**Object oriented programming:** OOP is a programming technique that think real world in terms of objects. Object maps the software model to real world concept.

In order to clearly understand the object orientation, let’s take your “hand” as an example. The “hand” is a class. Your body has two objects of type hand, named left hand and right hand. Their main functions are controlled/ managed by a set of electrical signals sent through your shoulders (through an interface). So the shoulder is an interface which your body uses to interact with your hands. The hand is a well architected class. The hand is being re-used to create the left hand and the right hand by slightly changing the properties of it.

**Structured** VS **Object Oriented:**

1. Process centered 1. Object centered

2. Reveals data 2. Hide data

3. Single unit 3. Modular unit

4. One time use 4. Reusable

**Objects:** Object is anything that is identifiable as a single material item and can perform a set of **related** activities. The set of activities that the object performs defines the object's behavior. For example, the hand can grip something or a Student (*object*) can give the name or address.

Objects are the components used to build OOP.

Object is a way to use classes.

**Class:** A class is simply a representation of a type of *object.*It is the blueprint/ plan/ template that describe the details of an *object.*

**Relationship between class and objects:** A class is a blueprint and an object is a building made from that blueprint.

**Class Members:** Each class can have different members:

* Fields: variables
* Properties: describe class data.
* Methods: define class behavior
* Events: provide communication between different classes and objects.
* Etc.

**Properties:**

Properties in C# provide the ability to protect a field by reading and writing to it through special methods called **accessors** (**get** and **set**). They are generally declared as public with the same data type as the field they are going to protect (**backing store**), followed by the name of the property and a code block that defines the get and set accessors.

* can validate data before allowing a change
* can transparently expose data on a class where that data is actually retrieved from some other source, such as a database
* allow to change the internal implementation of the property without breaking any programs that are using it

Properties, Interface etc. implement the encapsulation property of OOP.

class Time

{

private int seconds;

public int sec

{

get { return seconds; }

set { seconds = **value**; }

}

}

**Auto-implemented properties:** when no additional logic is required in the property accessors. Auto-implemented properties cannot be read-only or write-only.

public string Name { get; set; }

* A property without a set accessor is considered read-only.
* A property without a get accessor is considered write-only

**Methods:** A method is a code block that contains a series of statements. **It is also defined as a command for action**. In C#, every executed instruction is performed in the context of a method. The Main method is the entry point for every C# application and it is called by the common language runtime (CLR) when the program is started.

**Instantiating:** To create an object, you need to instantiate a class, or create a class instance. It is performed by **new** keyword.

* SampleClass sampleObject = new SampleClass();

#### Constructors: Constructors are special methods that implement the actions that are required to initialize an object (Class/Struct). They have the same name as the class or struct. You can create multiple constructor overloads in the same way as for any other method. If you do not provide a constructor for your object, C# will create one by default that instantiates the object.

public class SampleClass

{

public SampleClass()

{

// Add code here

}

}

#### Destructors: Destructors are used to destruct instances of classes. In the .NET Framework, the garbage collector automatically manages the allocation and release of memory for the managed objects in your application. However, you may still need destructors to clean up any unmanaged resources that your application creates

* Destructors cannot be defined in **structs**. They are only used with classes.
* A class can only have one destructor.
* Destructors cannot be inherited or overloaded.
* Destructors cannot be called. They are invoked automatically.
* A destructor does not take modifiers or have parameters.

class Car

{

~Car() // destructor

{

// cleanup statements...

}

}

**Garbage Collector, Finalize, Dispose:**

* Garbage collector to implicitly perform all the necessary memory management tasks (allocate and destroy) for managed code.
* The Finalize method is used **GC** to perform cleanup operations on unmanaged resources held by the current object before the current object is destroyed
* Developers *should* call Dispose explicitly to free unmanaged resources.
* Dispose doesn't remove the object itself from memory like Finalize; it can only free the object.

**Manage Code:** Code executing under the control of the CLR is called managed code. For example, any code written in C# or Visual Basic .NET is managed code.

