**CLASS, OBJECT, INSTANCE**

A **class** is a blueprint or template for creating objects, providing initial values for state (member variables) and implementations of behavior (member functions, methods).

An object is an instance of a class. When a class is declared, no memory is allocated so class is just a template. When the object of the class is declared, memory is allocated. Instance is Logical but object is Physical means occupies some memory.

* Instance is Logical but object is Physical means occupies some memory.
* We can create an instance for abstract class as well as for interface, but we cannot create an object for those.
* Object is instance of class and instance means representative of class i.e object.
* Instance refers to Reference of an object.
* Object is actually pointing to memory address of that instance.
* We can’t pass instance over the layers but we can pass the object over the layers.
* We can’t store an instance but we can store an object.
* A single object can have more than one instance.
* Instance will have the both class definition and the object definition where as in object it will have only the object definition.

**ABSTRACT CLASS**

 An abstract class is a class with at least one method defined as abstract. Abstract class cannot be instantiated. An abstract class can have one or more abstract methods and properties and other methods and properties like normal classes.

**CONSTRUCTOR**

Constructors are special methods, used when instantiating a class and makes sure that all objects are initialized before their use. A constructor can never return anything, which is why you don't have to define any return type for it.

class C

{

       private int x;

       private int y;

       public C (int i, int j)

       {

                 x = i;

                 y = j;

       }

       public void display ()

       {

               Console.WriteLine(x + "i+" + y);

       }

}

Basically constructors are 5 types those are

      1.    Default Constructor

      2.    Parameterized Constructor

      3.    Copy Constructor

      4.    Static Constructor

      5.    Private Constructor

**DESTRUCTORS**

Destructors are used to destruct instances of classes. A destructor, a method called once an object is disposed, can be used to cleanup resources used by the object. Destructor doesn’t look very much like other methods.

class Car

{

~Car()

{

Console.WriteLine("Out..");

}

}

* Destructors cannot be inherited or overloaded.
* Destructors cannot be called. They are invoked automatically.
* A destructor does not take modifiers or have parameters.

**Class:**

It is a collection of objects.

**Object:**

Object is an instance of a class. It is a real time entity.

**Abstraction:**

Abstraction is a process of hiding the implementation details and displaying the essential features.

**Encapsulation**:

Encapsulation is a process of binding the data members and member functions into a single unit.

Wrapping up of member data and member functions of a class in a single unit is called encapsulation. The visibility of the member functions, data members is set via access modifiers used in class.

**Public** 🡪 Accessible outside the class through object reference.

**Private** 🡪 Accessible inside the class only through member functions.

**Protected** 🡪 Just like private but Accessible in derived classes also through member functions.

**Internal** 🡪 Visible inside the assembly. Accessible through objects.

**Protected Internal** 🡪 Visible inside the assembly through objects and in derived classes outside the assembly through member functions.

**POLYMORPHISM**

‘Poly’ means many and ‘morphism’ means changing or form. The concepts introduces in the form of many behaviours of an object.

Polymorphism is the ability of an object or reference to take many different forms at different instances.

Polymorphism is two types:

1. Compile time polymorphism/Overloading/ Static
2. Runtime polymorphism/Overriding/ Dynamic

**Method Overriding:**

class BC

{

public virtual void Display()

{

System.Console.WriteLine("BC::Display");

}

}

class DC : BC

{

public override void Display()

{

System.Console.WriteLine("DC::Display");

}

}

class Demo

{

public static void Main()

{

DC b = new DC(); *// Compile-time call DC.*

b.Display(); *// Runtime execute DC.*

BC b = new BC(); *// Compile-time call BC.*

b.Display(); *// Runtime execute BC.*

BC b = new DC(); *// Compile-time call BC.*

b.Display(); *// Runtime execute DC.*

}

}

Output:

DC::Display

BC::Display

DC::Display

**Method Overloading:**

public class TestOverloading

{

public void Add(string a1, string a2)

