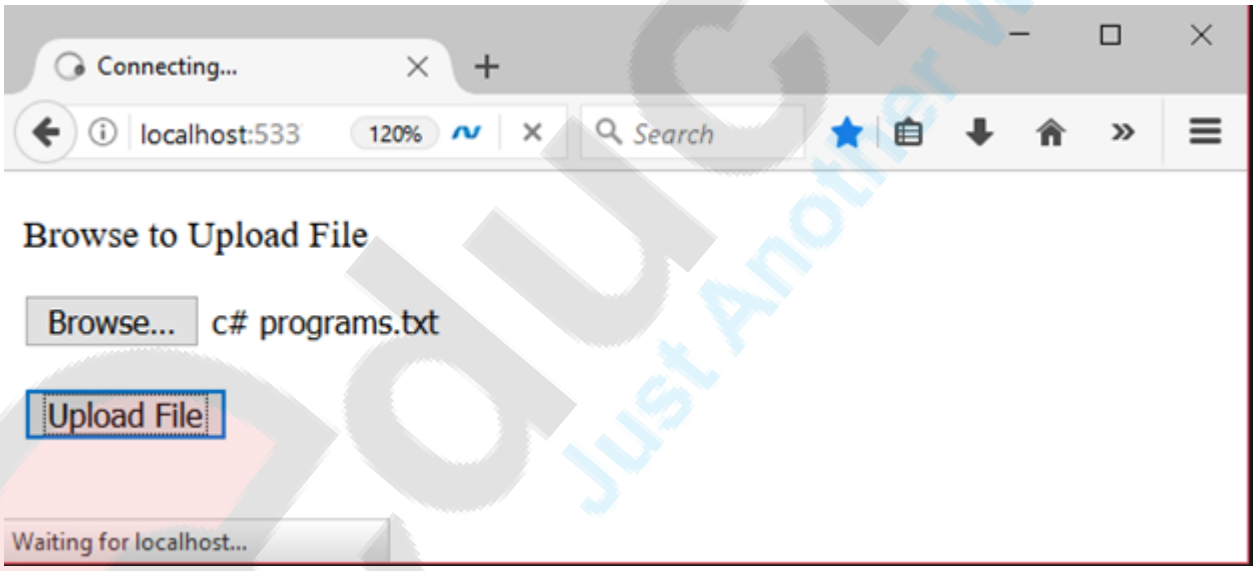


Output:

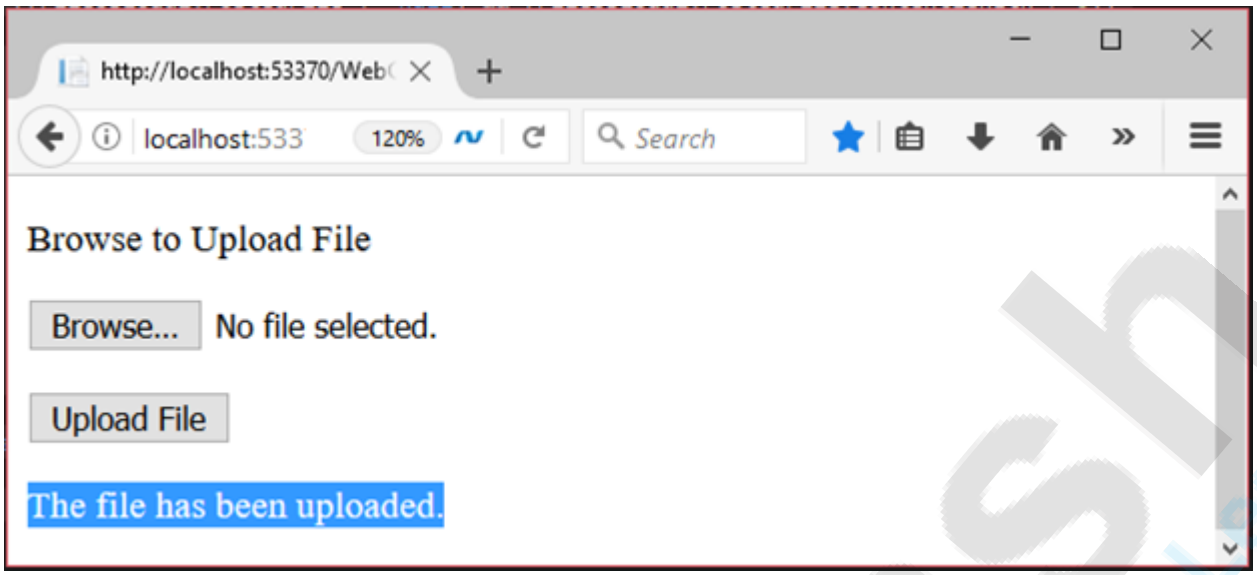
Run the code, it produces following output.



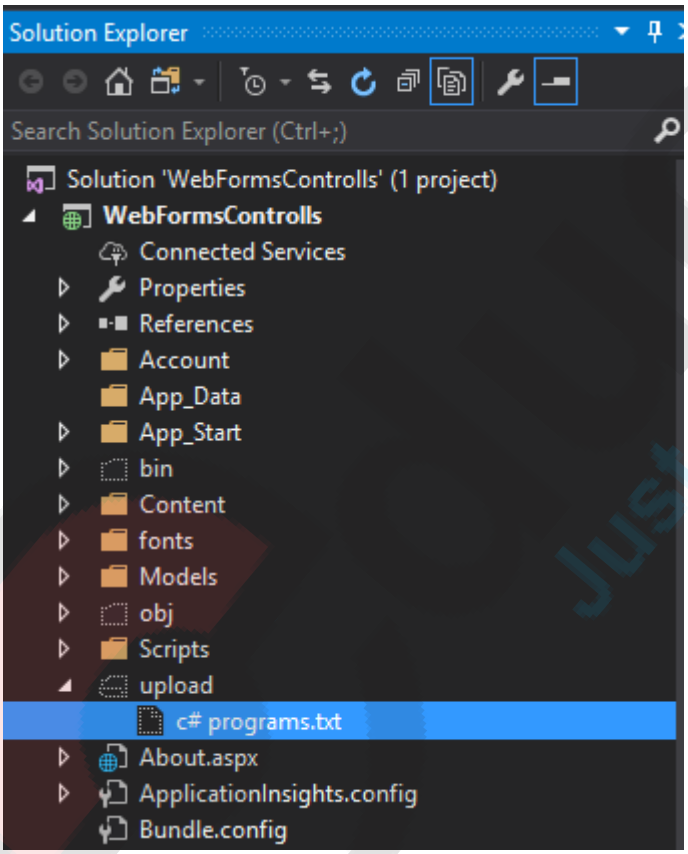
We are uploading a file **c# programs.txt.**



It displays a successful file uploaded message after uploading as shown in the following screenshot.



The file is stored into **upload** folder. Look inside the folder, it shows the uploaded file is present.



4.6.5) VALIDATION CONTROLS

ASP.NET validation controls validate the user input data to ensure that useless, unauthenticated, or contradictory data don't get stored.

ASP.NET provides the following validation controls:

- RequiredFieldValidator
- RangeValidator
- CompareValidator
- RegularExpressionValidator
- ValidationSummary

BaseValidator Class

The validation control classes are inherited from the BaseValidator class hence they inherit its properties and methods. Therefore, it would help to take a look at the properties and the methods of this base class, which are common for all the validation controls:

Members	Description
ControlToValidate	Indicates the input control to validate.
Display	Indicates how the error message is shown.
EnableClientScript	Indicates whether client side validation will take.
Enabled	Enables or disables the validator.
ErrorMessage	Indicates error string.
Text	Error text to be shown if validation fails.
IsValid	Indicates whether the value of the control is valid.
SetFocusOnError	It indicates whether in case of an invalid control, the focus should switch to the related input control.
ValidationGroup	The logical group of multiple validators, where this control belongs.
Validate()	This method revalidates the control and updates the IsValid property.

1) ASP.NET RequiredFieldValidator Control

This validator is used to make an input control required. It will throw an error if user leaves input control empty.

It is used to mandate form control required and restrict the user to provide data.

Note: It removes extra spaces from the beginning and end of the input value before validation is performed.

The ControlToValidateproperty should be set with the ID of control to validate.

RequiredFieldValidator Properties

Property	Description
AccessKey	It is used to set keyboard shortcut for the control.
BackColor	It is used to set background color of the control.
BorderColor	It is used to set border color of the control.
Font	It is used to set font for the control text.
ForeColor	It is used to set color of the control text.
Text	It is used to set text to be shown for the control.
ToolTip	It displays the text when mouse is over the control.
Visible	To set visibility of control on the form.
Height	It is used to set height of the control.
Width	It is used to set width of the control.
ErrorMessage	It is used to set error message that display when validation fails.
ControlToValidate	It takes ID of control to validate.

Example

Here, in the following example, we are explaining **RequiredFieldValidator** control and creating to mandatory TextBox controls.

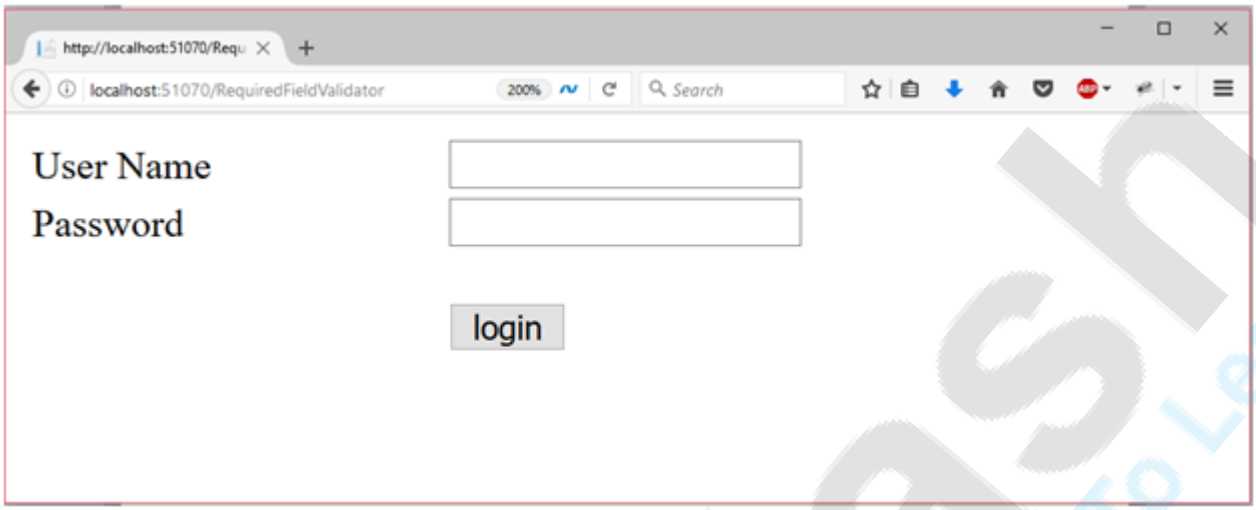
// RequiredFieldValidator.aspx

<%@ Page Language="C#" AutoEventWireup="true" CodeBehind="RequiredFieldValidator.aspx.cs"

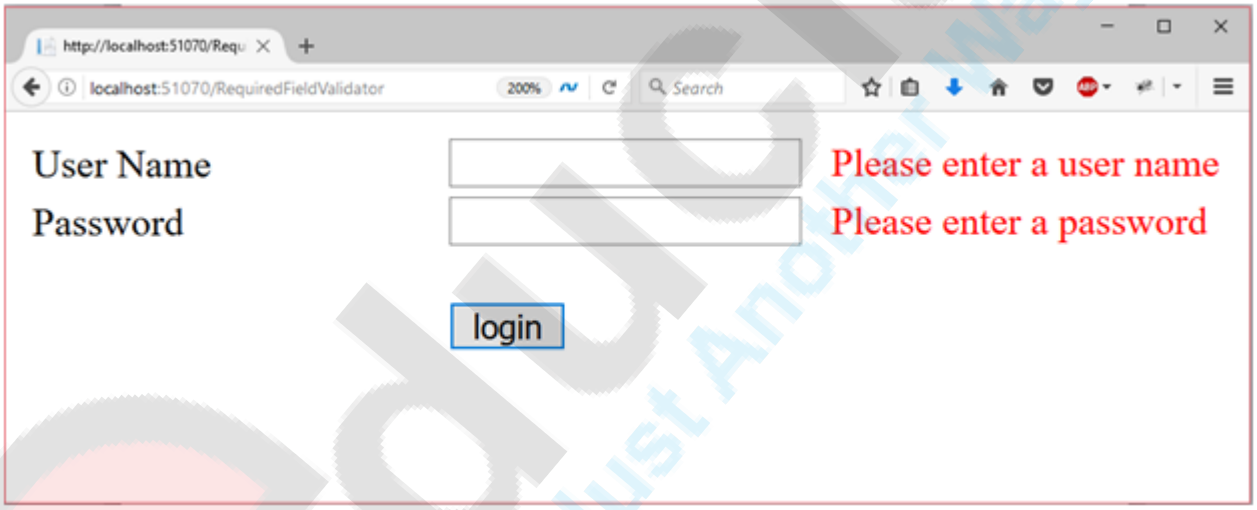
```
Inherits="asp.netexample.RequiredFieldValidator" %>
<!DOCTYPE html>
<html xmlns="http://www.w3.org/1999/xhtml">
<head runat="server">
<title></title>
<style type="text/css">
.auto-style1 {
width: 100%;
    }
.auto-style2 {
width: 165px;
    }
</style>
</head>
<body>
<form id="form1" runat="server">
<div>
</div>
<table class="auto-style1">
<tr>
<td class="auto-style2">User Name</td>
<td>
<asp:TextBox ID="username" runat="server"></asp:TextBox>
<asp:RequiredFieldValidatorID asp:RequiredFieldValidatorID="user" runat="server" Co
ntrolToValidate="username"
ErrorMessage="Please enter a user name" ForeColor="Red"></asp:RequiredFieldVali
dator>
</td>
</tr>
<tr>
<td class="auto-style2">Password</td>
<td>
<asp:TextBox ID="password" runat="server"></asp:TextBox>
<asp:RequiredFieldValidator ID="pass" runat="server" ControlToValidate="password"
ErrorMessage="Please enter a password"
ForeColor="Red"></asp:RequiredFieldValidator>
</td>
</tr>
<tr>
<td class="auto-style2">&#65533;</td>
<td>
<br/>
<asp:Button ID="Button1" runat="server" Text="login"/>
</td>
</tr>
</table>
</form>
</body>
</html>
```

Output:

It produces the following output when view in the browser.



It throws error messages when user login with empty controls.



2) ASP.NET RangeValidator Control

This validator evaluates the value of an input control to check that the value lies between specified ranges.

It allows us to check whether the user input is between a specified upper and lower boundary. This range can be numbers, alphabetic characters and dates.

Note: if the input control is empty, no validation will be performed.

The **ControlToValidate**property is used to specify the control to validate. The **MinimumValue** and **MaximumValue** properties are used to set minimum and maximum boundaries for the control.

RangeValidator Properties

Property	Description
AccessKey	It is used to set keyboard shortcut for the control.
TabIndex	The tab order of the control.
BackColor	It is used to set background color of the control.
BorderColor	It is used to set border color of the control.
BorderWidth	It is used to set width of border of the control.
Font	It is used to set font for the control text.
ForeColor	It is used to set color of the control text.
Text	It is used to set text to be shown for the control.
ToolTip	It displays the text when mouse is over the control.
Visible	To set visibility of control on the form.
Height	It is used to set height of the control.
Width	It is used to set width of the control.
ControlToValidate	It takes ID of control to validate.
ErrorMessage	It is used to display error message when validation failed.
Type	It is used to set datatype of the control value.
MaximumValue	It is used to set upper boundary of the range.
MinimumValue	It is used to set lower boundary of the range.

Example

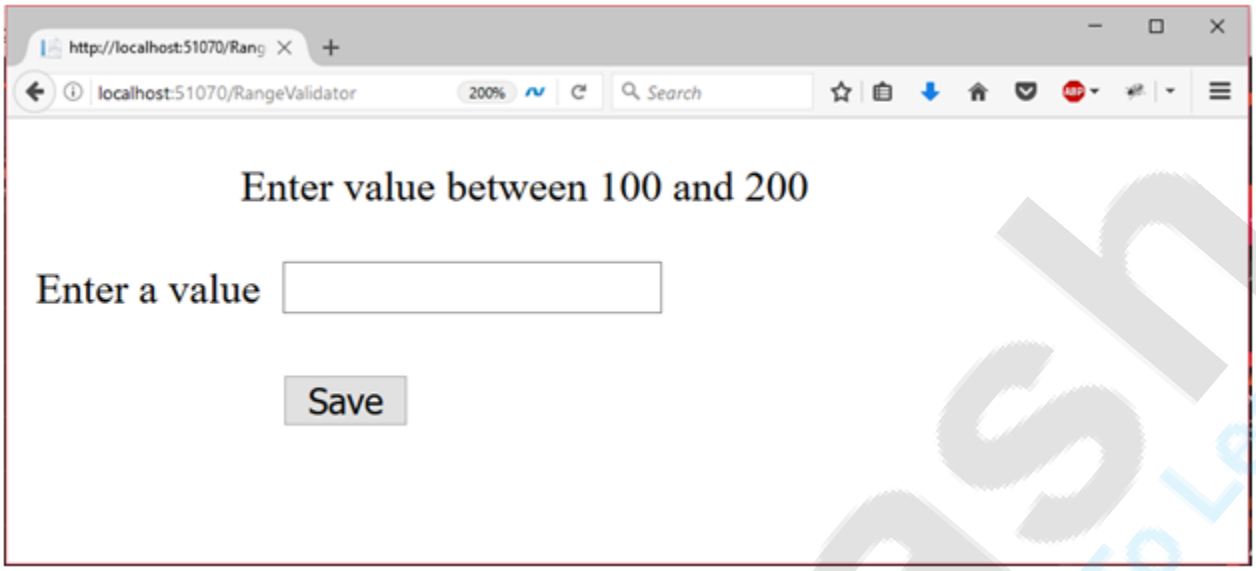
In the following example, we are using **RangeValidator** to validate user input in specified range.

// RangeValidator.aspx

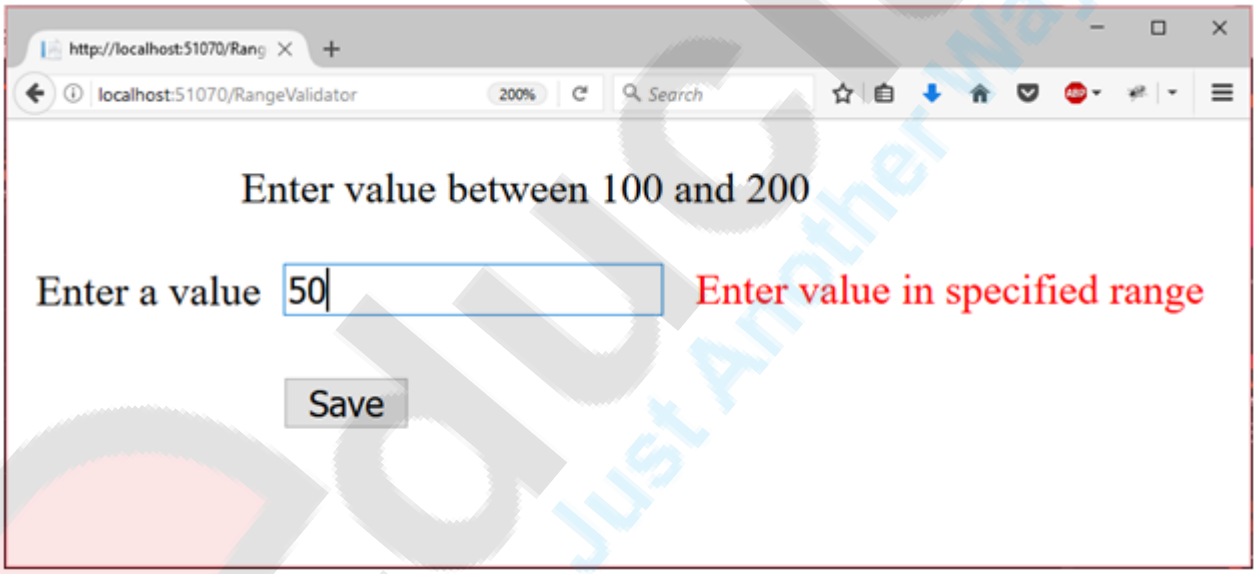
```
<%@ Page Language="C#" AutoEventWireup="true" CodeBehind="RangeValidator.aspx.cs"

Inherits="asp.netexample.RangeValidator" %>
<!DOCTYPE html>
<html xmlns="http://www.w3.org/1999/xhtml">
<head runat="server">
<title></title>
```

```
<style type="text/css">
.auto-style1 {
height: 82px;
    }
.auto-style2 {
width: 100%;
    }
.auto-style3 {
width: 89px;
    }
.auto-style4 {
margin-left: 80px;
    }
</style>
</head>
<body>
<form id="form1" runat="server">
<div class="auto-style1">
<p class="auto-style4">
    Enter value between 100 and 200<br/>
</p>
<table class="auto-style2">
<tr>
<td class="auto-style3">
<asp:Label ID="Label2" runat="server" Text="Enter a value"></asp:Label>
</td>
<td>
<asp:TextBox ID="uesrInput"runat="server"></asp:TextBox>
<asp:RangeValidator ID="RangeValidator1" runat="server" ControlToValidate="uesrInp
ut"
ErrorMessage="Enter value in specified range" ForeColor="Red" MaximumValue="199" Min
imumValue="101"
SetFocusOnError="True"Type=" Integer"></asp:RangeValidator>
</td>
</tr>
<tr>
<td class="auto-style3">?</td>
<td>
<br/>
<asp:Button ID="Button2" runat="server" Text="Save"/>
</td>
</tr>
</table>
<br/>
<br/>
</div>
</form>
</body>
</html>
```

Output:



It throws an error message when the input is not in range.



3) ASP.NET CompareValidator Control

This validator evaluates the value of an input control against another input control on the basis of specified operator.

We can use comparison operators like: less than, equal to, greater than etc.

Note: If the input filed is empty, no validation will be performed.

CompareValidator Properties

Property	Description

AccessKey	It is used to set keyboard shortcut for the control.
TabIndex	The tab order of the control.
BackColor	It is used to set background color of the control.
BorderColor	It is used to set border color of the control.
BorderWidth	It is used to set width of border of the control.
Font	It is used to set font for the control text.
ForeColor	It is used to set color of the control text.
Text	It is used to set text to be shown for the control.
ToolTip	It displays the text when mouse is over the control.
Visible	To set visibility of control on the form.
Height	It is used to set height of the control.
Width	It is used to set width of the control.
ControlToCompare	It takes ID of control to compare with.
ControlToValidate	It takes ID of control to validate.
ErrorMessage	It is used to display error message when validation failed.
Operator	It is used set comparison operator.