**Unmanaged Code (File, Window, Connections):** Code that runs outside the CLR is referred to as "unmanaged code." COM components, ActiveX components, and Win32 API functions are examples of unmanaged code.

* COM: Component Object Model technology is to create re-usable software components, link components together to build applications.
* ActiveX components: **ActiveX** is a framework used to develop COM
* Examples: Audio, Video, Excel, Calendar etc.
* Examples of unmanaged code include the Microsoft Win32 API, legacy DLLs and EXEs created for Windows applications prior to the Microsoft .NET Framework and COM objects.

## Abstract class: An abstract class cannot be instantiated. It can only be used as a super-class for other classes that extend the abstract class. The purpose of an abstract class is to provide a common definition of a base class that multiple derived classes can share. An abstract class is a class that cannot be instantiated, but must be inherited from. An abstract class may be fully implemented, but is more usually partially implemented or not implemented at all, thereby encapsulating common functionality for inherited classes.

**Sealed Class:** A sealed class cannot be inherited. It is an error to use a sealed class as a base class. Use the **sealed** modifier in a class declaration to prevent inheritance of the class.

sealed class MyClass

{

    public int Add(int x, int y)

    {

        return x + y;

    }

}

**Partial Class:** Split a class in more than one file. It is done by using keyword **partial**.

public partial class Employee

{

public void DoWork()

{

}

}

public partial class Employee

{

public void GoToLunch()

{

}

}

**Advantages:**

* More than one developer can simultaneously write the code for the class.
* You can easily write your code (for extended functionality) for a VS.NET generated class. This will allow you to write the code of your own need without messing with the system generated code.

**Interface:** An interface contains only the signatures of methods, delegates, Properties or events. The implementation of the methods is done in the class that implements the interface, as shown in the following example:

* Since it is not possible to derived from more than one class, so when we need from two or more classes then we use Interface.

interface ISampleInterface

{

void SampleMethod();

}

class ImplementationClass : ISampleInterface

{

// Explicit interface member implementation:

void ISampleInterface.SampleMethod()

{

// Method implementation.

}

}

**Difference between Interface and Abstract class:**

1. Interface contains only the signatures of methods, there is no implementation. Abstract class can have complete or incomplete methods and can be extended.
2. Interface can inherit more than one interface. Abstract class can implement more than one interfaces, but can inherit only one class

## When to use an Abstract Class and an Interface:

1. Use Interface when no need a massive hierarchical inheritance but need to inherit from more than one class.

If you want to provide common, implemented functionality among all implementations of your component, use an abstract class. Abstract classes allow you to partially implement your class, whereas interfaces contain no implementation for any members.

1. Abstract classes should be used primarily for objects that are closely related, whereas interfaces are best suited for providing common functionality to unrelated classes

You have an Animal class. It is an abstract class, because you cannot instantiate a generic "Animal," but it provides base functionality.  
You have many devired classes of Animal.

Animals are capable of doing different things like flying, digging and walking, but there are some common operations as well like eating and sleeping. Some common operations are performed by all animals, but in a different way as well. When an operation is performed in a different way, it is a good candidate for an abstract method (forcing subclasses to provide a custom implementation).

Now you want to make something fly. What can fly? Birds and Houseflys (among others), so these classes should provide similar functionality, even though they are widely spaced on our inheritance tree. The solution? Make them use interfaces

**Major principles/properties of OOP:**

■ **Encapsulation:** is the enclosing of both properties and methods together in a common structure. Encapsulation is mainly achieved by creating classes; the classes expose public methods and properties.

Grouping related information and processes into self-contained unit, it makes object (class) easy to change without hurting the overall functioning of the system. By hiding internal implementation it makes object (class) easy to use. Suppose when a class (claas1) is used inside another class creating its object then it is not needed to know how class1 is internally implemented.

■ **Inheritance:** Ability of a new class to be created, from an existing class by extending it.

public class Exception public class IOException : Exception

{ {

} }

According to the above example the new class (*IOException*), which is called the derived class or subclass, inherits the members of an existing class (*Exception*), which is called the base class or super-class. The class *IOException* can extend the functionality of the class Exception by adding new types and methods and by overriding existing ones.