{

Console.WriteLine("Adding Two String :" + a1 + a2);

}

public void Add(int a1, int a2)

{

Console.WriteLine("Adding Two Integer :" + a1 + a2);

}

}

static void Main(string[] args)

{

TestOverloading obj = new TestOverloading();

obj.Add("Manish " , "Agrahari");

obj.Add(5, 10);

Console.ReadLine();

}

**STATIC CLASS AND METHOD**

A Static class is basically the same as a non-static class, but there is one difference: a static class cannot be instantiated. In other words, we cannot use the [new](http://msdn.microsoft.com/en-us/library/51y09td4%28v=vs.90%29.aspx) keyword to create object of the class. Because there is no instance variable, we access the members of a static class by using the class name itself. For example, if we have a static class that is named “Program” that has a public method named “MethodA”, we call the method as: Program.MethodA();

Whenever we write a function or declare a variable, it doesn’t create instance in a memory until we create object of class. But if we declare any function or variable with static modifier, it directly create instance in a memory and acts globally. The static modifier doesn’t reference with any object.

1. class SomeClass {

public int InstanceMethod()

{

return 30;

}

public static int StaticMethod()

{

return 40;

}

}

static void Main()

{

SomeClass instance = new SomeClass();

instance.InstanceMethod();

SomeClass.StaticMethod();

}

1. static class myStaticClass

{

public static void someFunction()

{

return 95;

}

}

static void Main()

{

myStaticClass.someFunction();

}

**PARTIAL CLASS**

A class defined in two or more files is called a partial class. The keyword partial is used to define the class. When working on large projects, spreading a class over separate files allows multiple programmers to work on it simultaneously. During compile time all the partial class is compiled into one type only.

partial class A

{

public static void A1()

{

Console.WriteLine("A1");

}

}

partial class A

{

public static void A2()

{

Console.WriteLine("A2");

}

}

class Program

{

static void Main()

{

A.A1();

A.A2();

}

}

However, namespaces can hold other types as follows:

* Classes
* Structures
* Interfaces
* Enumerations
* Delegates

**INHERITANCE**

Module Module1

Public Class s1

Public a As Integer = 5

Public Function val() As Integer

Return a

End Function

End Class

Public Class s2

Inherits s1

Public c As Integer = 20

Public Function add() As Integer

Return c + a

End Function

End Class

Sub Main()

Dim res As New s2

System.Console.WriteLine("Final Value is::")

System.Console.WriteLine(res.add())

Console.Read()

End Sub

End Module

**Result:**

Final Value is: 25

**Description:**

In the above example the value of **a** is inherited to from base class **s1** to the derived class **s2**. Using the instance **res** of class **s2** the values of 'a' as well as 'c' is added to give the result.

**INTERFACE**

An interface is not a class. It is an entity that is defined by the word Interface. An interface has no implementation; it only has the signature or in other words, just the definition of the methods without the body.

An *interface* looks like a class, but has no implementation. The only thing it contains is declarations of *events*, *indexers*, *methods* and/or *properties*. The reason *interfaces* only provide declarations is because they are inherited by *classes* and *structs*, which must provide an implementation for each interface member declared.

**Example:**

Module Module1

Public Interface Interface1

Function Add(ByVal x As Integer) As Integer

End Interface

Public Class first

Implements Interface1

Public Function Add(ByVal x As Integer)

As Integer Implements Interface1.Add

Console.WriteLine("Implementing x+x

in first class::" & (x + x))

End Function

End Class

Public Class second

Implements Interface1

Public Function Add(ByVal x As Integer)

As Integer Implements Interface1.Add

Console.WriteLine("Implementing x+x+x

in second class::" & (x + x + x))

End Function

End Class

Sub Main()

Dim obj1 As New first

Dim obj2 As New second

obj1.Add(10)

obj2.Add(50)

Console.Read()