Example

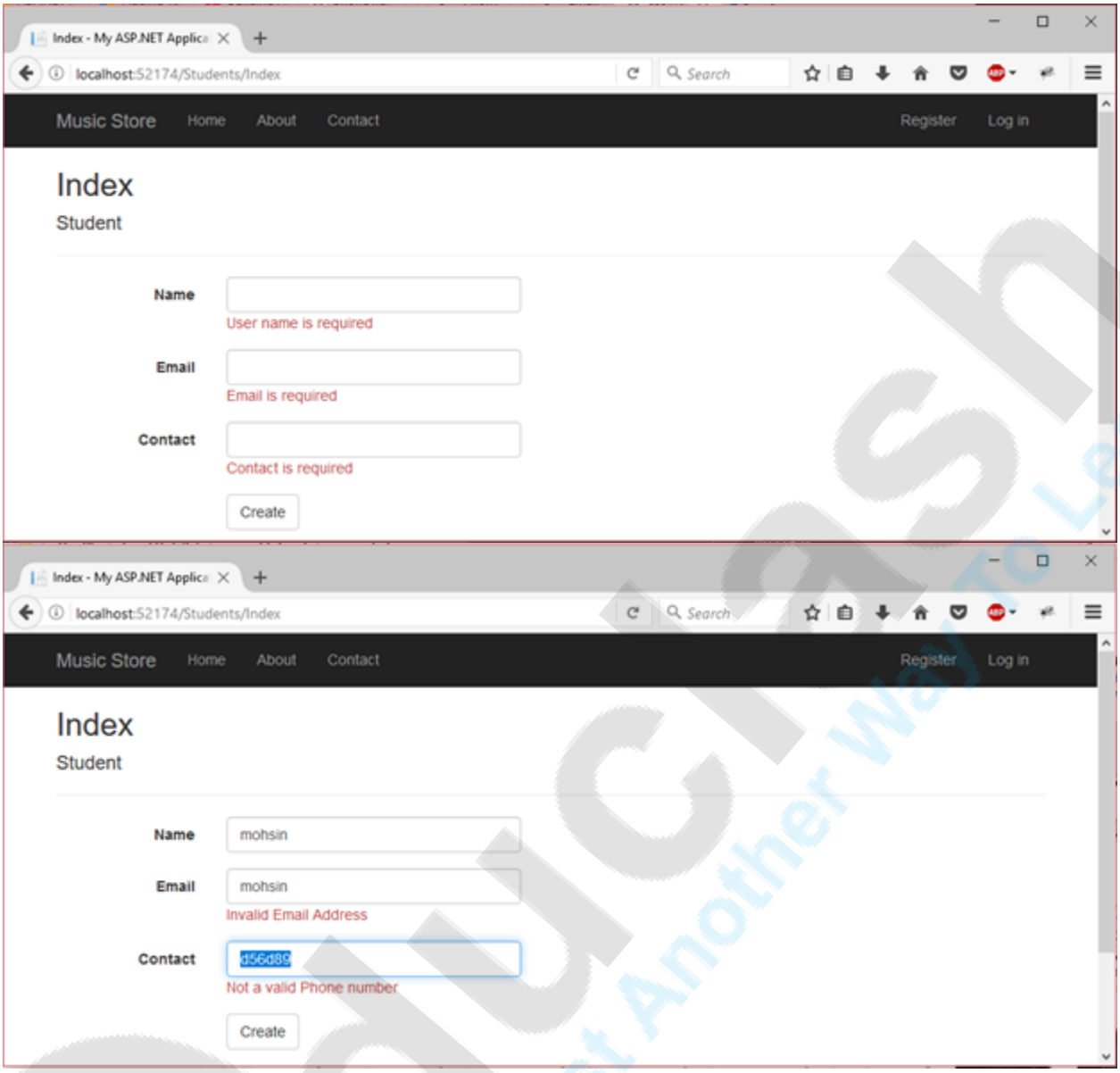
Here, in the following example, we are validating user input by using CompareValidator controller. Source code of the example is given below.

// compare_validator_demo.aspx

```
<%@ Page Language="C#" AutoEventWireup="true" CodeBehind="compare_validator_
demo.aspx.cs"
Inherits="asp.netexample.compare_validator_demo" %>
<!DOCTYPE html>
<html xmlns="http://www.w3.org/1999/xhtml">
<head runat="server">
<title></title>
<style type="text/css">
.auto-style1 {
width: 100%;
    }
.auto-style2 {
height: 26px;
```

```
        }
.auto-style3 {
height: 26px;
width: 93px;
        }
.auto-style4 {
width: 93px;
        }
</style>
</head>
<body>
<form id="form1" runat="server">
<table class="auto-style1">
<tr>
<td class="auto-style3">
                    First value</td>
<td class="auto-style2">
<asp:TextBox ID="firstval" runat="server" required="true"></asp:TextBox>
</td>
</tr>
<tr>
<td class="auto-style4">
    Second value</td>
<td>
<asp:TextBox ID="secondval" runat="server"></asp:TextBox>
    It should be greater than first value</td>
</tr>
<tr>
<td class="auto-style4"></td>
<td>
<asp:Button ID="Button1" runat="server" Text="save"/>
</td>
</tr>
</table>
< asp:CompareValidator ID="CompareValidator1" runat="server" ControlToCompare
="secondval"
ControlToValidate="firstval" Display="Dynamic" ErrorMessage="Enter valid value" For
eColor="Red"
Operator="LessThan" Type="Integer"></asp:CompareValidator>
</form>
</body>
</html>
```

Output:



4)ASP.NET RegularExpressionValidator Control

This validator is used to validate the value of an input control against the pattern defined by a regular expression.

It allows us to check and validate predictable sequences of characters like: e-mail address, telephone number etc.

The **ValidationExpression** property is used to specify the regular expression, this expression is used to validate input control.

RegularExpression Properties

Property	Description
AccessKey	It is used to set keyboard shortcut for the control.

BackColor	It is used to set background color of the control.
BorderColor	It is used to set border color of the control.
Font	It is used to set font for the control text.
ForeColor	It is used to set color of the control text.
Text	It is used to set text to be shown for the control.
ToolTip	It displays the text when mouse is over the control.
Visible	To set visibility of control on the form.
Height	It is used to set height of the control.
Width	It is used to set width of the control.
ErrorMessage	It is used to set error message that display when validation fails.
ControlToValidate	It takes ID of control to validate.
ValidationExpression	It is used to set regular expression to determine validity.

Example

Here, in the following example, we are explaining how to use RegularExpressionValidator control to validate the user input against the given pattern.

// RegularExpressionDemo.aspx

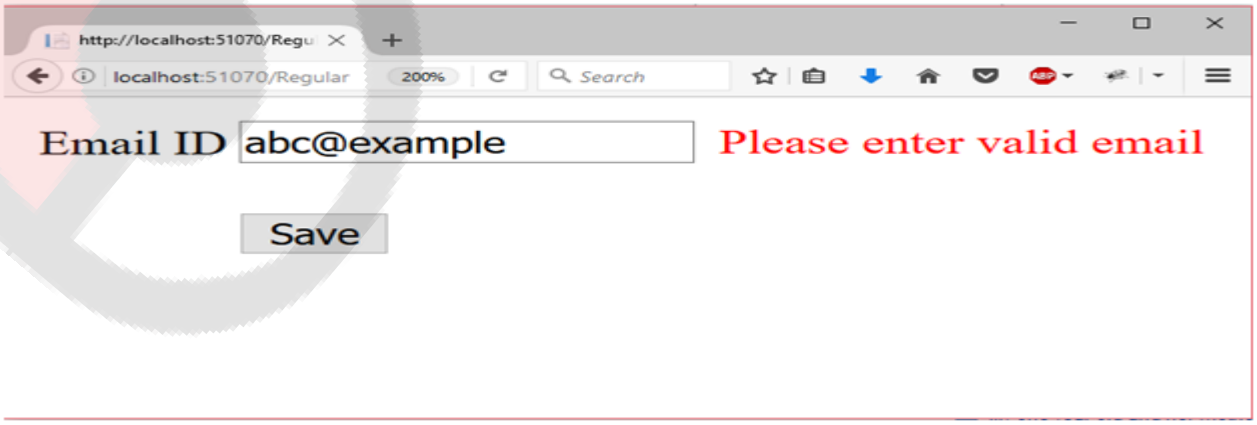
```
<%@ Page Language="C#" AutoEventWireup="true" CodeBehind="RegularExpressionDemo.aspx.cs"
Inherits="asp.netexample.RegularExpressionDemo" %>
<!DOCTYPE html>
<html xmlns="http://www.w3.org/1999/xhtml">
<head runat="server">
<title></title>
</head>
<body>
<form id="form1" runat="server">
<div>
<table class="auto-style1">
<tr>
<td class="auto-style2">Email ID</td>
<td>
<asp:TextBox ID="username" runat="server"></asp:TextBox>
<asp:RegularExpressionValidator ID="RegularExpressionValidator1" runat="server"ControlToValidate="username"
ErrorMessage="Please enter valid email" ForeColor="Red"ValidationExpression="\w+([-+.']\w+)*@\w+([-.]\w+)*\.\w+([-.]\w+)*">
```

```
</asp:RegularExpressionValidator>
</td>
</tr>
<tr>
<td class="auto-style2"></td>
<td>
<br/>
<asp:Button ID="Button1" runat="server" Text="Save"/>
</td>
</tr>
</table>
</div>
</form>
</body>
</html>
```

Output:

It produces the following output when view in the browser.

It will validate email format as we specified in regular expression. If validation fails, it throws an error message.



5) ASP.NET ValidationSummary Control

This validator is used to display list of all validation errors in the web form.

It allows us to summarize the error messages at a single location.

We can set **DisplayMode** property to display error messages as a list, bullet list or single paragraph.

ValidationSummary Properties

This control has following properties.

Property	Description
AccessKey	It is used to set keyboard shortcut for the control.
BackColor	It is used to set background color of the control.
BorderColor	It is used to set border color of the control.
Font	It is used to set font for the control text.
ForeColor	It is used to set color of the control text.
Text	It is used to set text to be shown for the control.
ToolTip	It displays the text when mouse is over the control.
Visible	To set visibility of control on the form.
Height	It is used to set height of the control.
Width	It is used to set width of the control.
ShowMessageBox	It displays a message box on error in up-level browsers.
ShowSummary	It is used to show summary text on the form page.
ShowValidationErrors	It is used to set whether the validation summary should be shown or not.

Example

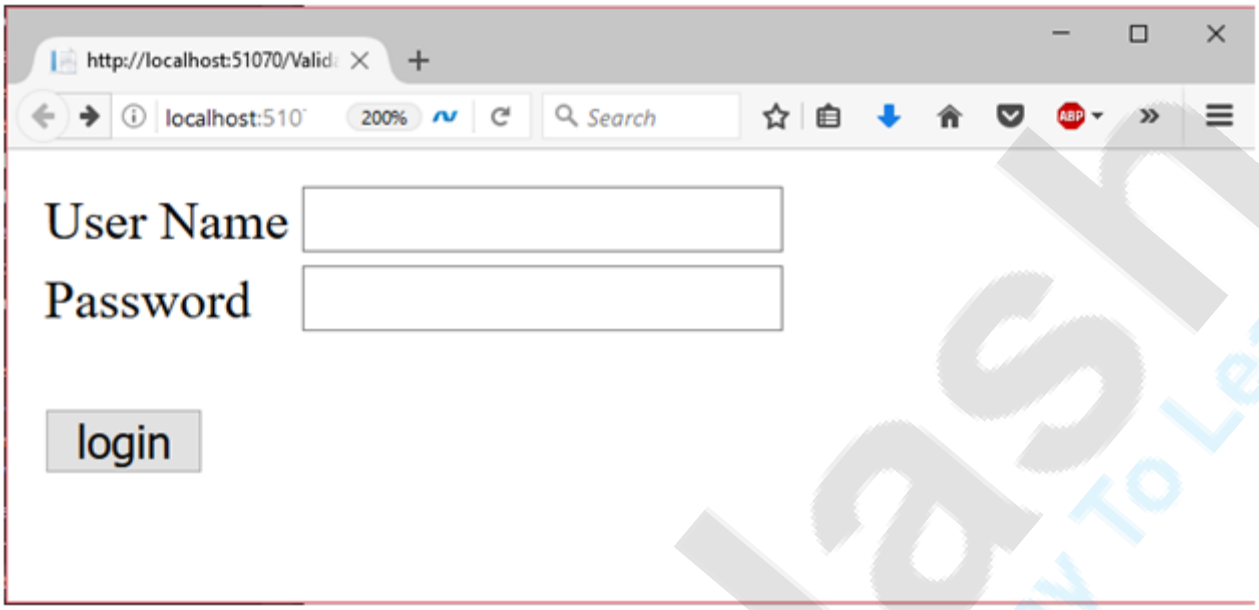
The following example explains how to use **ValidationSummery** control in the application.

// ValidationSummeryDemo.aspx

```
<%@ Page Language="C#" AutoEventWireup="true" CodeBehind="ValidationSummeryDemo.aspx.cs"
Inherits="asp.netexample.ValidationSummeryDemo" %>
<!DOCTYPE html>
<html xmlns="http://www.w3.org/1999/xhtml">
<head runat="server">
<title></title>
</head>
<body>
<form id="form1" runat="server">
<div>
</div>
<table class="auto-style1">
<tr>
<td class="auto-style2">User Name</td>
<td>
<asp:TextBox ID="username" runat="server"></asp:TextBox>
<asp:RequiredFieldValidator ID="user" runat="server" ControlToValidate="username"
ErrorMessage="Please enter a user name" ForeColor="Red">*</asp:RequiredFieldValidator>
</td>
</tr>
<tr>
<td class="auto-style2">Password</td>
<td>
<asp:TextBox ID="password" runat="server"></asp:TextBox>
<asp:RequiredFieldValidator ID="pass" runat="server" ControlToValidate="password"
ErrorMessage="Please enter a password" ForeColor="Red">*</asp:RequiredFieldValidator>
</td>
</tr>
<tr>
<td class="auto-style2">
<br/>
<asp:Button ID="Button1" runat="server"Text="login"/>
</td>
<td>
<asp:ValidationSummary ID="ValidationSummary1" runat="server" ForeColor="Red" />
<br/>
</td>
</tr>
</table>
</form>
</body>
</html>
```

Output:

It produces the following output when view in the browser.



It throws error summary when user login without credentials.

UNIT 5

DATA AND STAE MANAGEMENT WITH ASP.NET

INTRODUCTION: -

Hyper Text Transfer Protocol (HTTP) is a stateless protocol. When the client disconnects from the server, the ASP.NET engine discards the page objects. This way, each web application can scale up to serve numerous requests simultaneously without running out of server memory.

However, there needs to be some technique to store the information between requests and to retrieve it when required. This information i.e., the current value of all the controls and variables for the current user in the current session is called the State.

ASP.NET manages four types of states:

- View State
- Control State
- Session State
- Application State

View State

The view state is the state of the page and all its controls. It is automatically maintained across posts by the ASP.NET framework.

When a page is sent back to the client, the changes in the properties of the page and its controls are determined, and stored in the value of a hidden input field named _VIEWSTATE. When the page is again posted back, the _VIEWSTATE field is sent to the server with the HTTP request.

The view state could be enabled or disabled for:

- **The entire application** by setting the EnableViewState property in the <pages> section of web.config file.
- **A page** by setting the EnableViewState attribute of the Page directive, as <%@ Page Language="C#" EnableViewState="false" %>
- **A control** by setting the Control.EnableViewState property.

It is implemented using a view state object defined by the StateBag class which defines a collection of view state items. The state bag is a data structure containing attribute value pairs, stored as strings associated with objects.

The StateBag class has the following properties:

Properties	Description
Item(name)	The value of the view state item with the specified name. This is the default property of the StateBag class.
Count	The number of items in the view state collection.
Keys	Collection of keys for all the items in the collection.
Values	Collection of values for all the items in the collection.

The StateBag class has the following methods:

Methods	Description
Add(name, value)	Adds an item to the view state collection and existing item is updated.
Clear	Removes all the items from the collection.
Equals(Object)	Determines whether the specified object is equal to the current object.
Finalize	Allows it to free resources and perform other cleanup operations.
GetEnumerator	Returns an enumerator that iterates over all the key/value pairs of the StateItem objects stored in the StateBag object.
GetType	Gets the type of the current instance.
IsItemDirty	Checks a StateItem object stored in the StateBag object to evaluate whether it has been modified.
Remove(name)	Removes the specified item.
SetDirty	Sets the state of the StateBag object as well as the Dirty property of each of the StateItem objects contained by it.
SetItemDirty	Sets the Dirty property for the specified StateItem object in the StateBag object.
ToString	Returns a string representing the state bag object.

Example

The following example demonstrates the concept of storing view state. Let us keep a counter, which is incremented each time the page is posted back by clicking a button on the page. A label control shows the value in the counter.

The markup file code is as follows:

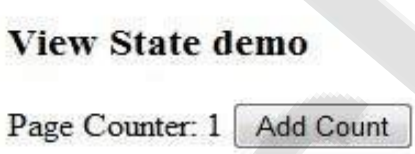
```
<%@ Page Language="C#" AutoEventWireup="true" CodeBehind="Default.aspx.cs"
Inherits="statedemo._Default" %>

<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
"http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">

<html xmlns="http://www.w3.org/1999/xhtml" >

  <head runat="server">
    <title>
      Untitled Page
    </title>
  </head>

  <body>
    <form id="form1" runat="server">

      <div>
        <h3>View State demo</h3>

        Page Counter:

        <asp:Label ID="lblCounter" runat="server" />
        <asp:Button ID="btnIncrement" runat="server" Text="Add Count"
onclick="btnIncrement_Click" />
      </div>

    </form>
  </body>

</html>
```

The code behind file for the example is shown here:

```
public partial class _Default : System.Web.UI.Page
{
  public int counter
  {
    get
    {
      if (ViewState["pcounter"] != null)
      {
        return ((int)ViewState["pcounter"]);
      }
      else
```

```
      {
        return 0;
      }
    }

    set
    {
      ViewState["pcounter"] = value;
    }
  }

  protected void Page_Load(object sender, EventArgs e)
  {
    lblCounter.Text = counter.ToString();
    counter++;
  }
}
```

It would produce the following result:



Control State

Control state cannot be modified, accessed directly, or disabled.

Session State

When a user connects to an ASP.NET website, a new session object is created. When session state is turned on, a new session state object is created for each new request. This session state object becomes part of the context and it is available through the page.

Session state is generally used for storing application data such as inventory, supplier list, customer record, or shopping cart. It can also keep information about the user and his preferences, and keep the track of pending operations.

Sessions are identified and tracked with a 120-bit SessionID, which is passed from client to server and back as cookie or a modified URL. The SessionID is globally unique and random.

The session state object is created from the HttpSessionState class, which defines a collection of session state items.

The HttpSessionState class has the following properties:

Properties	Description
SessionID	The unique session identifier.
Item(name)	The value of the session state item with the specified name. This is the default property of the HttpSessionState class.
Count	The number of items in the session state collection.
TimeOut	Gets and sets the amount of time, in minutes, allowed between requests before the session-state provider terminates the session.

The HttpSessionState class has the following methods:

Methods	Description
Add(name, value)	Adds an item to the session state collection.
Clear	Removes all the items from session state collection.
Remove(name)	Removes the specified item from the session state collection.
RemoveAll	Removes all keys and values from the session-state collection.
RemoveAt	Deletes an item at a specified index from the session-state collection.

The session state object is a name-value pair to store and retrieve some information from the session state object. You could use the following code for the same:

```
void StoreSessionInfo()
{
  String fromuser = TextBox1.Text;
  Session["fromuser"] = fromuser;
}

void RetrieveSessionInfo()
{
  String fromuser = Session["fromuser"];
  Label1.Text = fromuser;
}
```

The above code stores only strings in the Session dictionary object, however, it can store all the primitive data types and arrays composed of primitive data types, as well as the DataSet, DataTable, HashTable, and Image objects, as well as any user-defined class that inherits from the ISerializable object.

Example

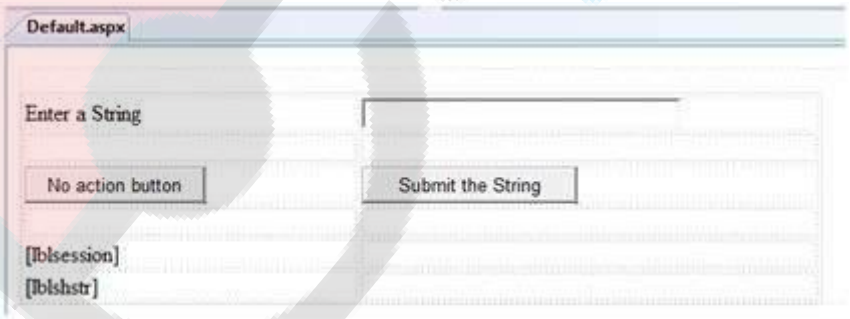
The following example demonstrates the concept of storing session state. There are two buttons on the page, a text box to enter string and a label to display the text stored from last session.

The mark up file code is as follows:

```
<%@ Page Language="C#" AutoEventWireup="true"  CodeFile="Default.aspx.cs" Inherits="_Default"
%>

<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
"http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">

<html xmlns="http://www.w3.org/1999/xhtml" >

  <head runat="server">
    <title>
      Untitled Page
    </title>
  </head>

  <body>
    <form id="form1" runat="server">
      <div>
             

        <table style="width: 568px; height: 103px">

          <tr>
            <td style="width: 209px">
              <asp:Label ID="lblstr" runat="server" Text="Enter a String"  style="width:94px">
              </asp:Label>
            </td>

            <td style="width: 317px">
              <asp:TextBox ID="txtstr" runat="server" style="width:227px">
              </asp:TextBox>
            </td>
          </tr>

          <tr>
            <td style="width: 209px"> </td>
            <td style="width: 317px"> </td>
          </tr>

          <tr>
            <td style="width: 209px">
              <asp:Button ID="btnnrm" runat="server"
                Text="No action button" style="width:128px" />
            </td>

            <td style="width: 317px">
```

```
            <asp:Button ID="btnstr" runat="server"
              OnClick="btnstr_Click" Text="Submit the String" />
          </td>
        </tr>

        <tr>
          <td style="width: 209px"> </td>

          <td style="width: 317px"> </td>
        </tr>

        <tr>
          <td style="width: 209px">
            <asp:Label ID="lblsession" runat="server"  style="width:231px" >
            </asp:Label>
          </td>

          <td style="width: 317px"> </td>
        </tr>

        <tr>
          <td style="width: 209px">
            <asp:Label ID="lblshstr" runat="server">
            </asp:Label>
          </td>

          <td style="width: 317px"> </td>
        </tr>

      </table>

    </div>
  </form>
 </body>
</html>
```

It should look like the following in design view:



The code behind file is given here:

```
public partial class _Default : System.Web.UI.Page
{
  String mystr;
```

```
  protected void Page_Load(object sender, EventArgs e)
  {
    this.lblshstr.Text = this.mystr;
    this.lblsession.Text = (String)this.Session["str"];
  }

  protected void btnstr_Click(object sender, EventArgs e)
  {
    this.mystr = this.txtstr.Text;
    this.Session["str"] = this.txtstr.Text;
    this.lblshstr.Text = this.mystr;
    this.lblsession.Text = (String)this.Session["str"];
  }
}
```

Execute the file and observe how it works:

Application State

The ASP.NET application is the collection of all web pages, code and other files within a single virtual directory on a web server. When information is stored in application state, it is available to all the users.

To provide for the use of application state, ASP.NET creates an application state object for each application from the HTTPApplicationState class and stores this object in server memory. This object is represented by class file global.asax.

Application State is mostly used to store hit counters and other statistical data, global application data like tax rate, discount rate etc. and to keep the track of users visiting the site.

The HttpApplicationState class has the following properties:

Properties	Description
Item(name)	The value of the application state item with the specified name. This is the default property of the HttpApplicationState class.
Count	The number of items in the application state collection.

The HttpApplicationState class has the following methods:

Methods	Description
Add(name, value)	Adds an item to the application state collection.
Clear	Removes all the items from the application state collection.
Remove(name)	Removes the specified item from the application state collection.
RemoveAll	Removes all objects from an HttpApplicationState collection.
RemoveAt	Removes an HttpApplicationState object from a collection by index.
Lock()	Locks the application state collection so only the current user can access it.
Unlock()	Unlocks the application state collection so all the users can access it.