- Primary benefit: Code reuse

■ **Polymorphism:** Polymorphism means 'many shapes'. More precisely *Polymorphism means* the ability to request that the same operations be performed by a wide range of different types of things.

Polymorphism means *one name, many forms*. Polymorphism manifests itself by having multiple methods all with the same name, but slightly different functionality

In *OOP* the *polymorphisms* is achieved by using: **Overloading and Overriding**

■ **Method Overloading:** Same name with different parameter.

■ **Method Overriding:** Creating a method in derived class with same signature (same name, same return type, and same arguments) as a method in base class is called as method overriding. Method overriding is possible only in derived classes, but not within the same class.

When derived class needs a method with same signature as in base class, but wants to execute different code than provided by base class then method overriding will be used. To allow the derived class to override a method of the base class, C# provides two options, **virtual** methods and **abstract** methods.

■ **Abstract Classes and Abstract Methods:** There may be a situation where it is not possible to define a method in base class and every derived class must override that method. In this situation abstract methods are used. When a method is declared as abstract in the base class then it is not possible to define it in the base class (so it is an empty method in base class) and every derived class of that class must provide its own definition for that method. When a class contains at least one abstract method, then the class must be declared as **abstract class**.

It is **mandatory** to override abstract method in the derived class. When a class is declared as abstract class, then it is not possible to create an instance for that class

//Abstract class

abstract class Shape1

{

protected float R, L, B;

//Abstract methods can have only declarations

public abstract float Area();

public abstract float Circumference();

}

class Rectangle1 : Shape1

{

public void GetLB()

{

Console.Write("Enter Length : ");

L = float.Parse(Console.ReadLine());

Console.Write("Enter Breadth : ");

B = float.Parse(Console.ReadLine());

}

public override float Area()

{

return L \* B;

}

public override float Circumference()

{

return 2 \* (L + B);

}

}

■ **Virtual Methods**: When you want to allow a derived class to override a method of the base class, within the base class method must be created as virtual method and within the derived class method must be created using the keyword override.

When a method declared as virtual in base class, then that method can be defined in base class and it is **optional** for the derived class to override that method. When it needs same definition as base class, then no need to override the method and if it needs different definition than provided by base class then it must override the method.

class Shape

{

protected float R, L, B;

public virtual float Area()

{

return 3.14F \* R \* R;

}

public virtual float Circumference()

{

return 2 \* 3.14F \* R;

}

}

class Rectangle : Shape

{

public void GetLB()

{

Console.Write("Enter Length : ");

L = float.Parse(Console.ReadLine());

Console.Write("Enter Breadth : ");

B = float.Parse(Console.ReadLine());

}

public override float Area()

{

return L \* B;

}

public override float Circumference()

{

return 2 \* (L + B);

}

}

**Generic:** Generics allow to define type-safe data structures, without committing to actual data types. By using a generic type parameter T you can write a single class that other client code can use without incurring the cost or risk of runtime casts or boxing operations, as shown here:

// Declare the generic class

public class GenericList<T>

{

void Add(T input) { }

}

class TestGenericList

{

private class ExampleClass { }

static void Main()

{

// Declare a list of type int

GenericList<int> list1 = new GenericList<int>();

// Declare a list of type string

GenericList<string> list2 = new GenericList<string>();

// Declare a list of type ExampleClass

GenericList<ExampleClass> list3 = new GenericList<ExampleClass>();

}

}

**Exm: List, Queue, Dictionary**

## Inheritance and Generics: When deriving from a generic base class, you must provide a type argument instead of the base-class's generic type parameter:

public class BaseClass<T>

{...}

public class SubClass : BaseClass**<int>**

{...}

**Collections:** specialized classes for data storage and retrieval.

* Support for **stacks**, **queues**, **lists**, and **hash tables**.