End Sub

End Module

**Result:**

Implementing x+x in first class:: 20

Implementing x+x+x in second class:: 150

**JOIN**

## At the top level there are mainly 3 types of joins:

1. **INNER JOIN** - The INNER JOIN keyword selects all rows from both tables as long as there is a match between the columns in both tables.
2. **OUTER JOIN** -are of **3** types:
   1. LEFT OUTER JOIN - The LEFT OUTER JOIN keyword returns all rows from the left table (table1), with the matching rows in the right table (table2). The result is NULL in the right side when there is no match.
   2. RIGHT OUTER JOIN - The RIGHT OUTER JOIN keyword returns all rows from the right table (table2), with the matching rows in the left table (table1). The result is NULL in the left side when there is no match.
   3. FULL OUTER JOIN - The FULL OUTER JOIN keyword returns all rows from the left table (table1) and from the right table (table2).
3. **CROSS JOIN**- Cross join is a cartesian join means cartesian product of both the tables. This join does not need any condition to join two tables. (select \* from table\_1 cross join table\_2)
4. **SELF JOIN-** when the table has a foreign key that references its own Primary Key.

* If we just mention JOIN then by default it is a INNER JOIN.
* An OUTER join has to be LEFT | RIGHT | FULL we cannot simply say OUTER JOIN.
* We can drop OUTER keyword and just say LEFT JOIN or RIGHT JOIN or FULL JOIN.

## C:\Users\BAPPA\Desktop\Visual_SQL_JOINS_orig.jpg

## Various Collection Classes and Their Usage

|  |  |
| --- | --- |
| [ArrayList](http://www.tutorialspoint.com/csharp/csharp_arraylist.htm) | It represents ordered collection of an object that can be **indexed** individually.  It is basically an alternative to an array. However, unlike array we 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. |
| [Hashtable](http://www.tutorialspoint.com/csharp/csharp_hashtable.htm) | It uses a **key** to access the elements in the collection.  A hash table is used when we need to access elements by using key, and we 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. |
| [SortedList](http://www.tutorialspoint.com/csharp/csharp_sortedlist.htm) | 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 we access items using an index, it is an ArrayList, and if we access items using a key , it is a Hashtable. The collection of items is always sorted by the key value. |
| [Stack](http://www.tutorialspoint.com/csharp/csharp_stack.htm) | It represents a **last-in, first out** collection of object. (LIFO)  It is used when we need a last-in, first-out access of items. When we add an item in the list, it is called **pushing** the item and when we remove it, it is called **popping** the item. |
| [Queue](http://www.tutorialspoint.com/csharp/csharp_queue.htm) | It represents a **first-in, first out** collection of object. (FIFO)  It is used when we need a first-in, first-out access of items. When we add an item in the list, it is called **enqueue** and when we remove an item, it is called **dequeue**. |
| [BitArray](http://www.tutorialspoint.com/csharp/csharp_bitarray.htm) | It represents an array of the **binary representation** using the values 1 and 0.  It is used when we need to store the bits but do not know the number of bits in advance. We can access items from the BitArray collection by using an **integer index**, which starts from zero. |

#### Dictionary:

* It returns error if we try to find a key which does not exist.
* It is faster than a Hashtable because there is no boxing and unboxing.
* Only public static members are thread safe.
* Dictionary is a generic type which means we can use it with any data type.

#### Hashtable:

* It returns null if we try to find a key which does not exist.
* It is slower than dictionary because it requires boxing and unboxing.
* All the members in a Hashtable are thread safe,
* Hashtable is not a generic type,

**Hash Table**

using System.Collections;

class Program

{

static Hashtable GetHashtable()

{

// Create and return new Hashtable.

Hashtable hashtable = new Hashtable();

hashtable.Add("Area", 1000);

hashtable.Add("Perimeter", 55);

hashtable.Add("Mortgage", 540);

return hashtable;

}

public static void Main()

{

Hashtable hashtable = GetHashtable();

// See if the Hashtable contains this key.