Application state data is generally maintained by writing handlers for the events:

- Application_Start
- Application_End
- Application_Error
- Session_Start
- Session_End

The following code snippet shows the basic syntax for storing application state information:

```
Void Application_Start(object sender, EventArgs e)
{
  Application["startMessage"] = "The application has started.";
}

Void Application_End(object sender, EventArgs e)
{
  Application["endtMessage"] = "The application has ended.";
}
```

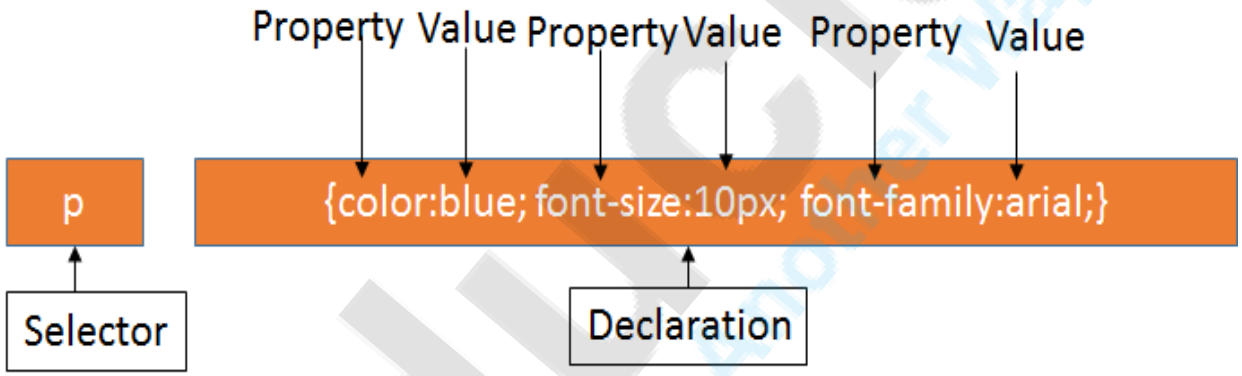
5.1) ASP.NET WEBSITES WITH THEMES AND MASTER PAGES

5.1.1) THEMES IN ASP.NET

CSS Introduction

CSS (Cascading Style Sheets) is used for defining styles to display the elements written in a markup language. It helps user separate the HTML or XHTML content from it style. The separation provides flexibility, faster accessibility to content, reduces complexity and repetition of data.

The syntax for CSS has two parts, a selector and one or more declarations. The selector contains HTML element for which the style is to added. The declaration contains property and a value associated with it. The pictorial representation of the syntax is shown below:



CSS comments: Comments are used for explaining codes. Comments are ignored by the browser for execution. The comments begin with '/*' and ends with '*/'. A sample code after adding comments is shown below:

```
?
1    h1
2    {
3        /* Set the font with the specified style*/
4        text-font : arial;
5        /*Set the background color*/
6        color:red;
7    }
```

There are two types of style sheets declaration used for designing. The list of details are explained below:

- **Internal Style Sheet**
- **External Style Sheet**
- **Inline Style Sheet**

Internal Style Sheet: The style is defined in the <head> section of the web page using the <style> tag. The styles defined are limited to the particular web page in which it is declared. User must create a new style for every new page added in the document. Internal Styles are also known as 'Embedded' styles.

A code sample of internal style sheet is mentioned below:

```
<html xmlns="http://www.w3.org/1999/xhtml">
 <head runat="server">
    <title>First Web Page</title>
     <style type="text/css" >
    h1 { color:Red}
    p{ background:gray}
     </style>
 </head>
 <body>
    <form id="form1" runat="server">
      <div>
        <h1>ASP Tutorial</h1>
    <p>The tutorial is a quick review for the topics required for web
development.</p>
      </div>
    </form>
 </body>
</html>
```

The output after executing the code is shown below:

ASP Tutorial

The tutorial is a quick review for the topics required for web development. Study is easier using these tutorials.

External Style Sheet: The styles created in an external style sheets can be reused by many applications. There is an external style page created and it can be linked to the web page. The external style sheet is a text file created with **.css** extension. The syntax to link the external style sheet to the web page is as shown below:

```
<head>
 <title>Web Creation</title>
 <link href="name1.css" rel="stylesheet" type="text/css" >
</head>
```

The code added in the CSS file is as shown below:

```
body
{
   Background-color:Aqua;
}
```

```
h1
{
   border :10pt, 5pt, 4pt, 4pt;
   background-color:Yellow;
}
p
{
    background-color:Green;
    border-style:dotted;
}
```

The code to link the CSS file with the web page is as shown below:

```
<html xmlns="http://www.w3.org/1999/xhtml">
 <head runat="server">
   <title>First Web Page</title>
   <link rel="Stylesheet" type="text/css" href="StyleSheet1.css" />
 </head>
 <body>
    <form id="form1" runat="server" >
     <div>
      <h1>ASP Tutorial</h1>
       <p> The tutorial is a quick review for the topics required for web development.
            Study is easier using these tutorials</p>
     </div>
 </body>
</html>
```

The output of the file when executed on the server is as shown below:

Inline Style Sheet: The Inline style sheets are used for adding style to the particular element in the web page. The inline style element is embedded directly in the html elements. The selector element is not required in Inline styles. An example of adding an Inline style in an html element is as mentioned below:

It is not efficient to use the Inline style for large codes as the style is limited for an individual element.

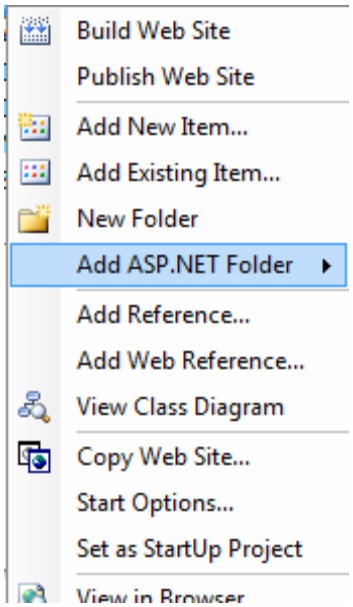
** **

Use Themes to customize the site

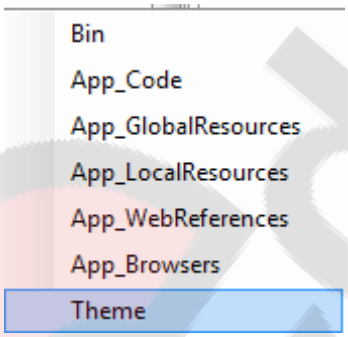
Themes are used to apply consistent look to all the pages in a website. When the theme is set at the website level, all the controls and pages override theme. The steps to add a Theme in ASP.NET application are as follows:

1) Create a web application in Visual Studio application.

2) Open the Solution Explorer window, right click the project name and select ASP.NET folder.



3) Select 'Theme' from the menu item list



4) The Add_Themes folder is created and rename the folder. The syntax to add theme to a web site is as mentioned below:

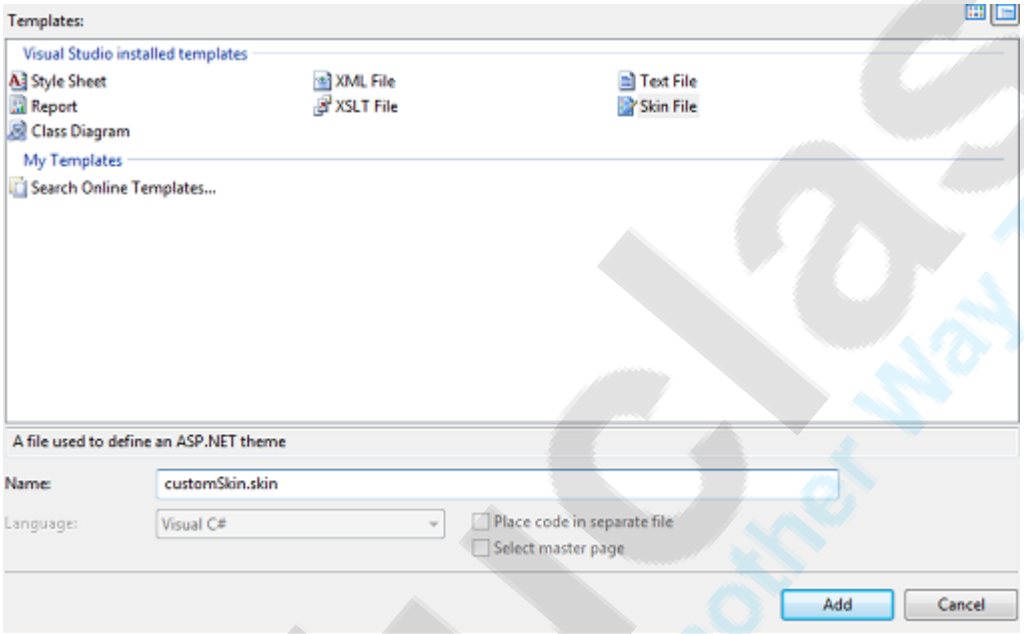
```
<configuration>
   <system.web>
     <pages theme="ThemeName" />
   </system.web>
</configuration>
```

Name skins within a Theme

The skin files are used for adding styles to the control in ASP.NET. The skin files are added in the Theme folder in the solution explorer. Every ASP.NET control has a unique **SkinID** attribute associated with it. The steps to add the skin file in application are as mentioned below:

1. In the Solution Explorer, right click to the theme folder and click 'Add New Item' option
2. In the Add New Item dialog, click 'Skin File' option
3. In the Name box, add a name for the .skin file and click 'Add'

The screen shot to add a skin file in an ASP.NET application is as shown below:



The code snippet for adding style through skin file is shown below:

```
<%--
Default skin template. The following skins are provided as examples only.

1.Named control skin. The SkinId should be uniquely defined because
duplicate SkinId's per control type are not allowed in the same theme.
<asp:GridVew runat="server" SkinID="gridviewSkin" BackColor="White" >
    <AlternatingRowStyle BackColor="Blue" />
</asp:GridView>
2: Default skin. The SkinId is not defined. Only one default control skin
per control type is allowed in the same theme.

--%>
<asp:Image runat="server" ForeColor="Red" SkinId="lblRed" />
```

The code for adding skinID attribute in the source code is as shown below:

```
<%@ Page Theme="mytheme" Language="C#" AutoEventWireup="true"
CodeFile="Default2.aspx.cs" %>
```

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN "
"http://www.w3.org/TR/xmtml1" >

<html xmlns="http://www.w3.org/1999/xhtml" >
 <head runat="server">
    <title></title>
 </head>
 <body>
   <form id="form1" runat="server">
    <div>
     <p>Demonstration of Themes</p>
     <asp:Label ID="lblmessage" runat="server" Text="New
Data"></asp:Label><br/>
     <asp:Label ID="Label1" runat="server" Text="New Data"
SkinID="lblRed"></asp:Label>
    </div>
   </form>
 </body>
</html><img cropbottom="11171f"
src="file:///C:\Users\sony\AppData\Local\Temp\msohtmlclip1\01\clip_image015.
png" class="cboxElement">
```

The output of the code sample is shown below:

Contents of Theme and Skin

Theme can be used as a style sheet in a web page. User can add theme as style sheet theme by adding a **styleSheetTheme** attribute in the Web.config file on the application. The syntax of adding theme as style sheet is as shown below:

```
?
   <configuration>
    <system.web>
      <pages styleSheetName="ThemeName" />
    </system.web>
   </configuration>
```

Themes can be applied to an individual page in the application. User can set the Theme or **StyleSheetTheme**attribute of the @Page directive to the name of the theme.

<%@ Page Theme=”ThemeName” %>
<%@ Page StyleSheetTheme=”ThemeName” %>

Themes can be disabled for a page in ASP.NET. The **EnableTheming** attribute of the @Page directive to false. The syntax to enable theme is as shown below:

<%@Page EnableTheming=”false” %>

Themes can be disabled for a specific control by setting the value of the control. The syntax to disable the Theme is as shown below:

<asp:Button id=”Button1” runat=”server” EnableTheming=”false” />

User can create global theme that can be applied to all websites on a server. The steps to create a global theme are as follows:

1) Create the Themes folder in the following path:
%windows%\Microsoft.NET\Framework\version\ASP.NETClienyFiles\Themes

2) Add the files to the folder containing style sheets and images that can be applied globally

3) Run the command **‘aspnet_regiis –c’** to install the theme on the server running IIS

4) If the user is testing the theme on FTP web site, the theme folder can be manually created in the following path:

IISRootWeb\aspnet_client\system_web\version\Themes

Named Skins: Skins without SkinID are called default skin. The skins with an ID are known as named skin. Server control with unique ID can have different layouts defined for them. They can be defined in same file or different files depending on the requirement. SkinID can be used as reference to call the named skin. An example of named skin is as shown below:

<asp:Label runat=”server” Fore-color=”red” SkinID=”LabelHeader” />

<asp:Label runat=”server” Fore-color=”Blue” SkinID=”LabelFooter” />

An example to demonstrate referencing of named skin through ID is shown below:

<asp:Label id=”Header” runat=”server” SkinID=”LabelHeader” />
<asp:Label id=”Header” runat=”server” SkinID=”LabelFooter” />

The skin files can be automatically registered in the application. User must set the attribute named **EnableEmbeddedSkins** to true and the **Skin** property is set to built-in skin. The control will be automatically registered to the CSS file.

Apply Themes and Profiles

- ASP.NET profiles are used to store information with an individual user in persistent format. The profile can be defined in **web.config** or **machine.config** file for the application. The data is not lost even when the session is expired or closed. Profiles are strongly typed and provide an intelliSense for development process.
- The <profile> section is the base object for developing profiles. The <profile> section is part of the <system.web> file. The <profile> section includes the

provider information. The <properties> section include the provider information. The default value for property is System.string.

The list of the attributes used for profile generation are as follows:

Attribute	Description
allowAnonymous	It is used for storing values for anonymous users. By default the value is false
customProviderData	It contains initialization of data for custom profile provider
defaultValue	It is used for indicating default value for the property
name	It is used to indicate the name of the property. It is a mandatory value
provider	It is used to indicate the name of the provider for reading and writing values for the profile properties
readOnly	It is used for the property value
serialzeAs	It indicates to serialize the value for the property
type	It indicates the type of the property

The code sample below indicates the profile used in the application.

```
<profile>
 <properties>
    <add name=”Grader” allowAnonymous=”true” />
    <add name=”Section” allowAnonymous=”true” />
 </properties>
</profile>
```

Profile Groups: User can create profile by grouping properties. It is easier to simulate multiple profiles for an application. The code sample for creating profile groups is mentioned below:

```
 <profile>
  <properties>
    <group name=”Books”>
     <add name=”Fiction” />
     <add name=”Creativity” />
    </group>
    <group name=”Publications>
     <add name=”Branch” />
     <add name=”city” />
    </group>
  </properties>
 </profile><img src="file:///C:\Users\sony\AppData\Local\Temp\msohtmlclip1\01\cli_ima
```

The code sample for creating profile in ASP.NET is as shown below:

```
<html xmlns="http://www.w3.org/1999/xhtml" >
 <head runat="server" >
    <title></title>
 </head>
 <body>
    <form id="form1" runat="server">
     <div>
     First Name:<asp:TextBox ID="TextBox1"
runat="server"></asp:TextBox><br/>
     Last Name:<asp:TextBox ID="TextBox2"
runat="server"></asp:TextBox><br/>
     <br/>
     </div>
    </form>
 </body>
</html>
```

The code in the web.config file for the application is as shown below:

```
<?xml version="1.0"?>
<!-
   For more information on how to configure your ASP.NET application,
please visit
   http://go.microsoft.com/fwlink/?LinkId=169433
-->
<configuration>
  <system.web>
    <compilation debug="true" targetFrameowork="4.0" />
    <authentication mode="Forms" />
    <profile>
       <properties>
      <add name="FirstName" defualtValue="Sam" />
      <add name="LastName" defualtValue="Henderson" />
       </properties>
    </profile>
  </system.web>
</configuration>
```

The action to be performed is written in the code behind file of the application is shown below:

```
    public partial class _Default : System.Web.UI.Page
    {
      protected void Page_Load( object sender, EventArgs e)
      {
        if(!IsPostBack)
        {
          this.TextBox1.Text = Profile.FirstName;
          this.TextBox2.Text = Profile.LastName;
        }
      }
    }
```

The output of the code when executed on the server is shown below:

5.1.2) ASP.NET MASTERPAGES

- Master pages allow you to create a consistent look and behavior for all the pages (or group of pages) in your web application.
- A master page provides a template for other pages, with shared layout and functionality. The master page defines placeholders for the content, which can be overridden by content pages. The output result is a combination of the master page and the content page.
- The content pages contain the content you want to display.
- When users request the content page, ASP.NET merges the pages to produce output that combines the layout of the master page with the content of the content page.

Master Page Example

```
<%@ Master %>

<html>
<body>
<h1>Standard Header From Masterpage</h1>
<asp:ContentPlaceHolder id="CPH1" runat="server">
</asp:ContentPlaceHolder>
</body>
</html>
```

The master page above is a normal HTML page designed as a template for other pages.

The **@ Master** directive defines it as a master page.

The master page contains a placeholder tag **<asp:ContentPlaceHolder>** for individual content.

The **id="CPH1"** attribute identifies the placeholder, allowing many placeholders in the same master page.

This master page was saved with the name **"master1.master".**

5.2) DATASOURCE CONTROLS

A data source control interacts with the data-bound controls and hides the complex data binding processes. These are the tools that provide data to the data bound controls and support execution of operations like insertions, deletions, sorting, and updates.

Each data source control wraps a particular data provider-relational databases, XML documents, or custom classes and helps in:

- Managing connection
- Selecting data
- Managing presentation aspects like paging, caching, etc.
- Manipulating data

There are many data source controls available in ASP.NET for accessing data from SQL Server, from ODBC or OLE DB servers, from XML files, and from business objects.

Based on type of data, these controls could be divided into two categories:

- Hierarchical data source controls
- Table-based data source controls

The data source controls used for hierarchical data are:

- **XMLDataSource** - It allows binding to XML files and strings with or without schema information.
- **SiteMapDataSource** - It allows binding to a provider that supplies site map information.

The data source controls used for tabular data are:

Data source controls	Description
SqlDataSource	It represents a connection to an ADO.NET data provider that returns SQL data, including data sources accessible via OLEDB and ODBC.
ObjectDataSource	It allows binding to a custom .Net business object that returns data.
LinqdataSource	It allows binding to the results of a Linq-to-SQL query (supported by ASP.NET 3.5 only).

AccessDataSource	It represents connection to a Microsoft Access database.

Data Source Views

Data source views are objects of the DataSourceView class. Which represent a customized view of data for different data operations such as sorting, filtering, etc.

The DataSourceView class serves as the base class for all data source view classes, which define the capabilities of data source controls.

The following table provides the properties of the DataSourceView class:

Properties	Description
CanDelete	Indicates whether deletion is allowed on the underlying data source.
CanInsert	Indicates whether insertion is allowed on the underlying data source.
CanPage	Indicates whether paging is allowed on the underlying data source.
CanRetrieveTotalRowCount	Indicates whether total row count information is available.
CanSort	Indicates whether the data could be sorted.
CanUpdate	Indicates whether updates are allowed on the underlying data source.
Events	Gets a list of event-handler delegates for the data source view.
Name	Name of the view.

The following table provides the methods of the DataSourceView class:

Methods	Description
CanExecute	Determines whether the specified command can be executed.
ExecuteCommand	Executes the specific command.
ExecuteDelete	Performs a delete operation on the list of data that the DataSourceView object represents.
ExecuteInsert	Performs an insert operation on the list of data that the DataSourceView object represents.

ExecuteSelect	Gets a list of data from the underlying data storage.
ExecuteUpdate	Performs an update operation on the list of data that the DataSourceView object represents.
Delete	Performs a delete operation on the data associated with the view.
Insert	Performs an insert operation on the data associated with the view.
Select	Returns the queried data.
Update	Performs an update operation on the data associated with the view.
OnDataSourceViewChanged	Raises the DataSourceViewChanged event.
RaiseUnsupportedCapabilitiesError	Called by the RaiseUnsupportedCapabilitiesError method to compare the capabilities requested for an ExecuteSelect operation against those that the view supports.

The SqlDataSource Control

The SqlDataSource control represents a connection to a relational database such as SQL Server or Oracle database, or data accessible through OLEDB or Open Database Connectivity (ODBC). Connection to data is made through two important properties ConnectionString and ProviderName.

The following code snippet provides the basic syntax of the control:

```
<asp:SqlDataSource runat="server" ID="MySqlSource"
  ProviderName='<%$ ConnectionStrings:LocalNWind.ProviderName  %>'
  ConnectionString='<%$ ConnectionStrings:LocalNWind %>'
  SelectionCommand= "SELECT * FROM EMPLOYEES" />

<asp:GridView ID="GridView1" runat="server" DataSourceID="MySqlSource" />
```

Configuring various data operations on the underlying data depends upon the various properties (property groups) of the data source control.

The following table provides the related sets of properties of the SqlDataSource control, which provides the programming interface of the control:

Property Group	Description

DeleteCommand, DeleteParameters, DeleteCommandType	Gets or sets the SQL statement, parameters, and type for deleting rows in the underlying data.
FilterExpression, FilterParameters	Gets or sets the data filtering string and parameters.
InsertCommand, InsertParameters, InsertCommandType	Gets or sets the SQL statement, parameters, and type for inserting rows in the underlying database.
SelectCommand, SelectParameters, SelectCommandType	Gets or sets the SQL statement, parameters, and type for retrieving rows from the underlying database.
SortParameterName	Gets or sets the name of an input parameter that the command's stored procedure will use to sort data.
UpdateCommand, UpdateParameters, UpdateCommandType	Gets or sets the SQL statement, parameters, and type for updating rows in the underlying data store.

The following code snippet shows a data source control enabled for data manipulation:

```
<asp:SqlDataSource runat="server" ID= "MySqlSource"
  ProviderName='<%$ ConnectionStrings:LocalNWind.ProviderName  %>'
  ConnectionString=' <%$ ConnectionStrings:LocalNWind %>'
  SelectCommand= "SELECT * FROM EMPLOYEES"
  UpdateCommand= "UPDATE EMPLOYEES SET LASTNAME=@lame"
  DeleteCommand= "DELETE FROM EMPLOYEES WHERE EMPLOYEEID=@eid"
  FilterExpression= "EMPLOYEEID > 10">
  .....
  .....
</asp:SqlDataSource>
```

The ObjectDataSource Control

The ObjectDataSource Control enables user-defined classes to associate the output of their methods to data bound controls. The programming interface of this class is almost same as the SqlDataSource control.

Following are two important aspects of binding business objects:

- The bindable class should have a default constructor, it should be stateless, and have methods that can be mapped to select, update, insert, and delete semantics.
- The object must update one item at a time, batch operations are not supported.

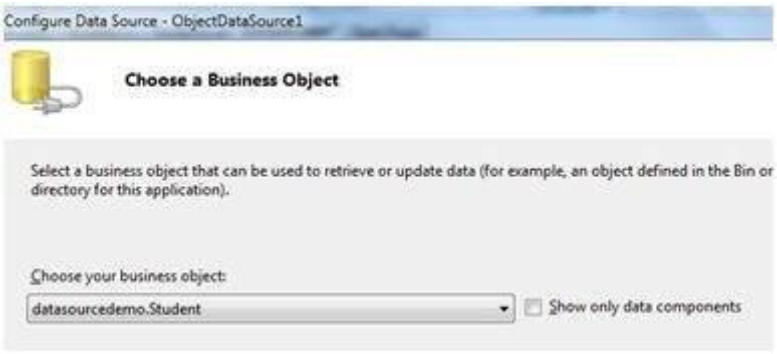
Let us go directly to an example to work with this control. The student class is the class to be used with an object data source. This class has three properties: a student id, name, and city. It has a default constructor and a GetStudents method for retrieving data.

The student class:

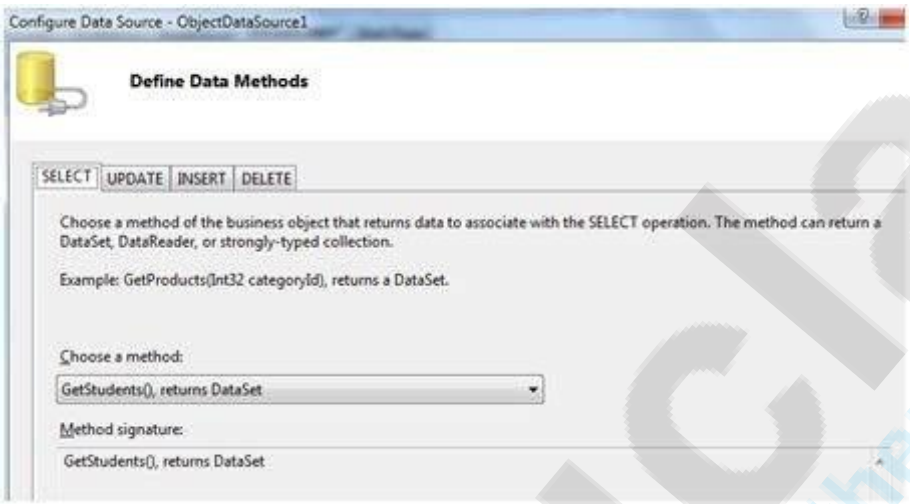
```
public class Student
{
  public int StudentID { get; set; }
  public string Name { get; set; }
  public string City { get; set; }

  public Student()
  { }

  public DataSet GetStudents()
  {
    DataSet ds = new DataSet();
    DataTable dt = new DataTable("Students");

    dt.Columns.Add("StudentID", typeof(System.Int32));
    dt.Columns.Add("StudentName", typeof(System.String));
    dt.Columns.Add("StudentCity", typeof(System.String));
    dt.Rows.Add(new object[] { 1, "M. H. Kabir", "Calcutta" });
    dt.Rows.Add(new object[] { 2, "Ayan J. Sarkar", "Calcutta" });
    ds.Tables.Add(dt);

    return ds;
  }
}
```

Take the following steps to bind the object with an object data source and retrieve data:

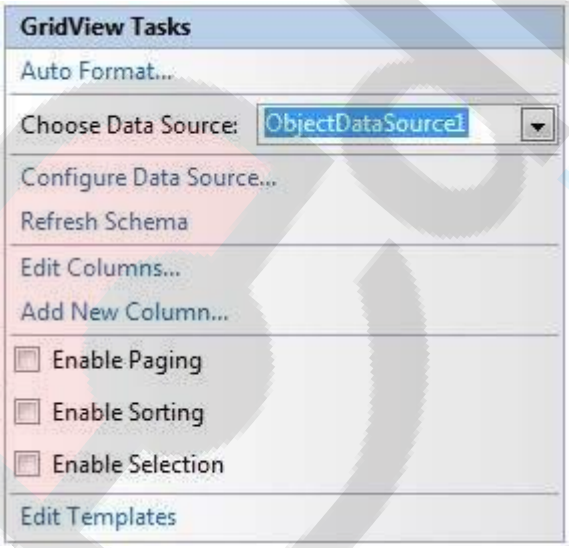
- Create a new web site.
- Add a class (Students.cs) to it by right clicking the project from the Solution Explorer, adding a class template, and placing the above code in it.
- Build the solution so that the application can use the reference to the class.
- Place an object data source control in the web form.
- Configure the data source by selecting the object.



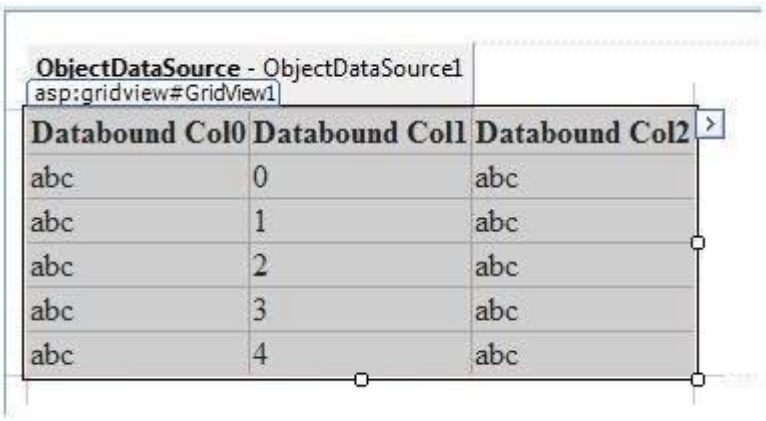
- Select a data method(s) for different operations on data. In this example, there is only one method.



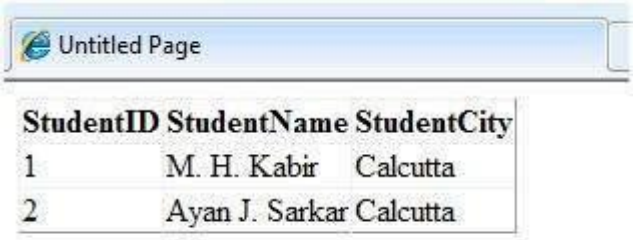
- Place a data bound control such as grid view on the page and select the object data source as its underlying data source.



- At this stage, the design view should look like the following:



- Run the project, it retrieves the hard coded tuples from the students class.



The AccessDataSource Control

The AccessDataSource control represents a connection to an Access database. It is based on the SqlDataSource control and provides simpler programming interface. The following code snippet provides the basic syntax for the data source:

```
<asp:AccessDataSource ID="AccessDataSource1 runat="server"

   DataFile="~/App_Data/ASPDotNetStepByStep.mdb" SelectCommand="SELECT * FROM
[DotNetReferences]">

</asp:AccessDataSource>
```

The AccessDataSource control opens the database in read-only mode. However, it can also be used for performing insert, update, or delete operations. This is done using the ADO.NET commands and parameter collection.

Updates are problematic for Access databases from within an ASP.NET application because an Access database is a plain file and the default account of the ASP.NET application might not have the permission to write to the database file.

5.3) DATA BOUND CONTROLS

- Data-bound web server controls are controls that can be bound to a data source control to make it easy to display and modify data in your web application. All of these controls provide a variety of properties that you can set to control the appearance of the UI that they generate. For scenarios where you need greater control over a control's UI or how it processes input, some of these controls let you specify the generated HTML directly using *templates*. A template is a block of HTML markup that includes special variables that you use to specify where and how the bound data is to be displayed. When the control is rendered, the variables are replaced with actual data and the HTML is rendered to the browser.

- Data-bound web server controls are composite controls that combine other ASP.NET web controls, such as Label and TextBox controls, into a single layout.

VARIOUS CONTROLS IN DATABOUND

1) SqlDataSource control

The SqlDataSource control is used to access data located in the relational database. It uses the ADO.NET classes to interact with any databases that are supported by ADO.NET. The providers that can be used are oracle, SQL server, ODBC and OleDb. Using the control user can access and manipulate data in the ASP.NET page.

To use the SqlDataSource control, set the **ProviderName** and **ConnectionString** property. The **ProviderName** is used for the type of the database used and **ConnectionString** for the connection to the database.

The data of the SqlDataSource control can be cached and retrieved in the application. The caching can be enabled by setting the **EnableCaching** property to true. The filter of the SqlDataSource control supports a **FilterExpression** property. User can add the selection criteria for filtering of data.

The SqlDataSource control code sample is as shown below:

```
1   <asp:SqlDataSource id=”sldatasource1” runat=”server” DataSourceMode=”DataReader” Connection
2    SelectCommand=”Select name from Employee” >
3   </asp:SqlDataSource>
```

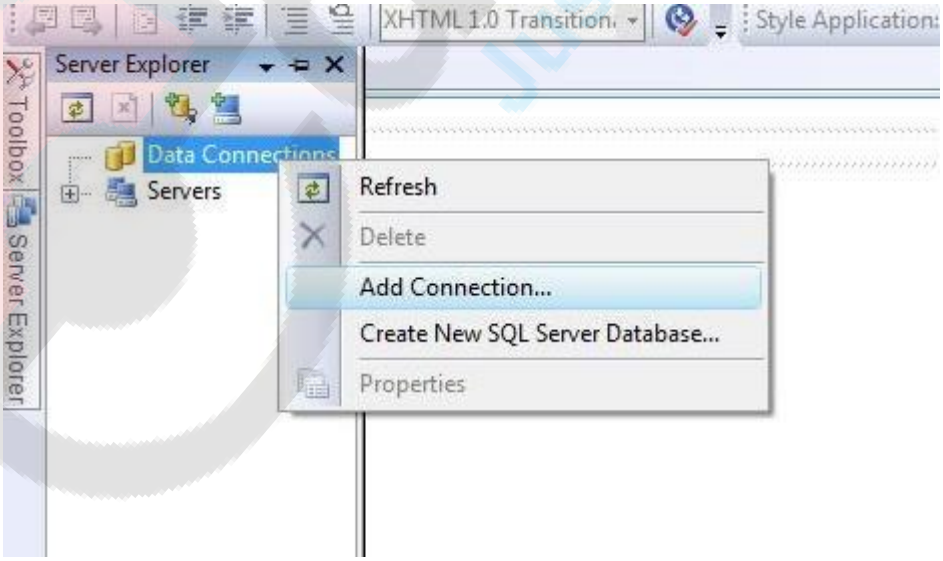
2) GridView Control

The GridView control is used to provide flexibility in working and display data from the database. The data in the GridView control is displayed in the tabular format. It has several properties assigned to the control. Some of the properties are as mentioned below:

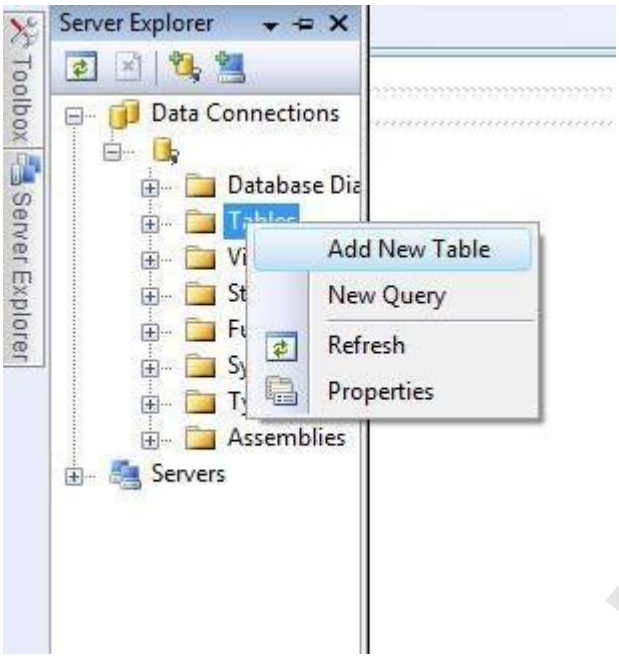
Property	Description
AllowPaging	It is a Boolean value indicating the control supports paging
AllowSorting	It is a Boolean value indicating the control supports sorting
SortExpression	It accepts the current expression determining the order of the row
Datasource	It is used to get or set the data source object containing the data to populate the control
AutoGenerateEditButton	It is a Boolean value indicating that the user can edit the record in the control
DataSourceID	It indicates the data source control to be used
AutoGenerateDeleteButton	It is a Boolean value indicating that the user can delete the record in the control
AutoGenerateColumns	It is a Boolean value to indicate the columns are automatically created for each field of the data source
AutoGenerateSelectButton	It is a Boolean value to indicate the column should be added to select the particular record
SortDirection	It gets the sorting direction of the column for the control
EmptyDataText	It indicates the text to appear when there is no record in the data source

The code sample of the GridView control is as shown below:

1) Add a new connection object to the ASP.NET web application as shown below:

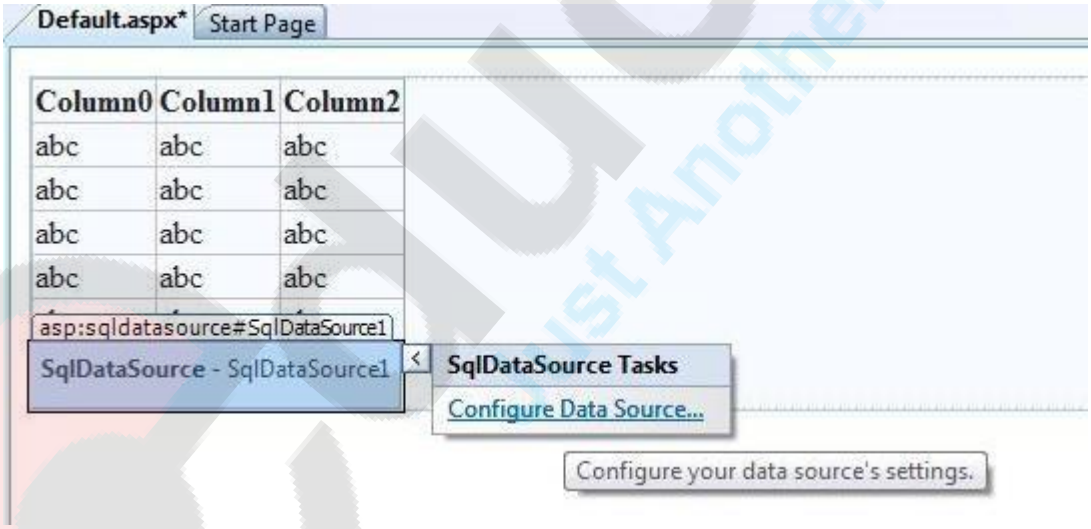


2) Next, add a new table for the connection object created above. The snippet for adding the table is as shown below:



3) Add the fields Sno, Name, Address in the table. Add values to the respective fields in the table

4) Add the GridView and SqlDataSource control to the design view of the web page.



5) The source code for the GridView control is as shown below:

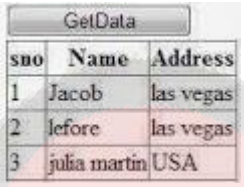
```
?
1 <%@Page Language="C#" AutoEventWireup="true" CodeFile="binding.aspx.cs"
Inherits="binding" %>
2
3 <!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
4 "http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
5 <html xmlns="http://www.w3.org/1999/xhtml">
6  <head runat="server">
7      <title></title>
  </head>
8  <body>
```

```
    <form id="form1" runat="server" >
      <asp:Button ID="Button1" runat="server" onclick="Button1_Click"
Text="GetData" Width="123px" />
      <br/>
      <div>
     <asp:GridView ID="GridView1" runat="server">
     </asp:GridView>
      </div>
   </form>
 </body>
</html>
```

6) The code behind file contains the following code

```
protected void Button1_Click(object sender, EventArgs e)
{
  SqlConnection con = new SqlConnection();
  con.ConnectionString = ConfigurationManager.ConnectionStrings
  [ "ConnectionString" ].ToString();
    con.Open();

  SqlCommand cmd = new SqlCommand();
  cmd.CommandText = "Select * from deltable";
  cmd.Connection = con;
  DataSet ds = new DataSet();
  da.Fill( ds, "deltable");
  GridView1.DataSource= ds;
  GridView1.DataBind();
}
```

7) **The output is:**



3) DetailsView control

Details view control is used as a data bound control. It is used to render one record at a time. User can insert, update and delete the record from the control. Some of the properties of the DetailsView control is as shown below:

Property	Description
AllowPaging	It is a Boolean value to indicate the control supports navigation
DataSource	It is used to populate the control with the data source object
DataSourceID	It indicates the bound data source control with corresponding adapters

AutoGenerateEditButton	It is a Boolean value to indicate the column can be edited by the user
AutoGenerateDeleteButton	It is a Boolean value to indicate the records can be deleted
DefaultMode	It is used to indicate the default display mode of the control
AutoGenerateRows	It is a Boolean value to indicate the rows are automatically created for each field in the data source

The sample code for the Details View control is as shown below:

```
<%@ Page Language="C#" %>

<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional //EN"
"http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd" >

<script runat="server">
</script>

<html xmlns="http://www.w3.org/1999/xhtml">
 <head runat="server">
    <title>asp.net DetailsView example: how to use DetailsView</title>
 </head>
 <body>
    <form id="form1" runat="server" >
    <div>
    <h2 style="color:Navy" >DetailsView Example</h2>
    <asp:SqlDataSource ID="SqlDataSource1" runat="server"
ConnectionString="<%$ ConnectionStrings: NorthwindConnectionString %>"
    Select Command = "SELECT ProductID, ProductName, UnitPrice FROM
Products";
    </asp:SqlDataSource>
    <asp:DetailsView ID="DetailsView" runat="server"
DataSourceID="SqlDataSource1" AllowPaging="true" ForeColor="DarkGreen"
BackColor="Snow" BorderColor="Tomato">
    </asp:DetailsView>
    </div>
    </form>
 </body>
</html>
```

The output is:

DetailsView Example

ProductID	1
ProductName	Chai
UnitPrice	18.0000
1 2 3 4 5 6 7 8 9 10 ...	

4) FormView Control

The FormView control is data bound control but it uses templates version of the DetailsView control. The templates are used inside the control for rendering it on the server. Some of the properties of the FormView control are as shown below:

Property	Description
EditItemTemplate	It is used when the record is being edited by the user
InsertItemTemplate	It is used when a record is being created by the user
ItemTemplate	It is the template used to render the record to display only in an application

Methods of FormView control

Methods	Description
InsertItem	It is used to insert record in the database
UpdateItem	It is used to update record in the database
DeleteItem	It is used to delete the record in the database
ChangeMode	It is used to change the working state of the control

The sample code for the FormView control is as shown below:

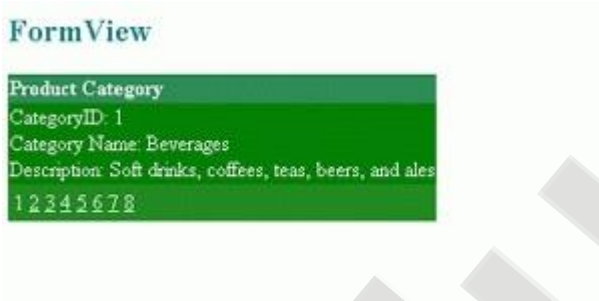
```
?
1  <%@ Page Languag"C#" AutoEventWireup="true" CodeFile="FormView.aspx.cs"
2  Inherits="FormView" %>
3
4  <!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
5  "http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd" >
6
7  <html xmlns="http://www.w3.org/1999/xhtml">
8   <head runat="server" >
9      <title>asp.net FormView: how to use</title>
10   </head>
11   <body>
12      <form id="form1" runat="server">
13      <div>
14         <h2 style="color:Teal">FormView</h2>
15         <asp:SqlDataSource ID="SqlDataSource1" runat="server"
16 ConnectionString="<%$ConnectionStrings:Northwind %>" SelectCommand =
17 "SELECT CategoryID, CategoryName, Description FROM Categories">
18         </asp:SqlDataSource>
19         <asp:FormView ID="FormView1" runat="server"
20 DataSourceID="SqlDataSource1" AllowPaging="true">
21          <HeaderTemplate>
22            Product Catogory
23          </HeaderTemplate>
24          <ItemTemplate>
25            CategoryID:<%#Eval ( "CategoryID")%><br/>
26            CategoryName:<%#Eval ( "CategoryName")%><br/>
27            Description:<%#Eval ( "Description" )%>
28          </ItemTemplate>
          <pagerSettings Mode="Numeric" />
        </asp:FormView>
    </div>
    </form>
 </body>
</html>
```

29

The code behind file for the control is as shown below:

```
1
2     public partial class FormView: System.Web.UI.Page
3     {
4        protected void Page_Load( object sender, EventArgs e)
5        {
6         if(!this.IsPostBack)
7         {
8           FormView1.HeaderStyle.BackColor=System.Drawing.Color.SeaGreen;
9           FormView1.HeaderStyle.ForeColor=System.Drawing.Color.Snow;
10          FormView1.HeaderStyle.Font.Bold=true;
11          FormView1.PagerStyle.BackColor=System.Drawing.Color.ForestGreen;
12          FormView1.PagerStyle.ForeColor=System.Drawing.Color.AliceBlue;
13          FormView1.RowStyle.BackColor=System.Drawing.Color.Green;
14          FormView1.RowStyle.ForeColor=System.Drawing.Color.White;
15        }
16       }
     }
```

Output is:



5) Data List Control

The DataList control is used to display a repeated list of items that are bound to the control. There are different templates using which user can design the layout of the control. The template property are mentioned below:

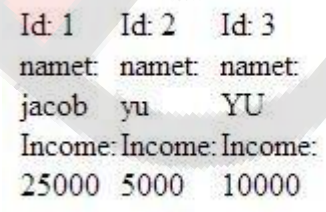
Template Property	Description
ItemTemplate	It contains the HTML elements and controls to render for each row in the data source
AlternatingItemTemplate	It contains the HTML elements and controls to render once for every other row in the data source
SelectedItemTemplate	It contains the elements to render when the user selects an item in the DataList control
EditItemTemplate	It specifies the layout of an item when the edit mode is working
HeaderTemplate	It contains all the text and controls to be rendered at the beginning of the list

FooterTemplate	It contains all the text and controls to be rendered at the end of the list
SeperatorTemplate	It contains all elements to render between each item

A sample code to demonstrate the DataList control is as shown below:

```
<%@Page Language="C#" %>
<! DOCTYPE html>
<script runat="server">
</script>
<html xmlns="http://www.w3.org/1999/xhtml">
 <head runat="server">
   <title></title>
 </head>
 <body>
   <form id="form1" runat="server">
     <div>
     <asp:DataList ID="DataList1" runat="server" DataKeyField="Id"
DataSourceID="SqlDataSource1" Height="285px" RepeatColumns="3"
RepeatDirection="Horizontal" Width="134px">
    <ItemTemplate>
    Id:
    <asp:Label ID="IdLabel" runat="server" Text='<%# Eval ( "Id" )%>' />
    <br/>
    name:
    <asp:Label ID="nameLabel" runat="server" Text='<%# Eval ( "name" )%>'
/>
    <br/>
    Income:
    <asp:Label ID="IncomeLabel" runat="server" Text='<%# Eval ( "Income"
)%>' />
    <br/>
    <br/>
       </ItemTemplate>
       </asp:DataList>
       <asp:SqlDataSource ID="SqlDataSource1" runat="server"
ConnectionString='<%$ConnectionStrings:ConnectionString %>"
       SelectCommand="SELECT * FROM [footerex]">
       </asp:SqlDataSource>
     </div>
   </form>
 </body>
</html>
```

Output is:



6) Repeater Control

The Repeater control is a data bound control created by using the templates to display items. The control does not support editing, paging or sorting of data rendered through the control. The list of templates supported by the Repeater control are as mentioned below:

Templates	Description
HeaderTemplate	It contains all the text and controls to be rendered at the beginning of the list
FooterTemplate	It contains all the text and controls to be rendered at the end of the list
AlternatingItemTemplate	It contains the HTML elements and controls to render once for every other row in the data source
SeperatorTemplate	It contains all elements to render between each item
ItemTemplate	It contains the HTML elements and controls to render for each row in the data source

The sample code for the Repeater control is as shown below:

```
?
1     <body>
2       <form id=”form1” runat=”server”>
3         <div>
4           <asp:Repeater ID=”RepeaterInformation” runat=”server”>
5         <HeaderTemplate>
6             <table class=”tblcolor”>
7             <tr>
8                 <b>
9                 <td>
10                Roll No
11                </td>
12                <td>
13                StudentName
14                </td>
15                <td>
16                Total Fees
17                </td>
18                </b>
19            </HeaderTemplate>
20            </ItemTemplate>
21            <tr class=”tblrowcolor”>
22            <td>
23                <%#DataBinder.Eval ( Contiane.”DataItem.RollNo”)%>
24            </td>
25            <td>
26                <%#DataBinder.Eval(Contianer,”DataItem.Name”)%>
            </td>
            <td>
                <%#DataBinder.Eval(Contianer.”DataItem.Fees”)%>
            </td>
            </tr>
```

```
            </ItemTemplate>
            <SeperatorTemplate>
            <tr>
            <td>
                <hr/>
            </td>
            <td>
                <hr/>
            </td>
            <td>
                <hr/>
            </td>
            </tr>
            </SeperatorTemplate>
            <FooterTemplate>
            <tr>
            <td>
                School Records displayed
            </td>
            </tr>
            </table>
            </FooterTemplate>
        </asp:Repeater>
         </div>
      </form>
    </body>
```

The code behind file is as shown below:

```
public partial class _Default: System.Web.UI.Page
{
  SqlConnection con;
  SqlCommand cmd = new SqlCommand();
  protected void Page_Load( object sender, EventArgs e)
  {
    con=new SqlConnection(
ConfigurationManager.ConnectionStrings["constr"].ConnectionString);
    cmd.Connection=con;
    com.Open();
    RepeatInformation.DataSource = cmd.ExecuteReader();
    RepeatInformation.DataBind();
    con.Close();
  }
}
```

Output is :

Roll No	Student Name	Total Fees
1	Patrick	3400
2	John	4500
3	York	1200
School Records displayed		

7) ListView control

The ListView control is used to bind to data items returned to the data source and display them. The control displays data in a format defined by using templates and styles. The list of templates supported by the control are as shown below:

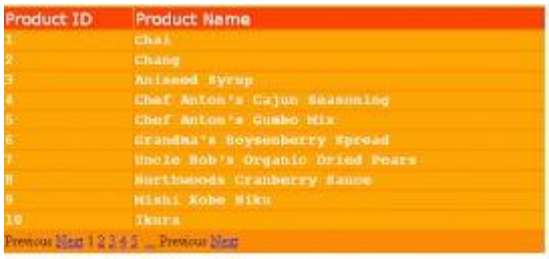
Templates	Description
ItemTemplate	It identifies the data bound content to display for single items
ItemSeperatorTemplate	It identifies the content to be rendered between the items
LayoutTemplate	It identifies the root template that defines the main layout
GroupTemplate	It identifies the content of the group layout
GroupSeperatorTemplate	It identifies the content to be rendered between the group of items
EmptyItemTemplate	It identifies the control to render for an empty item when the GroupTemplate is used
EmptyDataTemplate	It identifies the content to render if the data source returns no data
SelectedItemTemplate	It identifies the content to render for the selected data item to differentiate the selected item from the other displayed items
EditItemTemplate	It identifies the content to render when the item is lost
InsertItemTemplate	It identifies the content to render when an item is being inserted

A sample code for the list control is as shown below:

```
<html xmlns="http://www.w3.org/1999/xhtml">
 <head id"Head1" runat="server">
   <title>ListViewControl - how to use ListView control in asp.net
</title>
     <style type="text/css">
    .TableCSS
    {
      border-style:none;
      background-color:DarkOrange;
       width:600px;
    }
    .TableHeader
    {
       background-color:OrangeRed;
       color:snow;
       font-size:large;
       font-family:Verdana;
    }
    .TableData
    {
       Background-color:Orange;
       color:Snow;
```

```
        font-family:Courier New;
        font-size: medium;
        font-weight:bold;
         }
     </style>
</head>

<body>
   <form id="form1" runat="server">
     <div>
    <h2 style="color:Navy; font-style:italic;">ListView Control Example:
How to Use ListView Control</h2>
    <hr width="550" align="left" color="PowderBlue" />
    <asp:SqlDataSource ID="SqlDataSource1" runat="server"
ConnectionString="<%$ConnectionStrings:NorthwindConnectionString" %>"
    SelectCommand = "Select ProductID, ProductName From products Order By
ProductId"
    >
    </asp:SqlDataSource>
    <br/>
    <asp:ListView ID="ListView1" runat="server"
DataSourceID="SqlDataSource1" >
    <LayoutTemplate>
    <table runat="server" class="TableCSS">
       <tr runat="server" class="TableHeader">
        <td runat="server">ProductID</td>
        <td runat="server">ProductName</td>
       </tr>
       <tr id="ItemPlaceHolder" runat="server">
       <tr runat="server">
        <td runat="server" colspan="2">
        <asp:DataPager ID="DataPager1" runat="server">
        <Fields>
         <asp:NextPreviousPageField ButtonType="Link" />
        </Fields>
        </asp:DataPager>
        </td>
       </tr>
    </table>
    </LayoutTemplate>
      <ItemTemplate>
        <tr class="TableData" >
        <td>
          <asp:Label ID="Label1" runat="server"
Text='<%#Eval("ProductID")%>'>
            </asp:Label>
        </td>
        <td>
            <asp:Label ID="Label2" runat="server" Text='<%#Eval
("ProductName")%>'>
            </asp:Label>
        </td>
        </tr>
      </ItemTemplate>
    </asp:ListView>
     </div>
   </form>
</body>
```

Output is:



Product ID	Product Name
1	Chai
2	Chang
3	Aniseed Syrup
4	Chef Anton's Cajun Seasoning
5	Chef Anton's Gumbo Mix
6	Grandma's Boysenberry Spread
7	Uncle Bob's Organic Dried Pears
8	Northwoods Cranberry Sauce
9	Mishi Kobe Niku
10	Ikura

Previous Next 1 2 3 4 5 ... Previous Next

8) Data Pager control

Data Pager control is used to create interface for navigation through multiple pages containing multiple records. The DataPager control has some properties associated with it. The list of the properties is as mentioned below:

Property	Description
PageSize	It is used to get or set the number of data items to display in a page
PagedControlID	It is used to set the control ID for the page
StartRowIndex	It is used to get the index of the first item in a page
MaximumRow	It is used to get the maximum number of rows to retrieve from the database for a page
TotalRowCount	It is used to get the total number of rows or items in the datasource

A sample code to demonstrate the Data Pager control is as shown below:

?

```
1  <html xmlns="http://www.w3.org/1999/xhtml">
2  <head id="head1" runat="server">
3      <title>DataPager control - how to use DataPager control in asp.net
   </title>
4      <style type="text/css">
5      .TableCSS
6      {
7          border-style:none;
8          background-color:OrangeRed;
9          width:600px;
       }
10     .TableHeader
11     {
12         background-color:DeepPink;
13         color:snow;
14         font-size:large;
15         font-family:Verdana;
16         height:45px;
       }
17     .TableData
18     {
19         background-color:HotPink;
20         color:snow;
21         font-size:medium;
           font-family:Courier New;
```

```
        font-weight:bold;
        height:30px;
    }
    .TablePager
    {
        background-color:DarkSeaGreen;
        height:50px;
    }
    .ButtonCSS
    {
        color:Green;
        height:35px;
        font-weight:bold;
    }
    .NumericButtonCSS
    {
        font-size: x-large;
        font-family: Courier New;
        color:Green;
        font-weight:bold;
    }
    .CurrentPageLabelCSS
    {
        color:Green;
        font-size:xx-large;
        font-family:Courier New;
        font-weight:bold;
    }
    .NextPreviousButtonCSS
    {
        color:Green;
        font-size:large;
        font-family:Courier New;
        font-weight:bold;
    }
</style>
  </head>
```

Output is:

5.4) ASP.NET STATE MANAGEMENT-CLIENT SIDE AND SERVER SIDE

5.4.1)CLIENT SIDE

ASP.NET client side coding has two aspects:

- **Client side scripts** : It runs on the browser and in turn speeds up the execution of page. For example, client side data validation which can catch invalid data and warn the user accordingly without making a round trip to the server.
- **Client side source code** : ASP.NET pages generate this. For example, the HTML source code of an ASP.NET page contains a number of hidden fields and automatically injected blocks of JavaScript code, which keeps information like view state or does other jobs to make the page work.

Client Side Scripts

All ASP.NET server controls allow calling client side code written using JavaScript or VBScript. Some ASP.NET server controls use client side scripting to provide response to the users without posting back to the server. For example, the validation controls.

Apart from these scripts, the Button control has a property OnClientClick, which allows executing client-side script, when the button is clicked.

The traditional and server HTML controls have the following events that can execute a script when they are raised:

Event	Description
onblur	When the control loses focus
onfocus	When the control receives focus
onclick	When the control is clicked
onchange	When the value of the control changes
onkeydown	When the user presses a key
onkeypress	When the user presses an alphanumeric key
onkeyup	When the user releases a key
onmouseover	When the user moves the mouse pointer over the control
onserverclick	It raises the ServerClick event of the control, when the control is clicked

Client Side Source Code

We have already discussed that, ASP.NET pages are generally written in two files:

- The content file or the markup file (.aspx)
- The code-behind file

The content file contains the HTML or ASP.NET control tags and literals to form the structure of the page. The code behind file contains the class definition. At run-time, the content file is parsed and transformed into a page class.

This class, along with the class definition in the code file, and system generated code, together make the executable code (assembly) that processes all posted data, generates response, and sends it back to the client.

Consider the simple page:

```
<%@ Page Language="C#" AutoEventWireup="true" CodeBehind="Default.aspx.cs"
   Inherits="clientside._Default" %>

<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
   "http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">

<html xmlns="http://www.w3.org/1999/xhtml" >

   <head runat="server">
      <title>
         Untitled Page
      </title>
   </head>

   <body>
      <form id="form1" runat="server">

         <div>
            <asp:TextBox ID="TextBox1" runat="server"></asp:TextBox>
            <asp:Button ID="Button1" runat="server"
OnClick="Button1_Click" Text="Click" />
         </div>

         <hr />

         <h3> <asp:Label ID="Msg" runat="server" Text=""> </asp:Label>
</h3>
      </form>
   </body>
```

```
</html>
```

When this page is run on the browser, the View Source option shows the HTML page sent to the browser by the ASP.Net runtime:

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
   "http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">

<html xmlns="http://www.w3.org/1999/xhtml" >

   <head>
      <title>
         Untitled Page
      </title>
   </head>

   <body>
      <form name="form1" method="post" action="Default.aspx" id="form1">

         <div>
            <input type="hidden" name="__VIEWSTATE" id="__VIEWSTATE"

value="/wEPDwUKMTU5MTA2ODYwOWRk31NudGDgvhhA7joJum9Qn5RxU2M=" />
         </div>

         <div>
            <input type="hidden" name="__EVENTVALIDATION"
id="__EVENTVALIDATION"

value="/wEWAwKpjZj0DALs0bLrBgKM54rGBhHsyM61rraxE+KnBTCS8cd1QDJ/"/>
         </div>

         <div>
            <input name="TextBox1" type="text" id="TextBox1" />
            <input type="submit" name="Button1" value="Click" id="Button1"
/>
         </div>

         <hr />
         <h3><span id="Msg"></span></h3>

      </form>
   </body>
</html>
```

If you go through the code properly, you can see that first two tags contain the hidden fields which store the view state and validation information.

5.4.2) SERVER SIDE

We have studied the page life cycle and how a page contains various controls. The page itself is instantiated as a control object. All web forms are basically instances of the ASP.NET Page class. The page class has the following extremely useful properties that correspond to intrinsic objects:

- Session
- Application
- Cache
- Request
- Response
- Server
- User
- Trace

We will discuss each of these objects in due time. In this tutorial we will explore the Server object, the Request object, and the Response object.

Server Object

The Server object in Asp.NET is an instance of the System.Web.HttpServerUtility class. The HttpServerUtility class provides numerous properties and methods to perform various jobs.

Properties and Methods of the Server object

The methods and properties of the HttpServerUtility class are exposed through the intrinsic Server object provided by ASP.NET.

The following table provides a list of the properties:

Property	Description
MachineName	Name of server computer
ScriptTimeOut	Gets and sets the request time-out value in seconds.

The following table provides a list of some important methods:

Method	Description
CreateObject(String)	Creates an instance of the COM object identified by its ProgID (Programmatic ID).
CreateObject(Type)	Creates an instance of the COM object identified by its Type.

Equals(Object)	Determines whether the specified Object is equal to the current Object.
Execute(String)	Executes the handler for the specified virtual path in the context of the current request.
Execute(String, Boolean)	Executes the handler for the specified virtual path in the context of the current request and specifies whether to clear the QueryString and Form collections.
GetLastError	Returns the previous exception.
GetType	Gets the Type of the current instance.
HtmlEncode	Changes an ordinary string into a string with legal HTML characters.
HtmlDecode	Converts an Html string into an ordinary string.
ToString	Returns a String that represents the current Object.
Transfer(String)	For the current request, terminates execution of the current page and starts execution of a new page by using the specified URL path of the page.
UrlDecode	Converts an URL string into an ordinary string.
UrlEncodeToken	Works same as UrlEncode, but on a byte array that contains Base64-encoded data.
UrlDecodeToken	Works same as UrlDecode, but on a byte array that contains Base64-encoded data.
MapPath	Return the physical path that corresponds to a specified virtual file path on the server.
Transfer	Transfers execution to another web page in the current application.

Request Object

The request object is an instance of the System.Web.HttpRequest class. It represents the values and properties of the HTTP request that makes the page loading into the browser.

The information presented by this object is wrapped by the higher level abstractions (the web control model). However, this object helps in checking some information such as the client browser and cookies.

Properties and Methods of the Request Object

The following table provides some noteworthy properties of the Request object:

Property	Description
AcceptTypes	Gets a string array of client-supported MIME accept types.
ApplicationPath	Gets the ASP.NET application's virtual application root path on the server.
Browser	Gets or sets information about the requesting client's browser capabilities.
ContentEncoding	Gets or sets the character set of the entity-body.
ContentLength	Specifies the length, in bytes, of content sent by the client.
ContentType	Gets or sets the MIME content type of the incoming request.
Cookies	Gets a collection of cookies sent by the client.
FilePath	Gets the virtual path of the current request.
Files	Gets the collection of files uploaded by the client, in multipart MIME format.
Form	Gets a collection of form variables.
Headers	Gets a collection of HTTP headers.
HttpMethod	Gets the HTTP data transfer method (such as GET, POST, or HEAD) used by the client.
InputStream	Gets the contents of the incoming HTTP entity body.
IsSecureConnection	Gets a value indicating whether the HTTP connection uses secure sockets (that is, HTTPS).
QueryString	Gets the collection of HTTP query string variables.
RawUrl	Gets the raw URL of the current request.
RequestType	Gets or sets the HTTP data transfer method (GET or POST) used by the client.
ServerVariables	Gets a collection of Web server variables.
TotalBytes	Gets the number of bytes in the current input stream.
Url	Gets information about the URL of the current request.

UrlReferrer	Gets information about the URL of the client's previous request that is linked to the current URL.
UserAgent	Gets the raw user agent string of the client browser.
UserHostAddress	Gets the IP host address of the remote client.
UserHostName	Gets the DNS name of the remote client.
UserLanguages	Gets a sorted string array of client language preferences.

The following table provides a list of some important methods:

Method	Description
BinaryRead	Performs a binary read of a specified number of bytes from the current input stream.
Equals(Object)	Determines whether the specified object is equal to the current object. (Inherited from object.)
GetType	Gets the Type of the current instance.
MapImageCoordinates	Maps an incoming image-field form parameter to appropriate x-coordinate and y-coordinate values.
MapPath(String)	Maps the specified virtual path to a physical path.
SaveAs	Saves an HTTP request to disk.
ToString	Returns a String that represents the current object.
ValidateInput	Causes validation to occur for the collections accessed through the Cookies, Form, and QueryString properties.

Response Object

The Response object represents the server's response to the client request. It is an instance of the System.Web.HttpResponse class.

In ASP.NET, the response object does not play any vital role in sending HTML text to the client, because the server-side controls have nested, object oriented methods for rendering themselves.

However, the HttpResponse object still provides some important functionalities, like the cookie feature and the Redirect() method. The Response.Redirect() method allows transferring the user to another page, inside as well as outside the application. It requires a round trip.

Properties and Methods of the Response Object

The following table provides some noteworthy properties of the Response object:

Property	Description
Buffer	Gets or sets a value indicating whether to buffer the output and send it after the complete response is finished processing.
BufferOutput	Gets or sets a value indicating whether to buffer the output and send it after the complete page is finished processing.
Charset	Gets or sets the HTTP character set of the output stream.
ContentEncoding	Gets or sets the HTTP character set of the output stream.
ContentType	Gets or sets the HTTP MIME type of the output stream.
Cookies	Gets the response cookie collection.
Expires	Gets or sets the number of minutes before a page cached on a browser expires.
ExpiresAbsolute	Gets or sets the absolute date and time at which to remove cached information from the cache.
HeaderEncoding	Gets or sets an encoding object that represents the encoding for the current header output stream.
Headers	Gets the collection of response headers.
IsClientConnected	Gets a value indicating whether the client is still connected to the server.
Output	Enables output of text to the outgoing HTTP response stream.
OutputStream	Enables binary output to the outgoing HTTP content body.
RedirectLocation	Gets or sets the value of the Http Location header.
Status	Sets the status line that is returned to the client.
StatusCode	Gets or sets the HTTP status code of the output returned to the client.
StatusDescription	Gets or sets the HTTP status string of the output returned to the client.
SubStatusCode	Gets or sets a value qualifying the status code of the response.
SuppressContent	Gets or sets a value indicating whether to send HTTP content to the client.

The following table provides a list of some important methods:

Method	Description
AddHeader	Adds an HTTP header to the output stream. AddHeader is provided for compatibility with earlier versions of ASP.
AppendCookie	Infrastructure adds an HTTP cookie to the intrinsic cookie collection.
AppendHeader	Adds an HTTP header to the output stream.
AppendToLog	Adds custom log information to the InterNET Information Services (IIS) log file.
BinaryWrite	Writes a string of binary characters to the HTTP output stream.
ClearContent	Clears all content output from the buffer stream.
Close	Closes the socket connection to a client.
End	Sends all currently buffered output to the client, stops execution of the page, and raises the EndRequest event.
Equals(Object)	Determines whether the specified object is equal to the current object.
Flush	Sends all currently buffered output to the client.
GetType	Gets the Type of the current instance.
Pics	Appends a HTTP PICS-Label header to the output stream.
Redirect(String)	Redirects a request to a new URL and specifies the new URL.
Redirect(String, Boolean)	Redirects a client to a new URL. Specifies the new URL and whether execution of the current page should terminate.
SetCookie	Updates an existing cookie in the cookie collection.
ToString	Returns a String that represents the current Object.
TransmitFile(String)	Writes the specified file directly to an HTTP response output stream, without buffering it in memory.
Write(Char)	Writes a character to an HTTP response output stream.
Write(Object)	Writes an object to an HTTP response stream.
Write(String)	Writes a string to an HTTP response output stream.

WriteFile(String)	Writes the contents of the specified file directly to an HTTP response output stream as a file block.
WriteFile(String, Boolean)	Writes the contents of the specified file directly to an HTTP response output stream as a memory block.

Example

The following simple example has a text box control where the user can enter name, a button to send the information to the server, and a label control to display the URL of the client computer.

The content file:

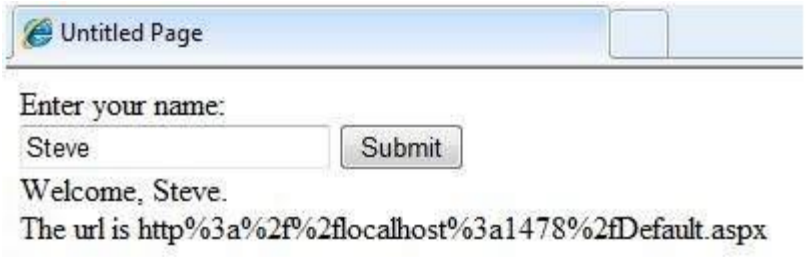
```
<%@ Page Language="C#" AutoEventWireup="true" CodeBehind="Default.aspx.cs"
   Inherits="server_side._Default" %>

<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
   "http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">

<html xmlns="http://www.w3.org/1999/xhtml" >

   <head runat="server">
      <title>Untitled Page</title>
   </head>

   <body>
      <form id="form1" runat="server">
         <div>

            Enter your name:
            <br />
            <asp:TextBox ID="TextBox1" runat="server"></asp:TextBox>
            <asp:Button ID="Button1" runat="server"
OnClick="Button1_Click" Text="Submit" />
            <br />
            <asp:Label ID="Label1" runat="server"/>

         </div>
      </form>
   </body>

</html>
```

The code behind Button1_Click:

```
protected void Button1_Click(object sender, EventArgs e) {

   if (!String.IsNullOrEmpty(TextBox1.Text)) {

      // Access the HttpServerUtility methods through
      // the intrinsic Server object.
      Label1.Text = "Welcome, " + Server.HtmlEncode(TextBox1.Text) + ".
<br/> The url is " + Server.UrlEncode(Request.Url.ToString())
```

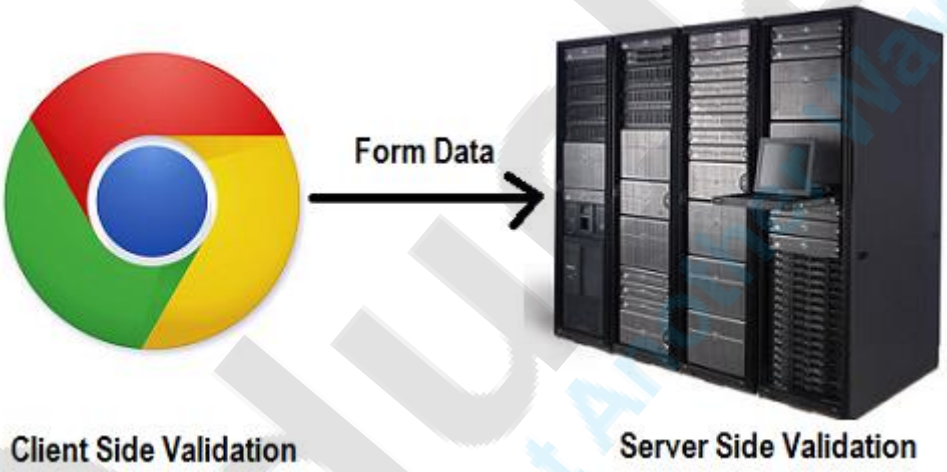
```
    }
}
```

Run the page to see the following result:



Client side validation Vs server side validation

There is a usual question that which type of validations is better or best? Server side validation or Client side Validation ?



Validations can be performed on the server side or on the client side (web browser). The user input validation take place on the Server Side during a post back session is called Server Side Validation and the user input validation take place on the Client Side (web browser) is called Client Side Validation. Client Side Validation does not require a postback. If the user request requires server resources to validate the user input, you should use Server Side Validation. If the user request does not require any server resources to validate the input , you can use Client Side Validation.

Server Side Validation

In the Server Side Validation, the input submitted by the user is being sent to the server and validated using one of server side scripting languages such as ASP.Net, PHP etc. After the validation process on the Server Side, the feedback is sent back to the client by a new dynamically generated web page. It is better to validate user input on Server Side because you can protect against the malicious users, who can easily bypass your Client Side scripting language and submit dangerous input to the server.

Client Side Validation

In the Client Side Validation you can provide a better user experience by responding quickly at the browser level. When you perform a Client Side Validation, all the user inputs validated in the user's browser itself. Client Side validation does not require a round trip to the server, so the network traffic which will help your server perform better. This type of validation is done on the browser side using script languages such as JavaScript, VBScript or HTML5 attributes.

For example, if the user enter an invalid email format, you can show an error message immediately before the user move to the next field, so the user can correct every field before they submit the form.

Mostly the Client Side Validation depends on the JavaScript Language, so if users turn JavaScript off, it can easily bypass and submit dangerous input to the server . So the Client Side Validation can not protect your application from malicious attacks on your server resources and databases.

As both the validation methods have their own significances, it is recommended that the Server side validation is more SECURE!

5.5) ASP.NET and AJAX

- AJAX stands for Asynchronous JavaScript and XML. This is a cross platform technology which speeds up response time. The AJAX server controls add script to the page which is executed and processed by the browser.
- However like other ASP.NET server controls, these AJAX server controls also can have methods and event handlers associated with them, which are processed on the server side.
- The control toolbox in the Visual Studio IDE contains a group of controls called the 'AJAX Extensions'

The ScriptManager Control

The ScriptManager control is the most important control and must be present on the page for other controls to work.

It has the basic syntax:

```
<asp:ScriptManager ID="ScriptManager1" runat="server">

</asp:ScriptManager>
```

If you create an 'Ajax Enabled site' or add an 'AJAX Web Form' from the 'Add Item' dialog box, the web form automatically contains the script manager control. The ScriptManager control takes care of the client-side script for all the server side controls.