**Serialization:** Serialization is the process of **converting an object into a** **stream of bytes (binary)** in order to persist it to memory, a database, or a file. Its main purpose is to save the state (Session) of an object in order to be able to recreate it when needed. **The reverse process is called deserialization**.

Format of Serialization:

BinaryFormatter:

* + - **Binary Serialization:**
      * Used when in same platform (like under.net)
      * Use [Serializable] attribute before class to implement.

SoapFormatter:

* + - **XML Serialization:** XML is a text-based file standard, so an object that is serialized by using XML can be easily processed by an application written for different operating system in different environment.
      * Can serialize only public data
      * Use [XmlRoot(“class name”)] attribute before class to implement.

**When we write code** under .NET, we need not to care about serialization. .NET framework itself takes care of it and it uses binary serialization. But if we want to store object in a file or other places (like web service), or if we use other session mode except **InProc**, then we need to care about it.

**Metadata:** Metadata is information about the data. For example, an image may include metadata that describes how large the picture is, the color depth, the image resolution, when the image was created, and other data.

**Attributes:** Attributes are a mechanism for adding metadata, such as compiler instructions and other data about your data, methods, and classes, to the program itself. Attributes are inserted into the metadata and are visible through metadata-reading tools.

* An **attribute** is an object that represents data you want to associate with an element in your program.
* **Attributes exist in two forms:** attributes that are defined in the Common Language Runtime's base class library and custom attributes that you can create, to add extra information to your code. This information can later be retrieved programmatically.

**Reflection:** Reflection is the process by which a program can read its own metadata. **In other words,** reflection provides objects that encapsulate assemblies, modules and types. Reflection can be used to:

* Dynamically create an instance of a type
* Bind the type to an existing object
* Get the type from an existing object and invoke its methods or access its fields and properties.

Ex: // Using GetType to obtain type information:

int i = 42;

System.Type type = i.GetType();

System.Console.WriteLine(type);

Output is: System.Int32

// Using Reflection to get information from an Assembly:

System.Reflection.Assembly o = System.Reflection.Assembly.Load("mscorlib.dll");

System.Console.WriteLine(o.GetName());

The output is:

mscorlib, Version=2.0.0.0, Culture=neutral, PublicKeyToken=b77a5c561934e089

**Membership:** Membership is a class that Validates user credentials and manages user settings. This class cannot be **inherited (Sealed class)**.

* It uses **aspnet\_regsql** to manage user.

**Access Modifier**

**Public:** The type or member can be accessed by any other code in the same assembly or another assembly that references it.

**Private:** The type or member can be accessed only by code in the same class or struct.

**Protected:** The type or member can be accessed only by code in the same class or struct, or in a class that is derived from that class.

**Internal:** The type or member can be accessed by any code in the same assembly, but not from another assembly.

**Protected internal:** The type or member can be accessed by any code in the assembly in which it is declared, or from within a derived class in another assembly.

**Software Development Life Cycle (SDLC):** A **software development process**, also known as a **software development life cycle (SDLC)**, is a structure imposed on the development of a software product

**Software Development Methodology (SDM):** A **software development methodology** or **system development methodology** in software engineering is a framework that is used to structure, plan, and control the process of developing an information system.