Console.WriteLine(hashtable.ContainsKey("Perimeter"));

// Test the Contains method. It works the same way.

Console.WriteLine(hashtable.Contains("Area"));

// Get value of Area with indexer.

int value = (int)hashtable["Area"];

// Write the value of Area.

Console.WriteLine(value);

}

}

Output:

True

True

1000

**Dictionary**

class Dict

{

static void Main()

{

// Example Dictionary again

Dictionary<string, int> d = new Dictionary<string, int>()

{

{"Lion", 2}, {"dog", 1}};

// Loop over pairs with foreach

foreach (KeyValuePair<string, int> pair in d)

{

Console.WriteLine ("{0}, {1}",pair.Key, pair.Value);

}

foreach (var pair in d)

{

Console.WriteLine("{0}, {1}", pair.Key, pair.Value);

}

Console.ReadKey();

}

}

**GENERIC CLASS**

Generic classes encapsulate operations that are not specific to a particular data type. The most common use for generic classes is with collections like linked lists, hash tables, stacks, queues, trees, and so on. Operations such as adding and removing items from the collection are performed in basically the same way regardless of the type of data being stored.

### Common Language Runtime (CLR):

The most important part of the .NET Framework is the .Net Common Language Runtime (CLR) also called .Net Runtime in short. It is a framework layer that resides above the Operating System and handles/manages the execution of the .NET applications. Our .Net programs don't directly communicate with the Operating System but through CLR.

### MSIL (Microsoft Intermediate Language) Code:

When we compile our .Net Program using any .Net compliant language like (C#, VB.NET, C++.NET) it does not get converted into the executable binary code but to an intermediate code, called MSIL or IL in short, understandable by CLR. MSIL is an OS and H/w independent code. When the program needs to be executed, this MSIL or intermediate code is converted to binary executable code, called native code. The presence of IL makes it possible the Cross Language Relationship as all the .Net compliant languages produce the similar standard IL code.

### 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. So, if a program runs for sometime, then it won't have any just in time performance penalty. As JITers are aware of processor and OS exactly at runtime, they can optimize the code extremely efficiently resulting in very robust applications. Also, since JITer knows the exact current state of executable code, they can also optimize the code by in-lining small function calls (like replacing body of small function when its called in a loop, saving the function call time). Although, Microsoft stated that C# and .Net are not competing with languages like C++ in efficiency, speed of execution, JITers can make wer code even faster than C++ code in some cases when program is run over extended period of time (like web-servers).

### Common Language Specification (CLS):

Earlier we used the term '.NET Compliant Language' and stated that all the .NET compliant languages can make use of CLR and FCL. But what makes a language '.NET compliant language'? The answer is Common Language Specification (CLS). Microsoft has released a small set of specification that each language should meet to qualify as a .NET Compliant Language. As IL is a very rich language, it is not necessary for a language to implement all the IL functionality, rather it meets the small subset of it, CLS, to qualify as a .NET compliant language, which is the reason why so many languages (procedural and OO) are now running under .Net umbrella. CLS basically addresses to language design issues and lays certain standards like there should be no global function declaration, no pointers, no multiple inheritance and things like that. The important point to note here is that if we keep wer code within CLS boundary, wer code is guaranteed to be usable in any other .Net language.

### Common Type System (CTS):

.NET also defines a Common Type System (CTS). Like CLS, CTS is also a set of standards. CTS defines the basic data types that IL understands. Each .NET compliant language should map its data types to these standard data types. This makes it possible for the 2 languages to communicate with each other by passing/receiving parameters to/from each other. For example, CTS defines a type Int32, an integral data type of 32 bits (4 bytes) which is mapped by C# through int and VB.Net through its Integer data type.