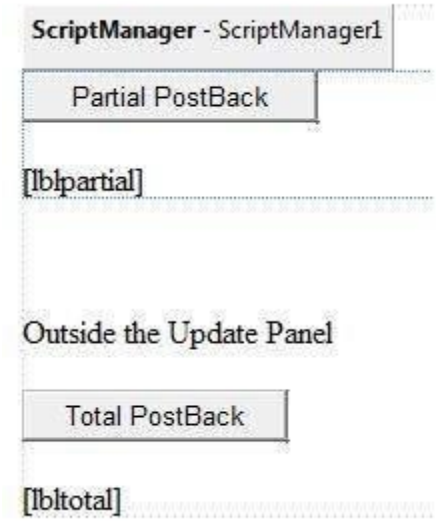
The UpdatePanel Control

The UpdatePanel control is a container control and derives from the Control class. It acts as a container for the child controls within it and does not have its own interface. When a control inside it triggers a post back, the UpdatePanel intervenes to initiate the post asynchronously and update just that portion of the page.

For example, if a button control is inside the update panel and it is clicked, only the controls within the update panel will be affected, the controls on the other parts of the page will not be affected. This is called the partial post back or the asynchronous post back.

Example

- Add an AJAX web form in your application. It contains the script manager control by default. Insert an update panel. Place a button control along with a label control within the update panel control. Place another set of button and label outside the panel.
- The design view looks as follows:



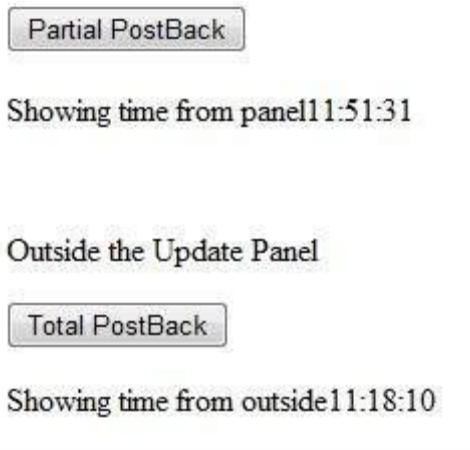
The source file is as follows:

```
<form id="form1" runat="server">
  <div>
    <asp:ScriptManager ID="ScriptManager1" runat="server" />
  </div>

  <asp:UpdatePanel ID="UpdatePanel1" runat="server">
    <ContentTemplate>
      <asp:Button ID="btnpartial" runat="server" onclick="btnpartial_Click" Text="Partial
PostBack"/>
      <br />
      <br />
      <asp:Label ID="lblpartial" runat="server"></asp:Label>
    </ContentTemplate>
  </asp:UpdatePanel>

  <p> </p>
  <p>Outside the Update Panel</p>
  <p>
    <asp:Button ID="btntotal" runat="server" onclick="btntotal_Click" Text="Total PostBack" />
  </p>

  <asp:Label ID="lbltotal" runat="server"></asp:Label>
</form>
```

Both the button controls have same code for the event handler:

```
string time = DateTime.Now.ToLongTimeString();

lblpartial.Text = "Showing time from panel" + time;

lbltotal.Text = "Showing time from outside" + time;
```

Observe that when the page is executed, if the total post back button is clicked, it updates time in both the labels but if the partial post back button is clicked, it only updates the label within the update panel.



A page can contain multiple update panels with each panel containing other controls like a grid and displaying different part of data.

When a total post back occurs, the update panel content is updated by default. This default mode could be changed by changing the UpdateMode property of the control. Let us look at other properties of the update panel.

Properties of the UpdatePanel Control

The following table shows the properties of the update panel control:

Properties	Description
ChildrenAsTriggers	This property indicates whether the post backs are coming from the child controls, which cause the update panel to refresh.
ContentTemplate	It is the content template and defines what appears in the update panel when it is rendered.
ContentTemplateContainer	Retrieves the dynamically created template container object and used for adding child controls programmatically.
IsInPartialRendering	Indicates whether the panel is being updated as part of the partial post back.
RenderMode	Shows the render modes. The available modes are Block and Inline.
UpdateMode	Gets or sets the rendering mode by determining some conditions.

Triggers	Defines the collection trigger objects each corresponding to an event causing the panel to refresh automatically.

Methods of the UpdatePanel Control

The following table shows the methods of the update panel control:

Methods	Description
CreateContentTemplateContainer	Creates a Control object that acts as a container for child controls that define the UpdatePanel control's content.
CreateControlCollection	Returns the collection of all controls that are contained in the UpdatePanel control.
Initialize	Initializes the UpdatePanel control trigger collection if partial-page rendering is enabled.
Update	Causes an update of the content of an UpdatePanel control.

The behavior of the update panel depends upon the values of the UpdateMode property and ChildrenAsTriggers property.

UpdateMode	ChildrenAsTriggers	Effect
Always	False	Illegal parameters.
Always	True	UpdatePanel refreshes if whole page refreshes or a child control on it posts back.
Conditional	False	UpdatePanel refreshes if whole page refreshes or a triggering control outside it initiates a refresh.
Conditional	True	UpdatePanel refreshes if whole page refreshes or a child control on it posts back or a triggering control outside it initiates a refresh.

The UpdateProgress Control

- The UpdateProgress control provides a sort of feedback on the browser while one or more update panel controls are being updated. For example, while a user logs in or waits for server response while performing some database oriented job.
- It provides a visual acknowledgement like "Loading page...", indicating the work is in progress.

The syntax for the UpdateProgress control is:

```
<asp:UpdateProgress ID="UpdateProgress1" runat="server" DynamicLayout="true"
AssociatedUpdatePanelID="UpdatePanel1" >

  <ProgressTemplate>
    Loading...
  </ProgressTemplate>

</asp:UpdateProgress>
```

The above snippet shows a simple message within the ProgressTemplate tag. However, it could be an image or other relevant controls. The UpdateProgress control displays for every asynchronous postback unless it is assigned to a single update panel using the AssociatedUpdatePanelID property.

Properties of the UpdateProgress Control

The following table shows the properties of the update progress control:

Properties	Description
AssociatedUpdatePanelID	Gets and sets the ID of the update panel with which this control is associated.
Attributes	Gets or sets the cascading style sheet (CSS) attributes of the UpdateProgress control.
DisplayAfter	Gets and sets the time in milliseconds after which the progress template is displayed. The default is 500.
DynamicLayout	Indicates whether the progress template is dynamically rendered.
ProgressTemplate	Indicates the template displayed during an asynchronous post back which takes more time than the DisplayAfter time.

Methods of the UpdateProgress Control

The following table shows the methods of the update progress control:

Methods	Description
GetScriptDescriptors	Returns a list of components, behaviors, and client controls that are required for the UpdateProgress control's client functionality.

GetScriptReferences	Returns a list of client script library dependencies for the UpdateProgress control.

The Timer Control

The timer control is used to initiate the post back automatically. This could be done in two ways:

(1) Setting the Triggers property of the UpdatePanel control:

```
<Triggers>

   <asp:AsyncPostBackTrigger ControlID="btnpanel2" EventName="Click" />

</Triggers>
```

(2) Placing a timer control directly inside the UpdatePanel to act as a child control trigger. A single timer can be the trigger for multiple UpdatePanels.

```
<asp:UpdatePanel ID="UpdatePanel1" runat="server" UpdateMode="Always">

  <ContentTemplate>
    <asp:Timer ID="Timer1" runat="server" Interval="1000">
      </asp:Timer>

    <asp:Label ID="Label1" runat="server" Height="101px" style="width:304px" >
      </asp:Label>
  </ContentTemplate>

</asp:UpdatePanel>
```

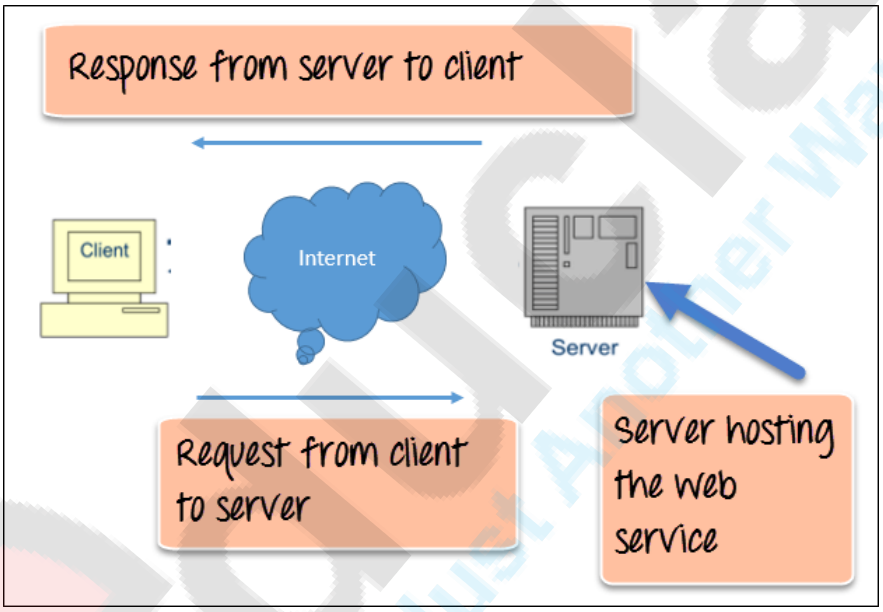
UNIT 6

WEB SERVICES

What is Web Service?

- Web service is a standardized medium to propagate communication between the client and server applications on the World Wide Web.

- A web service is a software module which is designed to perform a certain set of tasks.
  - The web services can be searched for over the network and can also be invoked accordingly.
  - When invoked the web service would be able to provide functionality to the client which invokes that web service.



Web Service Architecture Diagram

- The above diagram shows a very simplistic view of how a web service would actually work. The client would invoke a series of web service calls via requests to a server which would host the actual web service.
- These requests are made through what is known as remote procedure calls. Remote Procedure Calls(RPC) are calls made to methods which are hosted by the relevant web service.
- As an example, Amazon provides a web service that provides prices for products sold online via amazon.com. The front end or presentation layer can be in .Net or Java but either programming language would have the ability to communicate with the web service.
- The main component of a web service is the data which is transferred between the client and the server, and that is XML. XML (Extensible markup language) is a counterpart to HTML and easy to understand the intermediate language that is understood by many programming languages.

- So when applications talk to each other, they actually talk in XML. This provides a common platform for application developed in various programming languages to talk to each other.
- Web services use something known as SOAP (Simple Object Access Protocol) for sending the XML data between applications. The data is sent over normal HTTP. The data which is sent from the web service to the application is called a SOAP message. The SOAP message is nothing but an XML document. Since the document is written in XML, the client application calling the web service can be written in any programming language.

6.1) XML

XML stands for **E**xtensible **M**arkup **L**anguage. It is a text-based markup language derived from Standard Generalized Markup Language (SGML).

XML tags identify the data and are used to store and organize the data, rather than specifying how to display it like HTML tags, which are used to display the data. XML is not going to replace HTML in the near future, but it introduces new possibilities by adopting many successful features of HTML.

There are three important characteristics of XML that make it useful in a variety of systems and solutions –

- **XML is extensible** – XML allows you to create your own self-descriptive tags, or language, that suits your application.
- **XML carries the data, does not present it** – XML allows you to store the data irrespective of how it will be presented.
- **XML is a public standard** – XML was developed by an organization called the World Wide Web Consortium (W3C) and is available as an open standard.

XML Usage

A short list of XML usage says it all –

- XML can work behind the scene to simplify the creation of HTML documents for large web sites.
- XML can be used to exchange the information between organizations and systems.
- XML can be used for offloading and reloading of databases.
- XML can be used to store and arrange the data, which can customize your data handling needs.

- XML can easily be merged with style sheets to create almost any desired output.
- Virtually, any type of data can be expressed as an XML document.

What is Markup?

XML is a markup language that defines set of rules for encoding documents in a format that is both human-readable and machine-readable. So *what exactly is a markup language?* Markup is information added to a document that enhances its meaning in certain ways, in that it identifies the parts and how they relate to each other. More specifically, a markup language is a set of symbols that can be placed in the text of a document to demarcate and label the parts of that document.

Following example shows how XML markup looks, when embedded in a piece of text –

```
<message>
   <text>Hello, world!</text>
</message>
```

This snippet includes the markup symbols, or the tags such as <message>...</message> and <text>... </text>. The tags <message> and </message> mark the start and the end of the XML code fragment. The tags <text> and </text> surround the text Hello, world!.

6.2)Web service Architecture

Every framework needs some sort of architecture to make sure the entire framework works as desired. Similarly, in web services, there is an architecture which consists of three distinct roles as given below

1. **Provider** - The provider creates the web service and makes it available to client application who want to use it.
2. **Requestor** - A requestor is nothing but the client application that needs to contact a web service. The client application can be a .Net, Java, or any other language based application which looks for some sort of functionality via a web service.
3. **Broker** - The broker is nothing but the application which provides access to the UDDI. The UDDI, as discussed in the earlier topic enables the client application to locate the web service.

The diagram below showcases how the Service provider, the Service requestor and Service registry interact with each other.

4. **Publish** - A provider informs the broker (service registry) about the existence of the web service by using the broker's publish interface to make the service accessible to clients
5. **Find** - The requestor consults the broker to locate a published web service
6. **Bind** - With the information it gained from the broker(service registry) about the web service, the requestor is able to bind, or invoke, the web service.

Web service Characteristics

Web services have the following special behavioral characteristics:

7. **They are XML-Based** - Web Services uses XML to represent the data at the representation and data transportation layers. Using XML eliminates any networking, operating system, or platform sort of dependency since XML is the common language understood by all.

8. **Loosely Coupled** – Loosely coupled means that the client and the web service are not bound to each other, which means that even if the web service changes over time, it should not change the way the client calls the
9. web service. Adopting a loosely coupled architecture tends to make software systems more manageable and allows simpler integration between different systems.
10. **Synchronous or Asynchronous functionality**- Synchronicity refers to the binding of the client to the execution of the service. In synchronous operations, the client will actually wait for the web service to complete an operation.

- An example of this is probably a scenario wherein a database read and write operation are being performed. If data is read from one database and subsequently written to another, then the operations have to be done in a sequential manner.
- Asynchronous operations allow a client to invoke a service and then execute other functions in parallel.
- This is one of the common and probably the most preferred techniques for ensuring that other services are not stopped when a particular operation is being carried out.

11..**Ability to support Remote Procedure Calls (RPCs)** - Web services enable clients to invoke procedures, functions, and methods on remote objects using an XML-based protocol. Remote procedures expose input and output parameters that a web service must support.

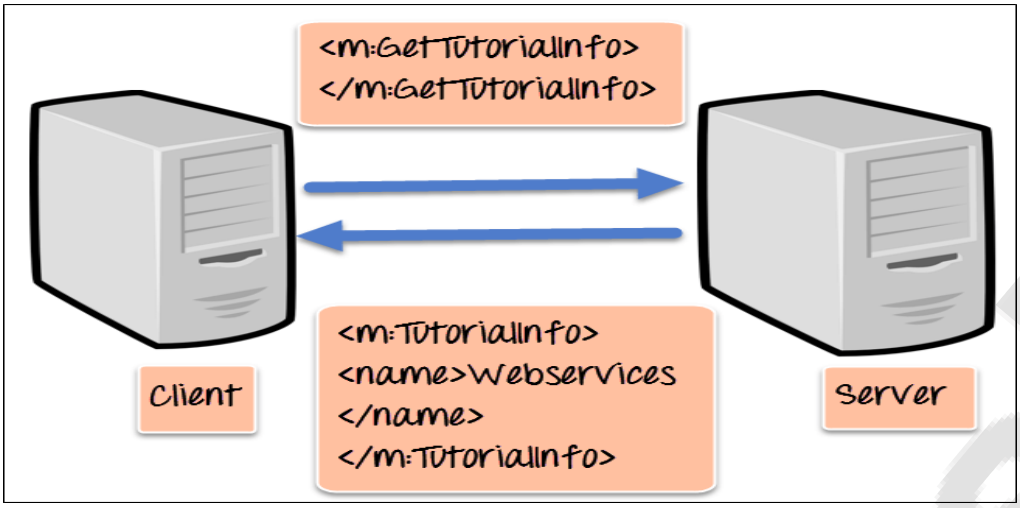
11. **Supports Document Exchange** - One of the key benefits of XML is its generic way of representing not only data but also complex documents. These documents can be as simple as representing a current address, or they can be as complex as representing an entire book.

6.3)Simple Object Access Protocol (SOAP)

- SOAP is known as a transport-independent messaging protocol. SOAP is based on transferring XML data as SOAP Messages. Each message has something which is known as an XML document. Only the structure of the XML document follows a specific pattern, but not the content. The best part of Web services and SOAP is that its all sent via HTTP, which is the standard web protocol.
- Here is what a SOAP message consists of

Each SOAP document needs to have a root element known as the <Envelope> element. The root element is the first element in an XML document.

- The "envelope" is in turn divided into 2 parts. The first is the header, and the next is the body.
- The header contains the routing data which is basically the information which tells the XML document to which client it needs to be sent to.
    - The body will contain the actual message.
    -
- The diagram below shows a simple example of the communication via SOAP.



SOAP is an XML-based protocol for accessing web services over HTTP. It has some specification which could be used across all applications.

SOAP is known as the Simple Object Access Protocol SOAP was developed as an intermediate language so that applications built on various programming languages could talk easily to each other and avoid the extreme development effort.

- SOAP Introduction
- Advantages of SOAP
- SOAP Building blocks
- SOAP Message Structure
- SOAP Envelope Element
- SOAP Communication Model
- Practical SOAP Example

- But data exchange between these heterogeneous applications would be complex. So will be the complexity of the code to accomplish this data exchange.
- One of the methods used to combat this complexity is to use XML (Extensible Markup Language) as the intermediate language for exchanging data between applications.
- Every programming language can understand the XML markup language. Hence, XML was used as the underlying medium for data exchange.
- But there are no standard specifications on use of XML across all programming languages for data exchange. That is where SOAP comes in.
- SOAP was designed to work with XML over HTTP and have some sort of specification which could be used across all applications.

Advantages of SOAP

SOAP is the protocol used for data interchange between applications. Below are some of the reasons as to why SOAP is used.

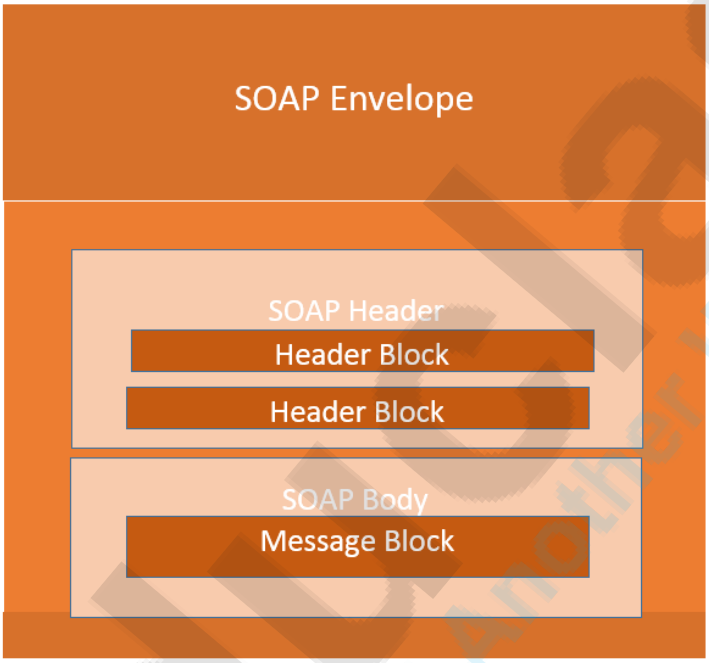
- When developing Web services, you need to have some of language which can be used for web services to talk with client applications. SOAP is the perfect medium which was developed in order to achieve this purpose. This protocol is also recommended by the W3C consortium which is the governing body for all web standards.
- SOAP is a light-weight protocol that is used for data interchange between applications. Note the keyword '**light**.' Since SOAP is based on the XML language, which itself is a light weight data interchange language, hence SOAP as a protocol that also falls in the same category.

- SOAP is designed to be platform independent and is also designed to be operating system independent. So the SOAP protocol can work any programming language based applications on both Windows and Linux platform.
- It works on the HTTP protocol –SOAP works on the HTTP protocol, which is the default protocol used by all web applications. Hence, there is no sort of customization which is required to run the web services built on the SOAP protocol to work on the World Wide Web.

SOAP FORMAT

The SOAP specification defines something known as a "**SOAP message**" which is what is sent to the web service and the client application.

The diagram below shows the various building blocks of a SOAP Message.



The SOAP message is nothing but a mere XML document which has the below components.

- An Envelope element that identifies the XML document as a SOAP message – This is the containing part of the SOAP message and is used to encapsulate all the details in the SOAP message. This is the root element in the SOAP message.
- A Header element that contains header information – The header element can contain information such as authentication credentials which can be used by the calling application. It can also contain the definition of complex types which could be used in the SOAP message. By default, the SOAP message can contain parameters which could be of simple types such as strings and numbers, but can also be a complex object type.

A simple example of a complex type is shown below.

Suppose we wanted to send a structured data type which had a combination of a "Tutorial Name" and a "Tutorial Description," then we would define the complex type as shown below.

The complex type is defined by the element tag <xsd:complexType>. All of the required elements of the structure along with their respective data types are then defined in the complex type collection.

```
<xsd:complexType>
```

```
<xsd:sequence>
            <xsd:element name="Tutorial Name" type="string"/>
            <xsd:element name="Tutorial Description"  type="string"/>
 </xsd:sequence>
</xsd:complexType>
```

- A Body element that contains call and response information – This element is what contains the actual data which needs to be sent between the web service and the calling application. Below is an example of the SOAP body which actually works on the complex type defined in the header section. Here is the response of the Tutorial Name and Tutorial Description that is sent to the calling application which calls this web service.

```
<soap:Body>
  <GetTutorialInfo>
                    <TutorialName>Web Services</TutorialName>
                    <TutorialDescription>All about web services</TutorialDescription>
  </GetTutorialInfo>
</soap:Body>
```

SOAP Message Structure

One thing to note is that SOAP messages are normally auto-generated by the web service when it is called. Whenever a client application calls a method in the web service, the web service will automatically generate a SOAP message which will have the necessary details of the data which will be sent from the web service to the client application.

As discussed in the previous topic, a simple SOAP Message has the following elements –

- The Envelope element
- The header element and
- The body element
- The Fault element (Optional)

Let's look at an example below of a simple SOAP message and see what element actually does.

1.As seen from the above SOAP message, the first part of the SOAP message is the envelope element which is used to encapsulate the entire SOAP message.

1.The next element is the SOAP body which contains the details of the actual message.

2.Our message contains a web service which has the name of "Guru99WebService".

3.The "Guru99Webservice" accepts a parameter of the type 'int' and has the name of TutorialID.
Now, the above SOAP message will be passed between the web service and the client application.

You can see how useful the above information is to the client application. The SOAP message tells the client application what is the name of the Web service, and also what parameters it expects and also what is the type of each parameter which is taken by the web service.

SOAP Envelope Element

- The first bit of the building block is the SOAP Envelope.
- The SOAP Envelope is used to encapsulate all of the necessary details of the SOAP messages, which are exchanged between the web service and the client application.
- The SOAP envelope element is used to indicate the beginning and end of a SOAP message. This enables the client application which calls the web service to know when the SOAP message ends.

The following points can be noted on the SOAP envelope element.