* 1 Overview
* [2 Software development activities](http://en.wikipedia.org/wiki/Software_development_process#Software_development_activities)
  + [2.1 Planning](http://en.wikipedia.org/wiki/Software_development_process#Planning)
  + [2.2 Implementation, testing and documenting](http://en.wikipedia.org/wiki/Software_development_process#Implementation.2C_testing_and_documenting)
  + [2.3 Deployment and maintenance](http://en.wikipedia.org/wiki/Software_development_process#Deployment_and_maintenance)
* [3 Software development **models**](http://en.wikipedia.org/wiki/Software_development_process#Software_development_models) **/ methodologies** 
  + [3.1 Waterfall model](http://en.wikipedia.org/wiki/Software_development_process#Waterfall_model)
  + [3.2 Spiral model](http://en.wikipedia.org/wiki/Software_development_process#Spiral_model)
  + [3.3 Iterative and incremental development](http://en.wikipedia.org/wiki/Software_development_process#Iterative_and_incremental_development)
  + [3.4 Agile development](http://en.wikipedia.org/wiki/Software_development_process#Agile_development)
  + [3.5 Code and fix](http://en.wikipedia.org/wiki/Software_development_process#Code_and_fix)
* [4 Process improvement models](http://en.wikipedia.org/wiki/Software_development_process#Process_improvement_models)
* [5 Formal methods](http://en.wikipedia.org/wiki/Software_development_process#Formal_methods)
* [6 See also](http://en.wikipedia.org/wiki/Software_development_process#See_also)
  + [6.1 Development methods](http://en.wikipedia.org/wiki/Software_development_process#Development_methods)
  + [6.2 Related subjects](http://en.wikipedia.org/wiki/Software_development_process#Related_subjects)
* [7 Bibliography](http://en.wikipedia.org/wiki/Software_development_process#Bibliography)
* [8 References](http://en.wikipedia.org/wiki/Software_development_process#References)
* [9 External links](http://en.wikipedia.org/wiki/Software_development_process#External_links)

**SOA:** In [software engineering](http://en.wikipedia.org/wiki/Software_engineering), a **Service-Oriented Architecture** (**SOA**) is a set of principles and methodologies for designing and developing software in the form of **interoperable services**.

Concept of SOA is to integrate widely disparate Web-based applications which may be even implementation in different platforms.

**Principles:** Reusability, Modularity, Interoperability.

Web services, WCF etc. use SOA concept.

**SOAP:** SOAP is a simple XML-based communication protocol to let applications exchange information over HTTP.

* Is a format for sending messages
* Platform independent
* Language independent

**Web service:** Web services are components on a Web server that a client application can use by making HTTP requests across the Web. Clients of a Web Services need not to know anything about the platform, object model, or programming language used to implement the service; they only need to understand how to send and receive messages using the web service.

* Access using URL

<http://www.simple-talk.com/dotnet/asp.net/using-webservices-with-asp.net/>

**Two uses:**

1. Reusable Application-Components: There are some applications needs very often, like- Weather forecasting, language conversion, etc. So why make these over and over again?
2. Interoperability. (Make usable to different platform)

**Web service Platform elements:**

* SOAP (Rest-Representational state server.)
* UDDI (Universal Description, Discovery and Integration): Is a directory for storing information about web service. Here companies can register and search for web services.
* WSDL (Web Service description Language): XML Based Language for locating and describing web services.

**Serialization of Web service:** By default web services are XML-serialized. But if the web service is used in same application of same platform (like my auto complete), then you can make it binary serialized using [Serializable] before the method.

In ASP.NET: System.Web.Services(.webservice)

[WebMethod]

Public … method()

{

}

Extension: .asmx

**WebMethod:** is an Integration Platform that supports the integration of diverse services, such as mapping data between formats and communication between systems.

* Java based, multiplatform supported
* You can call it as **Rest, SOAP**
* Named by an independent company

**Windows Service:** Microsoft Windows services, formerly known as NT (New technology) services, enable you to create long-running executable applications (that performs specific function) that run in their own Windows sessions (User specific session, window’s own session for its own application). These services can be automatically started when the computer boots, can be paused and restarted, and do not show any user interface. These features make services ideal for use on a server or whenever you need long-running functionality that does not interfere with other users who are working on the same computer.

Windows provides an interface called the **Service Control Manager** that manages the starting and stopping of services.

**When Use:**

* When an application needs to long time (continuously) run.
* To run code in the background without user interaction.
* Like: monitor something,….

**Why Use:**

* You can run it on a different server if necessary
* You can start and stop it independently of the web site

It could well have some impact on the performance of the web site itself - something you want to avoid.

The buzz-word here is "separation of concerns". The web site is concerned with presenting the data to the user, the service with checking the integrity of the data.

You can also update the web site and service independently of each other should you need to.