### Garbage Collector (GC):

CLR also contains Garbage Collector (GC) which runs in a low-priority thread and checks for un-referenced dynamically allocated memory space. If it finds some data that is no more referenced by any variable/reference, it re-claims it and returns the occupied memory back to the Operating System; so that it can be used by other programs as necessary. The presence of standard Garbage Collector frees the programmer from keeping track of dangling data.

# Globalization and Localization

Globalization is the process of designing and developing applications that function for multiple cultures.

Localization is the process of customizing the application for a given culture and locale.

# Page Life Cycle Events

|  |  |
| --- | --- |
| **PreInit** | All the Dynamic Controls, Master pages, Profiles and themes should be set in this event. |
| **Init** | This should be used to set the initial value of the Control properties. |
| **InitComplete** | This should be used to have custom ViewState data. This is the first place where ViewState has been loaded and can be changed. |
| **Preload** | This can be used to set the properties of the controls. |
| **Load** | All the Database connections and Data Binding can be performed here. Before this event finishes up all the validations will be done. Once the event is finished, the events for all the controls will execute before calling the next event in this list. |
| **LoadComplete** | This event can be used for activities on controls that require them to be fully loaded. |
| **PreRender** | This is the last page where the visual properties of the controls can be changed before getting them displayed on the page. |
| **PreRenderComplete** | This will be called when the page is ready and no changes in visual elements can be made. All data binding are done at this point. |
| **SaveStateComplete** | View State has been saved and from this point onwards changes in ViewState will not be preserved i.e. ViewState has been saved for the page already. |
| **Unload** | The Page processing is done now. This is the last event that will be called. |

# State Management

**Client State:**

* View state
* Control state
* Hidden field
* Cookie
* Query string

**Server State:**

* Application state
* Session state
* Profile Properties

### [View State](javascript:void(0))

The [ViewState](https://msdn.microsoft.com/en-us/library/system.web.ui.control.viewstate%28v=vs.140%29.aspx) property provides a dictionary object for retaining values between multiple requests for the same page. This is the default method that the page uses to preserve page and control property values between round trips.

When the page is processed, the current state of the page and controls is hashed into a string and saved in the page as a hidden field, or multiple hidden fields if the amount of data stored in the [ViewState](https://msdn.microsoft.com/en-us/library/system.web.ui.control.viewstate%28v=vs.140%29.aspx) property exceeds the specified value in the [MaxPageStateFieldLength](https://msdn.microsoft.com/en-us/library/system.web.ui.page.maxpagestatefieldlength%28v=vs.140%29.aspx) property. When the page is posted back to the server, the page parses the view-state string at page initialization and restores property information in the page. We can store values in view state as well.

### [Control State](javascript:void(0))

Sometimes we need to store control-state data in order for a control to work properly. For example, if we have written a custom control that has different tabs that show different information, in order for that control to work as expected, the control needs to know which tab is selected between round trips. The [ViewState](https://msdn.microsoft.com/en-us/library/system.web.ui.control.viewstate%28v=vs.140%29.aspx) property can be used for this purpose, but view state can be turned off at a page level by developers, effectively breaking wer control. To solve this, the ASP.NET page framework exposes a feature in ASP.NET called control state.

The [ControlState](https://msdn.microsoft.com/en-us/library/system.web.ui.pagestatepersister.controlstate%28v=vs.140%29.aspx) property allows we to persist property information that is specific to a control and cannot be turned off like the [ViewState](https://msdn.microsoft.com/en-us/library/system.web.ui.control.viewstate%28v=vs.140%29.aspx) property.

### Hidden Field

ASP.NET allows we to store information in a [HiddenField](https://msdn.microsoft.com/en-us/library/system.web.ui.webcontrols.hiddenfield%28v=vs.140%29.aspx) control, which renders as a standard HTML hidden field. A hidden field does not render visibly in the browser, but we can set its properties just as we can with a standard control. When a page is submitted to the server, the content of a hidden field is sent in the HTTP form collection along with the values of other controls. A hidden field acts as a repository for any page-specific information that we want to store directly in the page.