- Every SOAP message needs to have a root Envelope element. It is absolutely mandatory for SOAP essage to have an envelope element.
- Every Envelope element needs to have at least one soap body element.
- If an Envelope element contains a header element, it must contain no more than one, and it must appear as the first child of the Envelope, before the body element.
- The envelope changes when SOAP versions change.
- A v1.1-compliant SOAP processor generates a fault upon receiving a message containing the v1.2 envelope namespace.
- A v1.2-compliant SOAP processor generates a Version Mismatch fault if it receives a message that does not include the v1.2 envelope namespace.
- Below is an example of version 1.2 of the SOAP envelope element.

```
<?xml version="1.0"?>
<SOAP-ENV:Envelope      xmlns:SOAP-ENV="http://www.w3.org/2001/12/soap-envelope"      SOAP-
ENV:encodingStyle=" http://www.w3.org/2001/12/soap-encoding">
          <soap:Body>
        <Guru99WebService xmlns="http://tempuri.org/">
                  <TutorialID>int</TutorialID>
                </Guru99WebService>
          </soap:Body>
</SOAP-ENV:Envelope>
```

The Fault message

- When a request is made to a SOAP web service, the response returned can be of either 2 forms which are a successful response or an error response. When a success is generated, the response from the server will always be a SOAP message. But if SOAP faults are generated, they are returned as "HTTP 500" errors.
- The SOAP Fault message consists of the following elements.
- <**faultCode**>- This is the code that designates the code of the error. The fault code can be either of any below values
- SOAP-ENV:VersionMismatch – This is when an invalid namespace for the SOAP Envelope element is encountered.
    - SOAP-ENV:MustUnderstand - An immediate child element of the Header element, with the mustUnderstand attribute set to "1", was not understood.
    - SOAP-ENV:Client - The message was incorrectly formed or contained incorrect information.
    - SOAP-ENV:Server - There was a problem with the server, so the message could not proceed.

- **<faultString>** - This is the text message which gives a detailed description of the error.
- **<faultActor> (Optional)**- This is a text string which indicates who caused the fault.
- **<detail>(Optional)** - This is the element for application-specific error messages. So the application could have a specific error message for different business logic scenarios.

Example for Fault Message

An example of a fault message is given below. The error is generated if the scenario wherein the client tries to use a method called TutorialID in the class GetTutorial.
The below fault message gets generated in the event that the method does not exist in the defined class.

```
<?xml version='1.0' encoding='UTF-8'?>
<SOAP-ENV:Envelope                xmlns:SOAP-ENV="http://schemas.xmlsoap.org/soap/envelope/"
xmlns:xsi="http://www.w3.org/1999/XMLSchema-instance"
xmlns:xsd="http://www.w3.org/1999/XMLSchema">
      <SOAP-ENV:Body>
         <SOAP-ENV:Fault>
         <faultcode xsi:type="xsd:string">SOAP-ENV:Client</faultcode>
        <faultstring xsi:type="xsd:string">
            Failed to locate method (GetTutorialID) in class (GetTutorial)
         </faultstring>
    </SOAP-ENV:Fault>
   </SOAP-ENV:Body>
</SOAP-ENV:Envelope>
```

Output:

When you execute the above code, it will show the error like "Failed to locate method (GetTutorialID) in class (GetTutorial)"

6.4)Universal Description, Discovery, and Integration(UDDI)

UDDI is an XML-based standard for describing, publishing, and finding web services.

- UDDI stands for **Universal Description, Discovery, and Integration.**
- UDDI is a specification for a distributed registry of web services.
- UDDI is a platform-independent, open framework.
- UDDI can communicate via SOAP, CORBA, Java RMI Protocol.
- UDDI uses Web Service Definition Language(WSDL) to describe interfaces to web services.
- UDDI is seen with SOAP and WSDL as one of the three foundation standards of web services.
- UDDI is an open industry initiative, enabling businesses to discover each other and define how they interact over the Internet.

UDDI has two sections −

- A registry of all web service's metadata, including a pointer to the WSDL description of a service.
- A set of WSDL port type definitions for manipulating and searching that registry.

Private UDDI Registries

- As an alternative to using the public federated network of UDDI registries available on the Internet, companies or industry groups may choose to implement their own private UDDI registries.
- These exclusive services are designed for the sole purpose of allowing members of the company or of the industry group to share and advertise services amongst themselves.

- Regardless of whether the UDDI registry is a part of the global federated network or a privately owned and operated registry, the one thing that ties them all together is a common web services API for publishing and locating businesses and services advertised within the UDDI registry.

A business or a company can register three types of information into a UDDI registry. This information is contained in three elements of UDDI.

- These three elements are −
- White Pages,
- Yellow Pages, and
- Green Pages.

White Pages

- **White pages contain −**
- Basic information about the company and its business.
- Basic contact information including business name, address, contact phone number, etc.
- A Unique identifiers for the company tax IDs. This information allows others to discover your web service based upon your business identification.

Yellow Pages

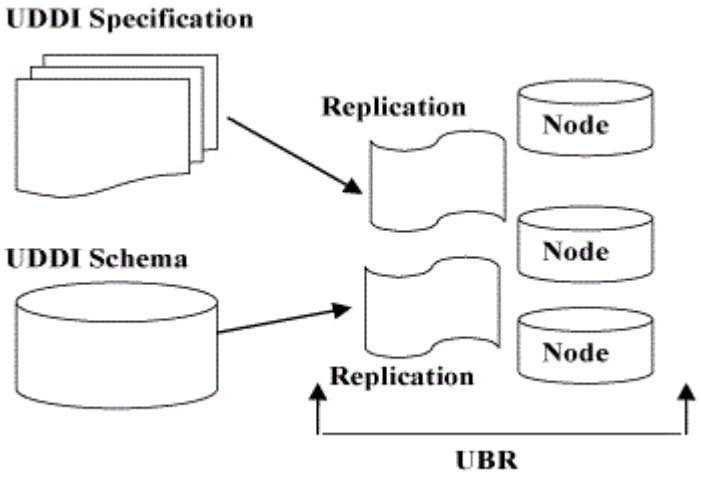
- Yellow pages contain more details about the company. They include descriptions of the kind of electronic capabilities the company can offer to anyone who wants to do business with it.
- Yellow pages uses commonly accepted industrial categorization schemes, industry codes, product codes, business identification codes and the like to make it easier for companies to search through the listings and find exactly what they want.

Green Pages

- Green pages contains technical information about a web service. A green page allows someone to bind to a Web service after it's been found. It includes −
- The various interfaces
- The URL locations
- Discovery information and similar data required to find and run the Web service.

UDDI data model

- UDDI Data Model is an XML Schema for describing businesses and web services. The data model is described in detail in the "UDDI Data Model" chapter.
- **UDDI API Specification**
- It is a specification of API for searching and publishing UDDI data.
- **UDDI Cloud Services**
- These are operator sites that provide implementations of the UDDI specification and synchronize all data on a scheduled basis.



- The UDDI Business Registry (UBR), also known as the Public Cloud, is a conceptually single system built from multiple nodes having their data synchronized through replication.
- The current cloud services provide a logically centralized, but physically distributed, directory. It means the data submitted to one root node will automatically be replicated across all the other root nodes. Currently, data replication occurs every 24 hours.
- UDDI cloud services are currently provided by Microsoft and IBM. Ariba had originally planned to offer an operator as well, but has since backed away from the commitment. Additional operators from other companies, including Hewlett-Packard, are planned for the near future.
- It is also possible to set up private UDDI registries. For example, a large company may set up its own private UDDI registry for registering all internal web services. As these registries are not automatically synchronized with the root UDDI nodes, they are not considered as a part of the UDDI cloud.

UDDI data model

UDDI includes an XML Schema that describes the following data structures −

- **businessEntity**
- **businessService**
- **bindingTemplate**
- **tModel**
- **publisherAssertion**

businessEntity Data Structure

- The business entity structure represents the provider of web services. Within the UDDI registry, this structure contains information about the company itself, including contact information, industry categories, business identifiers, and a list of services provided.

Here is an example of a fictitious business's UDDI registry entry −

```
<businessEntity businessKey = "uuid:C0E6D5A8-C446-4f01-99DA-70E212685A40"
  operator = "http://www.ibm.com" authorizedName = "John Doe">
  <name>Acme Company</name>
  <description>
    We create cool Web services
```

```
  </description>

  <contacts>
    <contact useType = "general info">
      <description>General Information</description>
      <personName>John Doe</personName>
      <phone>(123) 123-1234</phone>
      <email>jdoe@acme.com</email>
    </contact>
  </contacts>

  <businessServices>
    ...
  </businessServices>
  <identifierBag>
    <keyedReference tModelKey = "UUID:8609C81E-EE1F-4D5A-B202-3EB13AD01823"
      name = "D-U-N-S" value = "123456789" />
  </identifierBag>

  <categoryBag>
    <keyedReference tModelKey = "UUID:C0B9FE13-179F-413D-8A5B-5004DB8E5BB2"
      name = "NAICS" value = "111336" />
  </categoryBag>
</businessEntity>
```

businessService Data Structure

- The business service structure represents an individual web service provided by the business entity. Its description includes information on how to bind to the web service, what type of web service it is, and what taxonomical categories it belongs to.
- Here is an example of a business service structure for the Hello World web service.
-

```
<businessService serviceKey = "uuid:D6F1B765-BDB3-4837-828D-8284301E5A2A"
  businessKey = "uuid:C0E6D5A8-C446-4f01-99DA-70E212685A40">
  <name>Hello World Web Service</name>
  <description>A friendly Web service</description>
  <bindingTemplates>
    ...
  </bindingTemplates>
  <categoryBag />
</businessService>
```

Notice the use of the Universally Unique Identifiers (UUIDs) in the *businessKey* and *serviceKey* attributes. Every business entity and business service is uniquely identified in all the UDDI registries through the UUID assigned by the registry when the information is first entered.

bindingTemplate Data Structure

- Binding templates are the technical descriptions of the web services represented by the business service structure. A single business service may have multiple binding templates. The binding template represents the actual implementation of the web service.
- Here is an example of a binding template for Hello World.

```
<bindingTemplate serviceKey = "uuid:D6F1B765-BDB3-4837-828D-8284301E5A2A"
```

```
  bindingKey = "uuid:C0E6D5A8-C446-4f01-99DA-70E212685A40">
  <description>Hello World SOAP Binding</description>
  <accessPoint URLType = "http">http://localhost:8080</accessPoint>

  <tModelInstanceDetails>
    <tModelInstanceInfo tModelKey = "uuid:EB1B645F-CF2F-491f-811A-4868705F5904">
      <instanceDetails>
        <overviewDoc>
          <description>
            references the description of the WSDL service definition
          </description>

          <overviewURL>
            http://localhost/helloworld.wsdl
          </overviewURL>
        </overviewDoc>
      </instanceDetails>
    </tModelInstanceInfo>
  </tModelInstanceDetails>
</bindingTemplate>
```

As a business service may have multiple binding templates, the service may specify different implementations of the same service, each bound to a different set of protocols or a different network address.

tModel Data Structure

- tModel is the last core data type, but potentially the most difficult to grasp. tModel stands for technical model.
- tModel is a way of describing the various business, service, and template structures stored within the UDDI registry. Any abstract concept can be registered within the UDDI as a tModel. For instance, if you define a new WSDL port type, you can define a tModel that represents that port type within the UDDI. Then, you can specify that a given business service implements that port type by associating the tModel with one of that business service's binding templates.

Here is an example of a tModel representing the Hello World Interface port type.

```
<tModel tModelKey = "uuid:xyz987..." operator = "http://www.ibm.com"
  authorizedName = "John Doe">
  <name>HelloWorldInterface Port Type</name>
  <description>
    An interface for a friendly Web service
  </description>

  <overviewDoc>
    <overviewURL>
      http://localhost/helloworld.wsdl
    </overviewURL>
  </overviewDoc>
</tModel>
```

publisherAssertion Data Structure

- This is a relationship structure putting into association two or more businessEntity structures according to a specific type of relationship, such as subsidiary or department.
- The publisherAssertion structure consists of the three elements: fromKey (the first businessKey), toKey (the second businessKey), and keyedReference.
- The keyedReference designates the asserted relationship type in terms of a keyName keyValue pair within a tModel, uniquely referenced by a tModelKey.

```
<element name = "publisherAssertion" type = "uddi:publisherAssertion" />
<complexType name = "publisherAssertion">
  <sequence>
    <element ref = "uddi:fromKey" />
    <element ref = "uddi:toKey" />
    <element ref = "uddi:keyedReference" />
  </sequence>
</complexType>
```

UDDI interfaces

A registry is of no use without some way to access it. The UDDI standard version 2.0 specifies two interfaces for service consumers and service providers to interact with the registry.

- Service consumers use **Inquiry Interface** to find a service, and service providers use **Publisher Interface** to list a service.
- The core of the UDDI interface is the UDDI XML Schema definitions. These define the fundamental UDDI data types through which all the information flows.

The Publisher Interface

- The Publisher Interface defines sixteen operations for a service provider managing its entries in the UDDI registry −
- **get_authToken** − Retrieves an authorization token. All of the Publisher interface operations require that a valid authorization token be submitted with the request.
- **discard_authToken** − Tells the UDDI registry to no longer accept a given authorization token. This step is equivalent to logging out of the system.
- **save_business** − Creates or updates a business entity's information contained in the UDDI registry.
- **save_service** − Creates or updates information about the web services that a business entity provides.
- **save_binding** − Creates or updates the technical information about a web service's implementation.
- **save_tModel** − Creates or updates the registration of abstract concepts managed by the UDDI registry.
- **delete_business** − Removes the given business entities from the UDDI registry completely.
- **delete_service** − Removes the given web services from the UDDI registry completely.
- **delete_binding** − Removes the given web services technical details from the UDDI registry.
- **delete_tModel** − Removes the specified tModels from the UDDI registry.
- **get_registeredInfo** − Returns a summary of everything the UDDI registry is currently keeping track of for the user, including all businesses, all services, and all tModels.
- **set_publisherAssertions** − Manages all of the tracked relationship assertions associated with an individual publisher account.
- **add_publisherAssertions** − Causes one or more publisherAssertions to be added to an individual publisher's assertion collection.
- **delete_publisherAssertions** − Causes one or more publisherAssertion elements to be removed from a publisher's assertion collection.
- **get_assertionStatusReport** − Provides administrative support for determining the status of current and outstanding publisher assertions that involve any of the business registrations managed by the individual publisher account.
- **get_publisherAssertions** − Obtains the full set of publisher assertions that is associated with an individual publisher account.

The Inquiry Interface

- The inquiry interface defines ten operations for searching the UDDI registry and retrieving details about specific registrations −

- **find_binding** − Returns a list of web services that match a particular set of criteria based on the technical binding information.
- **find_business** − Returns a list of business entities that match a particular set of criteria.
- **find_ltservice** − Returns a list of web services that match a particular set of criteria.
- **find_tModel** − Returns a list of tModels that match a particular set of criteria.
- **get_bindingDetail** − Returns the complete registration information for a particular web service binding template.
- **get_businessDetail** − Returns the registration information for a business entity, including all services that entity provides.
- **get_businessDetailExt** − Returns the complete registration information for a business entity.
- **get_serviceDetail** − Returns the complete registration information for a web service.
- **get_tModelDetail** − Returns the complete registration information for a tModel.
- **find_relatedBusinesses** − Discovers businesses that have been related via the uddi-org:relationships model.

6.5)Web Services Description Language(WSDL).

- WSDL is an XML-based file which basically tells the client application what the web service does. It is known as the Web Services Description Language(WSDL).
- In this tutorial, we are going to focus on the last point which is the most important part of web services, and that is the WSDL or the Web services description language.
- The WSDL file is used to describe in a nutshell what the web service does and gives the client all the information required to connect to the web service and use all the functionality provided by the web service.

Structure of a WSDL Document

- A WSDL document is used to describe a web service. This description is required, so that client applications are able to understand what the web service actually does.
- The WSDL file contains the location of the web service and
- The methods which are exposed by the web service.
- The WSDL file itself can look very complex to any user, but it contains all the necessary information that any client application would require to use the relevant web service

- Below is the general structure of a WSDL file

- **Definition**
- **TargetNamespace**
- **DataTypes**
- **Messages**
- **Porttype**
- **Bindings**
- **service**

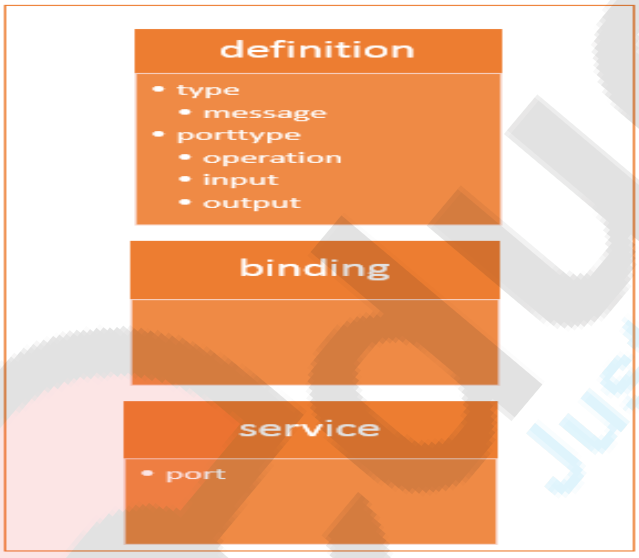
➢ One key thing to note here is that definition of messages, which is what is passed by the SOAP protocol is actually defined in the WSDL document.

- The WSDL document actually tells a client application what are the types of SOAP messages which are sent and accepted by the Web service.

- In other words, the WSDL is just like a postcard which has the address of a particular location. The address provides the details of the person who delivered the postcard. Hence, in the same way, the WSDL file is the postcard, which has the address of the web service which can deliver all the functionality that the client wants.

```
<!-- WSDL definition structure -->
<definitions
                        name="Guru99Service"
      targetNamespace=http://example.org/math/
      xmlns=http://schemas.xmlsoap.org/wsdl/>
            <!-- abstract definitions -->
                        <types> ...
                                    <message> ...
                                    <portType> ...

  <!-- concrete definitions -->
                        <binding> ...
                        <service> ...
</definition>
```

Below is a diagram of the structure of a WSDL file



WSDL Elements

The WSDL file contains the following main parts

- The **<types>** tag is used to define all the complex datatypes, which will be used in the message exchanged between the client application and the web service. This is an important aspect of the client application, because if the web service works with a complex data type, then the client application should know how to process the complex data type. Data types such as float, numbers, and strings are all simple data types, but there could be structured data types which may be provided by the web service.

For example, there could be a data type called EmployeeDataType which could have 2 elements called "EmployeeName" of type string and "EmployeeID" of type number or integer. Together they form a data structure which then becomes a complex data type.

1. The **<messages>** tag is used to define the message which is exchanged between the client application and the web server. These messages will explain the input and output operations which can be performed by the web service. An example of a message can be a message which accepts the EmployeeID of an employee, and the output message can be the name of the employee based on the EmpoyeeID provided.

2. The **<portType>** tag is used to encapsulate every input and output message into one logical operation. So there could be an operation called "GetEmployee" which combines the input message of accepting the EmployeeID from a client application and then sending the EmployeeName as the output message

3. The **<binding>** tag is used to bind the operation to the particular port type. This is so that when the client application calls the relevant port type, it will then be able to access the operations which are bound to this port type. Port types are just like interfaces. So if a client application needs to use a web service they need to use the binding information to ensure that they can connect to the interface provided by that web service.

4. 4.The **<service>** tag is a name given to the web service itself. Initially, when a client application makes a call to the web service, it will do by calling the name of the web service. For example, a web service can be located at an address such as **http://localhost/Guru99/Tutorial.asmx** . The service tag will actually have the URL defined as **http://localhost/Guru99/Tutorial.asmx**, which will actually tell the client application that there is a web service available at this location.

Creating WSDL File

- The WSDL file gets created whenever a web service is built in any programming language.
- Since the WSDL file is pretty complicated to be generated from plain scratch, all editors such as Visual Studio for .Net and Eclipse for Java automatically create the WSDL file.
- Below is an example of a WSDL file created in Visual Studio.

```
<?xml version="1.0"?>
<definitions name="Tutorial"
                        targetNamespace=http://Guru99.com/Tutorial.wsdl
      xmlns:tns=http://Guru99.com/Tutorial.wsdl
      xmlns:xsd1=http://Guru99.com/Tutorial.xsd
      xmlns:soap=http://schemas.xmlsoap.org/wsdl/soap/
      xmlns="http://schemas.xmlsoap.org/wsdl/">
  <types>
                        <schema targetNamespace=http://Guru99.com/Tutorial.xsd
      xmlns="http://www.w3.org/2000/10/XMLSchema">

      <element name="TutorialNameRequest">
            <complexType>
            <all>
            <element name="TutorialName" type="string"/>
           </all>
        </complexType>
     </element>
     <element name="TutorialIDRequest">
                        <complexType>
            <all>
```

```
            <element name="TutorialID" type="number"/>
          </all>
        </complexType>
    </element>
  </schema>
</types>
<message name="GetTutorialNameInput">
            <part name="body" element="xsd1:TutorialIDRequest"/>
</message>
<message name="GetTutorialNameOutput">
            <part name="body" element="xsd1:TutorialNameRequest"/>
</message>
<portType name="TutorialPortType">
            <operation name="GetTutorialName">
            <input message="tns:GetTutorialNameInput"/>
     <output message="tns:GetTutorialNameOutput"/>
  </operation>
 </portType>
 <binding name="TutorialSoapBinding" type="tns:TutorialPortType">
 <soap:binding style="document" transport="http://schemas.xmlsoap.org/soap/http"/>
             <operation name="GetTutorialName">
                        <soap:operation soapAction="http://Guru99.com/GetTutorialName"/>
            <input>
            <soap:body use="literal"/>
        </input>
     <output>
  <soap:body use="literal"/>
</output>
</operation>
</binding>

<service name="TutorialService">
            <documentation>TutorialService</documentation>
  <port name="TutorialPort" binding="tns:TutorialSoapBinding">
            <soap:address location="http://Guru99.com/Tutorial"/>
  </port>
 </service>
</definitions>
```

6.6)Creating and consuming the web services[SIMPLE]

Motivation of the XML Web Services

Most of the people does not understand that why we need a web service and make a wrong use of it. The main idea of a web service is to join two businesses together since they cannot join due to their graphical locations. Web Service is also used to link different systems together. The best thing about this is that the systems can be in different in nature. Meaning a web service enables a

windows application to interact and communicate with the linux application. This is done by using XML as the transferring medium. Since XML is understood by all the systems because its nothing but plain text that's why this is a perfect language to join systems together. XML web Service should never be used to transfer confidential data, not to be used in real time programming where time is the essence.