**Threading:** Threading is a technique which enables multiple tasks to be executed at the same time.

* Used for asynchronously service request.
* One of the benefits of using multiple threads in an application is that each thread executes asynchronously

**Thread Synchronization:** To make thread synchronization (so that other thread can’t interrupt when one is working) **lock** can be used.

public class TestThreading

{

private Object lockThis = new Object();

public void Function()

{

lock (lockThis)

{

// Access thread-sensitive resources.

}

}

}

UML

<http://stackoverflow.com/questions/6065056/does-asp-net-mvc-really-eliminates-viewstate>

<http://stackoverflow.com/questions/7674102/advantages-and-disadvantages-of-asp-net-webforms-vs-asp-net-mvc-some-points>

<http://stackoverflow.com/questions/4561719/asp-mvc-no-view-state-or-post-back>

http://msdn.microsoft.com/en-us/library/ms972976.aspx

**Software Design Pattern:** In software engineering, **a design pattern is a general reusable solution to a commonly occurring problem** within a given context in software design. A design pattern is not a finished design that can be transformed directly into code. **It is a description or template for how to solve a problem that can be used in many different situations. Object-oriented design patterns typically show relationships and interactions between classes or objects, without specifying the final application classes or objects that are involved.** Many patterns imply object-orientation or more generally mutable state, and so may not be as applicable in functional programming languages, in which data is immutable or treated as such.

**Stateless protocol:**

In computing, a **stateless protocol** is a [communications protocol](http://en.wikipedia.org/wiki/Communications_protocol) that treats each request as an independent transaction that is unrelated to any previous request so that the communication consists of independent pairs of requests and responses. A stateless protocol does not require the [server](http://en.wikipedia.org/wiki/Server_%28computing%29) to retain [session](http://en.wikipedia.org/wiki/Session_%28computer_science%29) information or status about each communications partner for the duration of multiple requests.

An example of a stateless protocol is the [Hypertext Transfer Protocol](http://en.wikipedia.org/wiki/Hypertext_Transfer_Protocol) (HTTP) which is the foundation of data communication for the [World Wide Web](http://en.wikipedia.org/wiki/World_Wide_Web).

**Advantages and Disadvantages:** The stateless design simplifies the server design because there is no need to dynamically allocate storage to deal with conversations in progress. If a client dies in mid-transaction, no part of the system needs to be responsible for cleaning the present state of the server.

**A disadvantage of statelessness** is that it may be necessary to include additional information in every request, and this extra information will need to be interpreted by the server.

**Markup Language:** A **markup language** is a modern system for annotating (a note that is made while reading any form of text) a document in a way that is syntactically distinguishable from the text.

**View State:** is a technique used by an ASP.NET Web page to persist (Continue to exist) changes to the state of a Web Form across post backs. The view state of a page is, **by default, placed in a hidden form field named \_\_VIEWSTATE.** This hidden form field can easily get very large, on the order of tens of kilobytes. Not only does the \_\_VIEWSTATE form field cause slower downloads, but, whenever the user posts back the Web page, the contents of this hidden form field must be posted back in the HTTP request, thereby lengthening the request time, as well.

**ASP.NET Web pages are made up of both an HTML portion and a code portion, with the HTML portion containing HTML markup and Web control syntax.** The ASP.NET engine converts the HTML portion from its free-form text representation into a series of programmatically-created Web controls.

When an ASP.NET Web page is visited for the first time after a change has been made to the HTML markup or Web control syntax in the .aspx page, the ASP.NET engine auto-generates a class. If you created your ASP.NET Web page using the code-behind technique, this auto generated class is derived from the page's associated code-behind class. This auto generated class, along with a compiled instance of the class, is stored in the **WINDOWS**\Microsoft.NET\Framework\**version**\Temporary ASP.NET Files folder, in part so that it doesn't need to be recreated for each page request.

**The purpose** of this auto generated class is to programmatically create the page's control hierarchy.

**What state doesn't need to be persisted across post backs?** Since **declarative** properties (text property, Font-Name property etc.) a control are automatically reassigned on each post back when the control hierarchy is constructed, there's no need to store these property values in the view state.