A [HiddenField](https://msdn.microsoft.com/en-us/library/system.web.ui.webcontrols.hiddenfield%28v=vs.140%29.aspx) control stores a single variable in its [Value](https://msdn.microsoft.com/en-us/library/system.web.ui.webcontrols.hiddenfield.value%28v=vs.140%29.aspx) property and must be explicitly added to the page.

### [Cookie](javascript:void(0))

A cookie is a small amount of data that is stored either in a text file on the client file system or in-memory in the client browser session. It contains site-specific information that the server sends to the client along with page output. Cookies can be temporary (with specific expiration times and dates) or persistent.

We can use cookies to store information about a particular client, session, or application. The cookies are saved on the client device, and when the browser requests a page, the client sends the information in the cookie along with the request information. The server can read the cookie and extract its value. A typical use is to store a token (perhaps encrypted) indicating that the user has already been authenticated in wer application.

### Query String

A query string is information that is appended to the end of a page URL.

Query strings provide a simple but limited way to maintain state information. For example, they are an easy way to pass information from one page to another, such as passing a product number from one page to another page where it will be processed. However, some browsers and client devices impose a 2083-character limit on the length of the URL.

## [Server-Based State Management Options](javascript:void(0))

ASP.NET offers us a variety of ways to maintain state information on the server, rather than persisting information on the client. With server-based state management, we can decrease the amount of information sent to the client in order to preserve state, however it can use costly resources on the server. The following sections describe three server-based state management features: application state, session state, and profile properties.

### [Application State](javascript:void(0))

ASP.NET allows we to save values using application state — which is an instance of the [HttpApplicationState](https://msdn.microsoft.com/en-us/library/system.web.httpapplicationstate%28v=vs.140%29.aspx) class — for each active Web application. Application state is a global storage mechanism that is accessible from all pages in the Web application. Thus, application state is useful for storing information that needs to be maintained between server round trips and between requests for pages.

Application state is stored in a key/value dictionary that is created during each request to a specific URL. We can add wer application-specific information to this structure to store it between page requests.

Once we add wer application-specific information to application state, the server manages it.

### [Session State](javascript:void(0))

Session state is similar to application state, except that it is scoped to the current browser session. If different users are using wer application, each user session will have a different session state. In addition, if a user leaves wer application and then returns later, the second user session will have a different session state from the first.

Session state is structured as a key/value dictionary for storing session-specific information that needs to be maintained between server round trips and between requests for pages.

### [Profile Properties](javascript:void(0))

To use profile properties, you must configure a profile provider. ASP.NET includes a [SqlProfileProvider](https://msdn.microsoft.com/en-us/library/system.web.profile.sqlprofileprovider%28v=vs.140%29.aspx) class that allows you to store profile data in a SQL database, but you can also create your own profile provider class that stores profile data in a custom format and to a custom storage mechanism such as an XML file, or even to a web service.

**Difference between String and StringBuilder**

|  |  |
| --- | --- |
| **String** | **StringBuilder** |
| It’s an immutable means once we create string object we cannot modify. | It’s mutable means once we create string builder object we can perform any operation. |
| Performance wise string is slow because every time it will create new instance. | Performance wise stringbuilder is high because it will use same instance of object to perform any action. |
| In string we don’t have append keyword. | In StringBuilder we can use append keyword. |
| String belongs to **System** namespace. | Stringbuilder belongs to **System.Text** namespace. |

**What is difference between constants, read-only and, static?**  
**Constants:** The value can’t be changed.

**Read-only:** The value will be initialized only once from the constructor of the class.

**Static:** Value can be initialized once.

## Constant

Constant fields or local variables must be assigned a value at the time of declaration and after that they cannot be modified. By default constant are static, hence you cannot define a constant type as static.