Making the Web Service

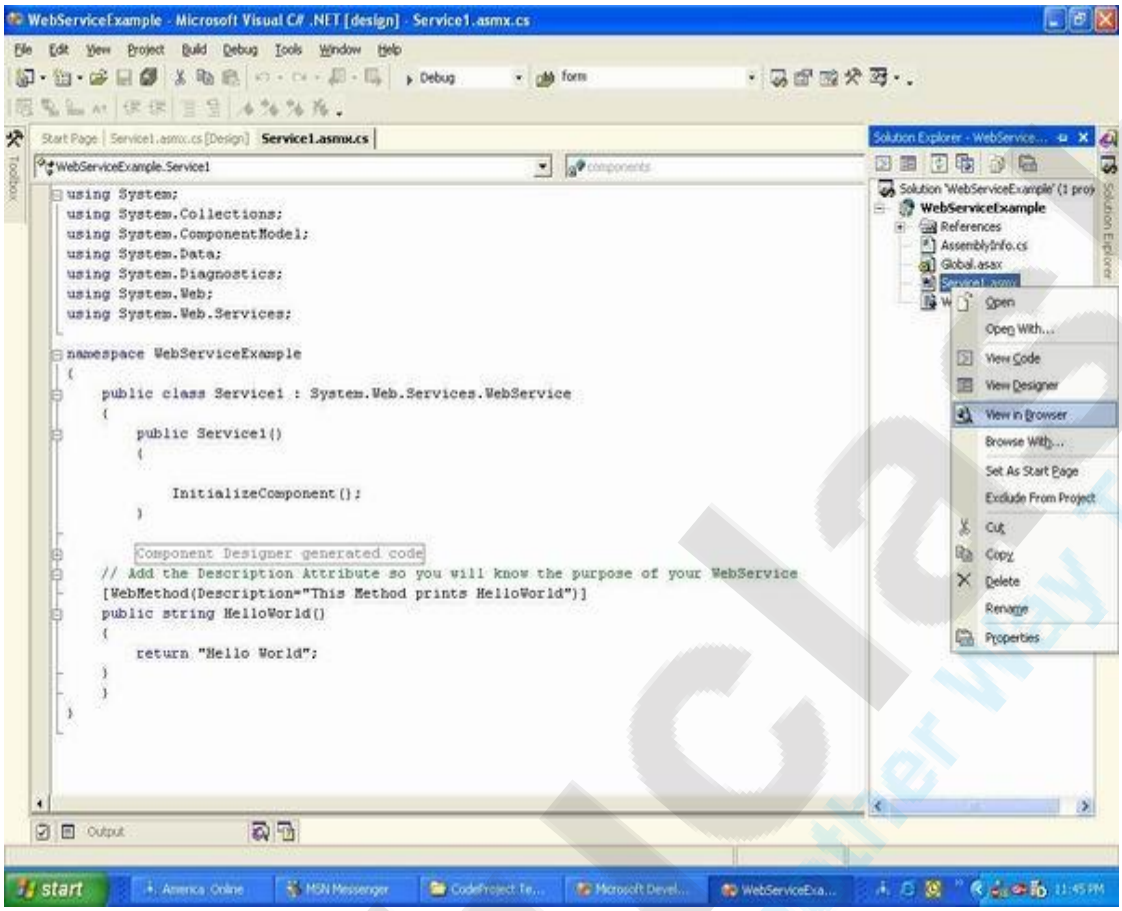
First, start your Visual Studio .NET, and in the project type, select ASP.NET WebService. In the left pane, choose the language of your choice. In this example, I will be using Visual C#.NET. Once you select the project, a page will appear which will be more like a design page, switch to its code view. In the code view, you can see lot of comments and C# code already written for you. You will also see that at the bottom, there is a method `HelloWorld` which is written for you by default, so you can test your service and of course say hello to the world. After removing the unwanted code and comments, your code will look like this:

```
using System;
using System.Collections;
using System.ComponentModel;
using System.Data;
using System.Diagnostics;
using System.Web;
using System.Web.Services;

namespace WebServiceExample
{
public class Service1 : System.Web.Services.WebService
{
public Service1()
{

InitializeComponent();
}


// Add the Description Attribute so you
// will know the purpose of your WebService
[WebMethod(Description="This Method prints HelloWorld")]
public string HelloWorld()
{
return "Hello World";
}
}
}
```

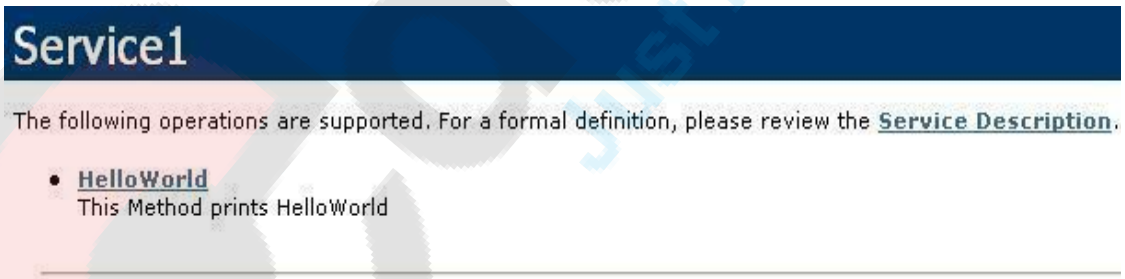
Let's dig into this small code. `[WebMethod]` Attribute denotes that this method will be used by the clients, also this method has to be public in order for a client to use it. Description inside the `WebMethod` Attribute just gives the method more meaning. Don't worry about the **`InitializeComponent()`** method since it's written by default.

Running the Web Service

OK, now you have made your first kick ass WebService (without even writing a single line of code). Let's run it and check whether it gives the correct result or not. In the Solution Explorer, right click on the *.asmx* file, and select *View in Browser* as shown below:

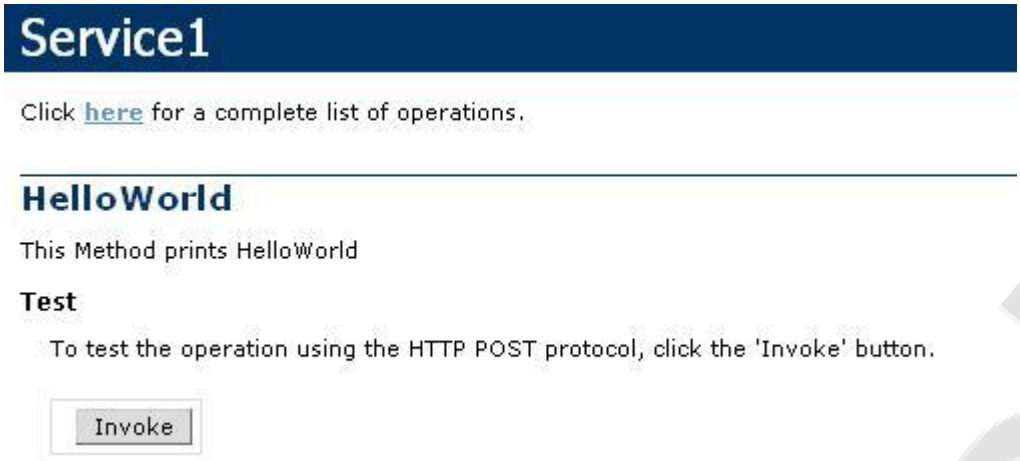


Once you click on "View in Browser", the next screen you will see will be something like this:



have erased most of the stuff, so you can only see the method that you need in this example. Below, you can see the method HelloWorld and the description you wrote for the method.

Now, click on the HelloWorld method. I don't want to scare you with all the SOAP and HTTP code produced, so I am only going to paste the screen shot which will be relevant to this example.



Alright, so far so good. Now, just press the Invoke button to see the result of your method named `HelloWorld()`.

```
<?xml version="1.0" encoding="utf-8" ?>
<string xmlns="http://tempuri.org/">Hello World</string>
```

You have just tested your first webservice and it ran since you didn't coded it. I know what you are thinking right now. Is the client going to see the result like this strange format (this is XML format). Well, of course not. That's why you need to make a Proxy class which consumes this service.

<p}< p="" style="color: rgb(0, 0, 0); font-family: "Droid Sans"; font-size: 14px; font-style: normal; font-variant-ligatures: normal; font-variant-caps: normal; font-weight: 400; letter-spacing: normal; orphans: 2; text-align: start; text-indent: 0px; text-transform: none; white-space: normal; widows: 2; word-spacing: 0px; -webkit-text-stroke-width: 0px; background-color: rgb(255, 255, 255); text-decoration-style: initial; text-decoration-color: initial;">

Making the Web Service Client

Let's make a Console application which consumes this service. You can use any language and platform to consume this service, that's the purpose of XML WebService. Now, this procedure requires some mouse clicking :). So I will write down the steps instead of pasting the screen shots.

1. Start a new project which will be a Console Application in Visual C#.NET.
2. Once you see the code view in the Console application, right click on the project name from the Solution Explorer. Remember that project name will be written in bold.
3. Click on "Add Web Reference".
4. Paste the URL of your WebService. You can get the URL of your WebService when you view your webservice in IE or any other browser.

5. Click GO.

6. Your webservice will be loaded. In the Web Reference Name textbox, write “MyService” and click Add Reference.

7. You will see that the web reference has been added in your Solution Explorer, meaning that webservice is ready to kick some butt.

Now, all you have to do is to make the instance of the WebService class using the reference name that you provided, which is “MyService”.

```
using System;

namespace MyClient
{
class Class1
{

[STAThread]
static void Main(string[] args)
{
// Make an instance of the WebService Class
// using the Web Reference you provided
MyService.Service1 service = new MyService.Service1();
// Assign message what ever is returned
// from HelloWorld in this case “HelloWorld”
string message = service.HelloWorld();
// Prints out the message on the screen
Console.WriteLine(message);

}
}
}
```

And that’s it. You use the webservice class just like any other class. Something you need to keep in mind is that if you decide to make a new method in your webservice and want to make it available to the client, then always remember to build your webservice solution so that the assembly can be updated. If you don’t build your webservice, you won’t be able to see the methods on the client side.

Http Clients Creating .NET Consumers

A consumer is the one who uses the service and a provider is the one who provides the service. Http clients can be an asp.net application or any other application that communicates with the web service through internet.

Web Services and Legacy Clients

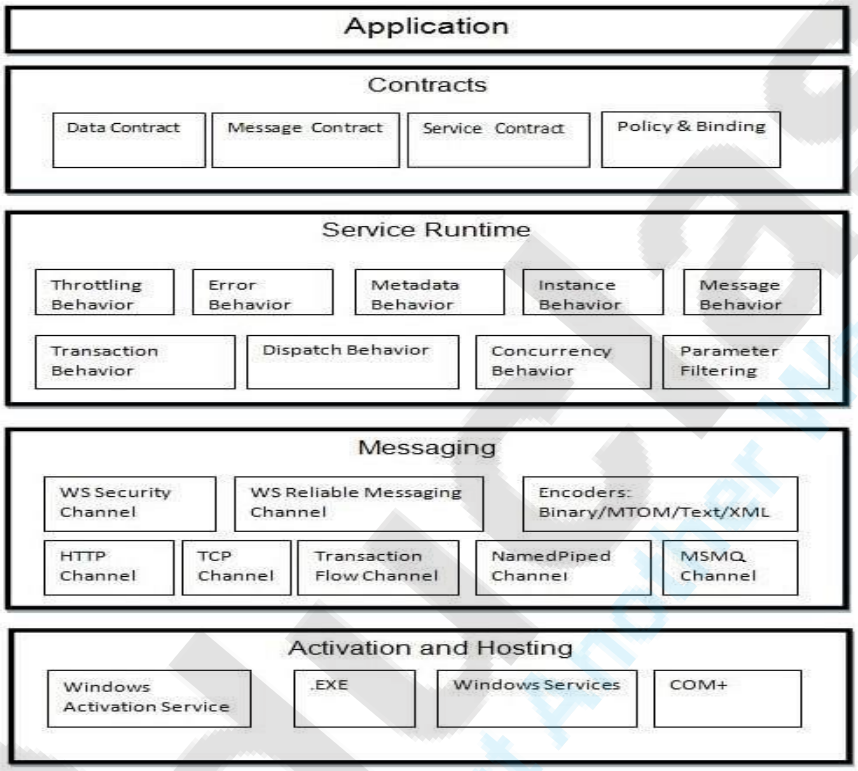
As I said previously that Web Services are used to communicate between different systems. These systems can be writting different operating systems but their communication medium

is xml which they all can understand. These legacy systems can be COBOL, FORTON and PASCAL. So instead of making a complete new application in Asp.net Web Service can be used to link different systems together.

6.7)WCF ARCHITECTURE

6.7.1) Architecture

WCF has a layered architecture that offers ample support for developing various distributed applications. The architecture is explained below in detail.



Contracts

- The contracts layer is just next to the application layer and contains information similar to that of a real-world contract that specifies the operation of a service and the kind of accessible information it will make. Contracts are basically of four types discussed below in brief –
- **Service contract** − This contract provides information to the client as well as to the outer world about the offerings of the endpoint, and the protocols to be used in the communication process.
- **Data contract** − The data exchanged by a service is defined by a data contract. Both the client and the service has to be in agreement with the data contract.
- **Message contract** − A data contract is controlled by a message contract. It primarily does the customization of the type formatting of the SOAP message parameters. Here, it should be mentioned that WCF employs SOAP format for the purpose of communication. SOAP stands for Simple Object Access Protocol.
- **Policy and Binding** − There are certain pre-conditions for communication with a service, and such conditions are defined by policy and binding contract. A client needs to follow this contract.

Service Runtime

- The service runtime layer is just below the contracts layer. It specifies the various service behaviors that occur during runtime. There are many types of behaviors that can undergo configuration and come under the service runtime

- **Throttling Behavior** − Manages the number of messages processed.
- **Error Behavior** − Defines the result of any internal service error occurrence.
- **Metadata Behavior** − Specifies the availability of metadata to the outside world.
- **Instance Behavior** − Defines the number of instances that needs to be created to make them available for the client.
- **Transaction Behavior** − Enables a change in transaction state in case of any failure.
- **Dispatch Behavior** − Controls the way by which a message gets processed by the infrastructure of WCF.
- **Concurrency Behavior** − Controls the functions that run parallel during a client-server communication.
- **Parameter Filtering** − Features the process of validation of parameters to a method before it gets invoked.

6.7.2)Endpoints

Messaging

This layer, composed of several channels, mainly deals with the message content to be communicated between two endpoints. A set of channels form a channel stack and the two major types of channels that comprise the channel stack are the following ones –

- **Transport Channels** − These channels are present at the bottom of a stack and are accountable for sending and receiving messages using transport protocols like HTTP, TCP, Peer-to-Peer, Named Pipes, and MSMQ.

- **Protocol Channels** − Present at the top of a stack, these channels also known as layered channels, implement wire-level protocols by modifying messages.

Activation and Hosting

The last layer of WCF architecture is the place where services are actually hosted or can be executed for easy access by the client. This is done by various mechanisms discussed below in brief.

- **IIS** − IIS stands for Internet Information Service. It offers a myriad of advantages using the HTTP protocol by a service. Here, it is not required to have the host code for activating the service code; instead, the service code gets activated automatically.
- **Windows Activation Service** − This is popularly known as WAS and comes with IIS 7.0. Both HTTP and non-HTTP based communication is possible here by using TCP or Namedpipe protocols.
- **Self-hosting** − This is a mechanism by which a WCF service gets self-hosted as a console application. This mechanism offers amazing flexibility in terms of choosing the desired protocols and setting own addressing scheme.
- **Windows Service** − Hosting a WCF service with this mechanism is advantageous, as the services then remain activated and accessible to the client due to no runtime activation.

6.7.3) TYPES OF CONTRACT

A WCF contract defines what a service does or what action a client can perform in the service. The contract is one of the elements of a WCF endpoint that contains information about the WCF

service. The contract also helps to serialize service information. There are two type of contracts, one is Service *Contracts, Data Contracts, Fault Contract* and *Message Contract.*

Service Contracts

The Service Contracts describes what action a client can perform in a service. This attribute is in the *System.ServiceModel namespace*. There are the following two types.

A. Service Contracts

1. Service Contract
2. Operation Contract

1. Service Contract

The Service Contract declares an interface in the WCF service for the client to get access to the interface.

```
1.  [ServiceContract]
2.  interface ICustomer
3.  {
4.
5.  }
```

2. Operation Contract

The Operation Contract declares a function inside the interface, the client will call this function. If you don't use the Operation contract in the preceding function then the client will not be able to call the function.

Example 1

```
1.  [OperationContract]
2.  Response AddNew(string customername);
```

Example 2

```
1.  [ServiceContract]
2.  interface ICustomer
3.  {
4.      [OperationContract]
5.      Response AddNew(string customername);
6.      Response Delete(int customerID);
7.  }
```

In the preceding Example 2 the clients will not be able to call the function name delete because the delete function has not used the **Operation Contract**.

B. Data Contracts

A Data Contract defines what data type to be passed to or from the client. In the WCF service, the Data Contract takes a major role for serialization and deserialization. There are two types of **Data Contracts.**

- **Data Contract**

  This contract declares and defines the class to be serialized for the client to access. If the Data Contract tag is not used then the class will not be serialized or deserialized.

```
[DataContract]
public class Customer
{
}
```

- **Data Member**

  This declares and defines properties inside a class, the property that doesn't use a Data Member tag will not be serialized or deserialized.

  Example

```
[DataContract]
public class Customer
{
    [DataMember]
    public int ID { get; set; }
    [DataMember]
    public string Name { get; set; }
        public string ContactNo { get; set; }
}
```

  In the preceding example the ContactNo property will not be given access to the client because it will not be serialized or deserialized though it is not used as a Data Member attribute.

C. Fault Contracts

In a simple WCF Service, errors/exceptions can be passed to the Client (WCF Service Consumer) using FaultContract. The fault contract defines the error to be raised by the service and how the service handles and propagates the error to its clients.

Example 1: Example of declaring a FaultContract in the service.

```
[ServiceContract]
public interface IService1
{
    [OperationContract]
    [FaultContract(typeof(ServiceData))]
    ServiceData TestConnection(string strConnectionString);
}
```

Example 2: Use of FaultContract in the client.

```
namespace FaultContractSampleWCF
{

    public class Service1 : IService1
    {
        public ServiceData TestConnection(string StrConnectionString)
        {
            ServiceData myServiceData = new ServiceData();
            try
            {
                SqlConnection con = new SqlConnection(StrConnectionString);
                con.Open();
                myServiceData.Result = true;
```

```
                con.Close();
                return myServiceData;
            }
            catch (SqlException sqlEx)
            {
                myServiceData.Result = true;
                myServiceData.ErrorMessage = "Connection can not open this " +
                   "time either connection string is wrong or Sever is down. Try la
ter";
                myServiceData.ErrorDetails = sqlEx.ToString();
                throw new FaultException<ServiceData>(myServiceData, sqlEx.ToString
());
            }
            catch (Exception ex)
            {
                myServiceData.Result = false;
                myServiceData.ErrorMessage = "unforeseen error occurred. Please try
 later.";
                myServiceData.ErrorDetails = ex.ToString();
                throw new FaultException<ServiceData>(myServiceData, ex.ToString())
;
            }
        }
    }
}
```

Example 3: Consuming FaultContract in the client.

```
class Program
{
    static void Main(string[] args)
    {
        try
        {
            Service1Client objServiceClient = new Service1Client();
            //Pass the connection string to the TestConnection Method.
            ServiceData objSeviceData = objServiceClient.TestConnection(
              @"integrated security=true;data source=localhost;initial catalog=mast
er");
            if (objSeviceData.Result == true)
                Console.WriteLine("Connection Succeeded");
            Console.ReadLine();
        }
        catch (FaultException<ServiceData> Fex)
        {
            Console.WriteLine("ErrorMessage::" + Fex.Detail.ErrorMessage + Environm
ent.NewLine);
            Console.WriteLine("ErrorDetails::" + Environment.NewLine + Fex.Detail.E
rrorDetails);
            Console.ReadLine();
        }
    }
}
```

D. Message Contract

A MessageContract controls the structure of a message body and serialization process. It is also used to send / access information in SOAP headers. By default, WCF takes care of creating SOAP messages depending on the service's *DataContracts* and *OperationContracts*. A MessageContract can be typed or untyped and are useful in iteroperability cases and when there is an existing message format we must comply with. Most of the time the developer concentrates

more on developing the DataContract, serializing the data and so on. Sometimes the developer will also require control of the SOAP message format. In that case WCF provides the MessageContract to customize the message depending on requirements. MessageContract is in *System.Net.Security*.

Example: Declaration of MessageContract.

```
1.  [MessageContract]
2.  public class EmployeeDetails
3.  {
4.      [MessageHeader(ProtectionLevel=ProtectionLevel.None)]
5.      public string EmpID;
6.      [MessageBodyMember(ProtectionLevel = ProtectionLevel.Sign )]
7.      public string Name;
8.      [MessageBodyMember(ProtectionLevel = ProtectionLevel.Sign )]
9.      public string Designation;
10.     [MessageBodyMember(ProtectionLevel=ProtectionLevel.EncryptAndSign)]
11.     public int Salary;
12.
13. }
```

Example: Use of MessageContract in service.

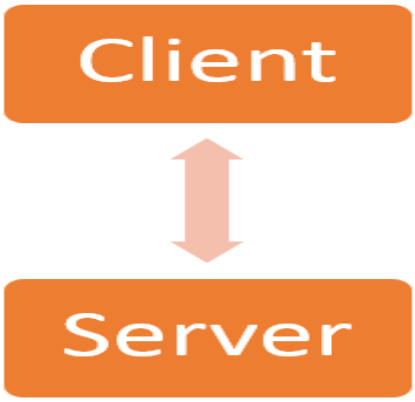
```
1.  [ServiceContract]
2.  interface ICuboidService
3.  {
4.      [OperationContract]
5.      [FaultContract(typeof(CuboidFaultException))]
6.      CuboidDetailResponse CalculateDetails1(CuboidInfoRequest cInfo);
7.
8.      [OperationContract]
9.      [FaultContract(typeof(CuboidFaultException))]
10.     CuboidDetail CalculateDetails2(CuboidInfo cInfo);
11.
12.     [OperationContract]
13.     [FaultContract(typeof(CuboidFaultException))]
14.     CuboidDetail CalculateDetails3(int nID, CuboidDimension cInfo);
15. }
```

Web security

WS Security is a standard that addresses security when data is exchanged as part of a Web service.This is a key feature in SOAP that makes it very popular for creating web services.

- Security is an important feature in any web application. Since almost all web applications are exposed to the internet, there is always a chance of a security threat to web applications. Hence, when developing web-based applications, it is always recommended to ensure that application is designed and developed with security in mind.

Process



In a standard HTTPS communication between the client and the server, the following steps take place

1. The client sends a request to the server via the client certificate. When the server sees the client certificate, it makes a note in its cache system so that it knows the response should only go back to this client.
2. The server then authenticates itself to the client by sending its certificate. This ensures that the client is communicating with the right server.
3. All communication thereafter between the client and server is encrypted. This ensures that if any other users try to break the security and get the required data, they would not be able to read it because it would be encrypted.

This is where SOAP comes in action to overcome such obstacles by having the WS Security specification in place. With this specification, all security related data is defined in the SOAP header element.

The header element can contain the below-mentioned information

1. If the message within the SOAP body has been signed with any security key, that key can be defined in the header element.
2. If any element within the SOAP Body is encrypted, the header would contain the necessary encryptions keys so that the message can be decrypted when it reaches the destination.

In a multiple server environments, the above technique of SOAP authentication helps in the following way.

- Since the SOAP body is encrypted, it will only be able to be decrypted by the web server that hosts the web service. This is because of how the SOAP protocol is designed.
- Suppose if the message is passed to the database server in an HTTP request, it cannot be decrypted because the database does not have right mechanisms to do so.
- Only when the request actually reaches the Web server as a SOAP protocol, it will be able to decipher the message and send the appropriate response back to the client.

Web Service Security Standards

- As discussed in the earlier section, the WS-Security standard revolves around having the security definition included in the SOAP Header.
- The credentials in the SOAP header is managed in 2 ways.
- First, it defines a special element called UsernameToken. This is used to pass the username and password to the web service.
- The other way is to use a Binary Token via the BinarySecurityToken. This is used in situations in which encryption techniques such as Kerberos or X.509 is used.
- The below diagram shows the flow of how the security model works in WS Security

Below are the steps which take place in the above workflow

1. A request can be sent from the Web service client to Security Token Service. This service can be an intermediate web service which is specifically built to supply usernames/passwords or certificates to the actual SOAP web service.
2. The security token is then passed to the Web service client.
3.
4. The Web service client then called the web service, but, this time, ensuring that the security token is embedded in the SOAP message.
5. The Web service then understands the SOAP message with the authentication token and can then contact the Security Token service to see if the security token is authentic or not.
6.
7. The below snippet shows the format of the authentication part which is part of the WSDL document. Now based on the below snippet, the SOAP message will contain 2 additional elements, one being the Username and the other being the Password.

```
<xs:element name="UsernameToken">
        <xs:complexType>
                <xs:sequence>
                        <xs:element ref="Username"/>
                        <xs:element ref="Password" minOccurs="0"/>
                </xs:sequence>
        <xs:attribute name="Id" type="xs:ID"/>
</xs:complexType></xs:element>
```

- When the SOAP Message is actually passed between the clients and the server, the part of the message which contains the user credentials could look like the one shown above. The wsse element name is a special element named defined for SOAP and means that it contains security based information.