**What need to be stored in the view state are any programmatic changes to the page's state.** For example, suppose that in addition to a Label Web control, the page also contained two Button Web controls, a Change Message Button and an Empty Postback button. The Change Message Button has a Click event handler that assigns the Label's Text property to "Goodbye, Everyone!” the Empty Postback Button just causes a postback, but doesn't execute any code. The change to the Label's Text property in the Change Message Button would need to be saved in the view state.

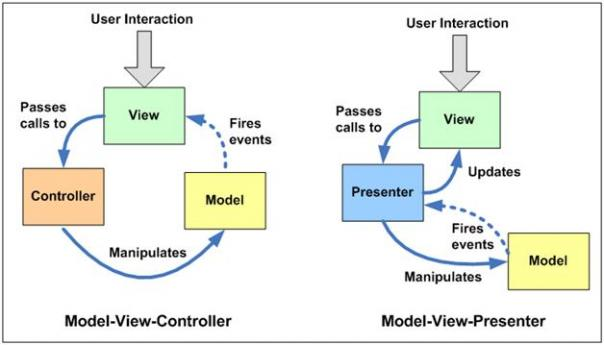
**Where ViewState store value and in which format:** In hidden field, encrypted format.

**MVC Vs Web Forms:**

**Problems with ASP.net Web Form:** Webforms introduced **event-driven approach and also introduced Viewstate and Postback. The end result is that web forms break the stateless nature of the Web. Both Viewstate and Postbacks have been made lot of problems and increased complexity of the web application development.** Many web pages having hundreds of KB size of Viewstate that affected the performance of the applications sometime. Developers do not have the control of the rendering HTML of web forms and Server controls that render html with mixed inline style and deprecated tags that does not follows standards. Another problem with Web Forms is the integration of JavaScript frameworks due to the naming conventions of rendered HTML. **The page life cycle of the Web Form is too complex and has the tightly coupling between all things in the ASP.net framework and a single class is used to display both output and handles user input. So unit testing is almost an impossible task.** Today unit testing is very important in modern software development especially when we following agile methodologies and practices. **Since web is a stateless thing, Events, Postbacks and Viewstate are not a good way.** Today **many ASP.net web form developers are facing different type of browser compatibility issues when developing public face internet applications**

**The ASP.net MVC way:** The ASP.NET MVC simplifies the complex parts of ASP.net Web Forms without any compromise of the power and flexibility of ASP.NET platform. ASP.net MVC implements Model-View-Controller UI pattern for web application development that lets you allows to develop applications in a loosely couples manner. MVC pattern is separating the application in three parts- Model, View and Controller. A view is responsible for rendering the user interface (UI) of the application and it is nothing more than html templates that filled with application’s data passed by the controller. The Model implements the logic for the application's data and it represents the business objects of the application that using the View for rendering user interface. Controllers are handles and responds to user input and interaction. The web request will be handled by the controller, and the controller will decide which model objects to use and which view objects to render. **The MVC model replaces the Web Form events with the controller actions.** The **main advantages of the MVC models are clear separation of concerns, unit testing facility, and more control over the URLs and HTML.** The **MVC model does not use Viewstate, Postbacks, Server controls, and server-based forms that enable full control over the application and html rendered by the Views.** MVC model is using Representational state transfer (REST) based URLs instead of file-name extensions used by the Web Form model so that we can make search engine optimization (SEO) URLs published by the application.

**MVC vs MVP:**



How Many Kinds of Session

**JavaScript:** is a scripting language that works in all major browsers, such as IE, Firefox, Chrome, Opera, and Safari. We use JavaScript along with ASP.NET for client-side programming.

* Validation
* Warning
* Tool Tip
* Online Watch
* Select unselect all check box

**AJAX (**Asynchronous JavaScript and XML): is the art of exchanging data with a server, and updating parts of a web page - without reloading the whole page.