* Compile-time constant
* Can't be declared static
* Can't be modified or changed
* Can be of any type of Access Modifier
* Local scope
* Needs to get initialized
* Declared at the time of declaration

## ReadOnly

A readonly field can be initialized either at the time of declaration or with in the constructor of same class. Therefore, readonly fields can be used for run-time constants.

* Run-time constant
* It can be static
* Global scope
* Can be declared in the constructer class
* Generally public

## Static

The static keyword is used to specify a static member, which means static members are common to all the objects and they do not tied to a specific object. This keyword can be used with classes, fields, methods, properties, operators, events, and constructors, but it cannot be used with indexers, destructors, or types other than classes.

* Can't be used with indexers
* Works with constructors too
* By default it is private
* Can be parameterized or public too
* If its applied to a class then all the class members need to be static

**ASSEMBLY**

Assembly is really a collection of types and resource information that are built to work together and form a logical unit of functionality. Every Assembly we create contains one or more program files and a Manifest. There are two types program files: **Process Assemblies (EXE)** and **Library Assemblies (DLL)**. Each Assembly can have only one entry point (that is, DllMain, WinMain, or Main).

When we compile an application, the **MSIL** code created and it is stored in an **assembly.**

The following are the two types of assemblies:

* **Private Assembly** - Refers to the assembly that is used by a single application. Private assemblies are kept in a local folder in which the client application has been installed.
* **Public or Shared Assembly** - Refers to the assembly that is allowed to be shared by multiple applications. A shared assembly must reside in Global Assembly Cache (GAC) with a strong name assigned to it.

**NAMESPACE**

Namespace keyword is used to declare a scope that contains a set of related objects. Within a namespace, you can declare one or more of the following types:

* another namespace
* [class](https://msdn.microsoft.com/en-us/library/0b0thckt.aspx)
* [interface](https://msdn.microsoft.com/en-us/library/87d83y5b.aspx)
* [struct](https://msdn.microsoft.com/en-us/library/ah19swz4.aspx)
* [enum](https://msdn.microsoft.com/en-us/library/sbbt4032.aspx)
* [delegate](https://msdn.microsoft.com/en-us/library/900fyy8e.aspx)

System.Data is a namespace, System.Data.DLL (the file) is an assembly.

**Assembly**:

An assembly provides a fundamental unit of physical code grouping. It is an Output Unit. It is a unit of Deployment & a unit of versioning. Assemblies contain MSIL code.

**Namespace**:

A namespace provides a fundamental unit of logical code grouping. It is a Collection of names where in each name is Unique.[Namespace](http://msdn.microsoft.com/en-us/library/0d941h9d.aspx) is a logical grouping of classes belongs to same functionality. Namespace must be specified in Project-Properties.

* An assembly provides a fundamental unit of physical code grouping.
* A namespace provides a fundamental unit of logical code grouping.

### Global.asax:

Global.asax is basically ASP.NET Application file. It’s a place to write code for Application-level events such as Application start, Application end, Session start and end, Application error etc. raised by ASP.NET or by HTTP Modules.

There is a good list of events that are fired but following are few of the important events in Global.asax:

* **Application\_Init** occurs in case of application initialization for the very first time.
* **Application\_Start** fires on application start.
* **Session\_Start** fires when a new user session starts
* **Application\_Error** occurs in case of an unhandled exception generated from application.
* **Session\_End** fires when user session ends.
* **Application\_End** fires when application ends or time out.

### What are Session state modes in ASP.NET?

ASP.NET supports different session state storage options:

* **In-Process** is the default approach. It stores session state locally on same web server memory where the application is running.
* **StateServer** mode stores session state in a process other than the one where application is running. Naturally, it has added advantages that session state is accessible from multiple web servers in a Web Form and also session state will remain preserved even web application is restarted.
* **SQLServer** mode stores session state in SQL Server database. It has the same advantages as that of StateServer.
* **Custom** modes allows to define our custom storage provider.

The default **timeout** of session is 20 minutes.

By default session id is **stored** in cookies. If cookies is disabled, it will append to the url of the page.