* Asynchronous post back

**Ajax Extensions**: Attached with ASP.NET installation.

* + - Update Panel
    - Scripting Manager
    - Timer

**Ajax Control Toolkit:** Need to install Later

* + - Calendar Extender
    - Tab Container
    - Color Picker Extnder
    - Password Strength
    - Text Box Water Mark Extender

**Parse Vs Convert:** Convert internally implement parse, additionally it checks null. That is, if Null is parsed to int, it will through exception, but convert will return 0.

**Authentication in ASP.NET:**

1. Windows Authentication : Application will use windows users
2. Forms Authentication : application-specific login form and perform authentication using your own code

**MSIL:** Microsoft Intermediate Language is a CPU-independent set of instructions for controlling applications. When asp.net code is compiled, it is translated into MSIL. Before code can be run, MSIL must be converted to CPU-specific code

**CLR:** Common Language Runtime is runtime execution environment for ASP.NET. It provide following services:

* Language Integration: ability to easily use components developed in others language.
* Provide Language features such as inheritance, interfaces, and overloading for object-oriented programming.
* Support for exception handling.
* Memory Management (Garbage Collection).

**GET Vs POST method in HTML:** both are used to send data. <form method=”post”></form>

**Get:** Data is shown in url. So

* Not secure
* Can be cached
* Can remain in browser history.
* Can be booked Mark
* Can be Hacked
* Good in AJAX environment.

**Post:** send data in encoded format.

* Secure
* Used for long data

**Can a solution/project contain more than one config file?**

Yes. In every folder there may have one (or more than one with different name) config file and the benefit is the class/forms in same folder will get necessary thing quickly. Actually class/ webforms first look in its own folder for webconfig. If not get then go its parent folder and so on.

Single Ton:

**Array:**

int[] intArray;

intArray = new int[5];

char[] charArray = new char[5];

bool[] boolArray = new bool[2];

string[] stringArray = new string[10];

// Initialize a fixed array one item at a time

int[] staticIntArray = new int[3];

staticIntArray[0] = 1;

staticIntArray[1] = 3;

staticIntArray[2] = 5;

// Read array items one by one

Console.WriteLine(staticIntArray[0]);

Console.WriteLine(staticIntArray[1]);

Console.WriteLine(staticIntArray[2]);

// Initialize a dynamic array items during declaration

string[] strArray = new string[] { "Mahesh Chand", "Mike Gold", "Raj Beniwal", "Praveen Kumar", "Dinesh Beniwal" };

// Read array items using foreach loop

foreach (string str in strArray)

{

Console.WriteLine(str);

}

int[] staticIntArray = new int[3] {1, 3, 5};

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

int[] source = new int[5];

source[0] = 1;

source[1] = 2;

source[2] = 3;

source[3] = 4;

source[4] = 5;

*//*

*// Instantiate and allocate the target array.*

*//*

int[] target = new int[5];

*//*

*// Copy the source to the target.*

*//*

Array.Copy(source, target, 5);

ArrayList

**MY CV**

**User Control Extension**: .ascx

**Tag Prefix:** for .NET control .asp, for other you can define tag, for example when we register Ajax Control toolkit, we generally set TagPrefix="cc1"

**For Asynchronous Post back:** Update panel of Ajax

Auto Complete: **AutoCompleteExtender of Ajax**

TargetControlID: TextBox

ServicePath: Path pf the webervice

ServiceMethod: Method Under webmethod

Information is collected from database using select statement with wildcard character(%, \_ [only a single character can put in place of \_. LIKE '\_la' Ex. Ola, ])

**SiteMap:**

* In webconfig: <siteMap>

<providers defaultProvider="DefaultSiteMap" >

……

……. All sitemap files

<providers>

<siteMap>

* Files are stored in folder under App\_Data folder
* In UI of Master page SiteMapDataSource
* Change SiteMapProvider of SiteMapDataSource frm code according to role.

**Multilanguage:**

* Store resource file under App\_GlobalResources folder
* Resource file contain: Name, value, comment.
* Name the default file as epaper.resx
* Name other file as epaper.English.resx
* Text='<%$ Resources:ePaper,EPDFs\_ImgDelete %>'