**ACID PROPERTY**

ACID are of set of properties that guarantee that database transactions are processed reliably. (A single logical operation on the data is called a Transaction.)

1. **Atomicity**

It says, Database transaction should complete 100% or zero.

Atomicity refers to the ability of the database to guarantee that either all of the tasks of a transaction are performed or none of them are.

1. **Consistency**

This property ensures that database transaction will bring database from one valid sate to another.

The consistency property ensures that the database remains in a consistent state before the start of the transaction and after the transaction is over (whether successful or not).

1. **Isolation**

Every transaction should be independent to each other. No transaction will affect the existence of any other transaction.

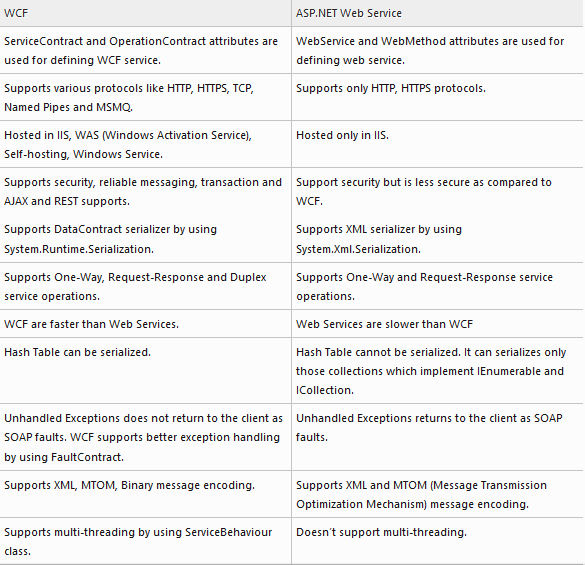
1. **Durability**

Change need to hold permanently one transaction is termed as successful. Durability refers to the ability of the system to recover committed transaction updates if either the system or the storage media fails.

**Difference Between Truncate, Delete Commands**

* TRUNCATE is a DDL command whereas DELETE is a DML command.
* TRUNCATE is much faster than DELETE.
* You can’t rollback in TRUNCATE but in DELETE you can rollback. TRUNCATE removes the record permanently.
* In case of TRUNCATE, Trigger doesn't get fired. But in DML commands like DELETE .Trigger get fired.
* You can't use conditions (WHERE clause) in TRUNCATE. But in DELETE you can write conditions using WHERE clause.

**Difference Between Wcf And Asp.Net Web Service**



**[Clustered & Non-Clustered Index](http://www.dotnetfunda.com/interviews/show/46/what-is-clustered-non-clustered-index" \o "What is Clustered & Non-Clustered Index?" \t "_blank)**

**Clustered Index:**

Clustered index physically rearrange the data that users inserts into table. A **clustered index** physically order the data on the disk, this is why you can have only one per table.   
  
**Non-Clustered Index:**

Non-Clustered Index contains pointers to the data that is stored in the data page. A **Non-Clustered Index** defines a logical order that does not match the physical order on disk. It is a kind of index backside of the book where you see only the reference of a kind of data.

**Clustered Index**

* There can be only one clustered index for a table.
* Faster to read than non clustered as data is physically stored in index order.
* Physically exist.
* Usually made on the primary key.

**Non Clustered Index**

* There can be only 249 non-clustered indexes for a table. Before SQL Server 2008 only 249 non-clustered Indexes can be created. With SQL Server 2008 and above 999 non-clustered Indexes can be created.
* Quicker for insert and update operations than a clustered index.
* Logically exist.
* Usually made on the any key.

**ACID Property**

**Atomicity** is an all-or-none proposition. This transaction is either fully completed or not begun at all.

**Consistency** guarantees that a transaction never leaves your database in a half-finished state. Data is either committed or roll back.

**Isolation** keeps transactions separated from each other until they’re finished.

**Durability** guarantees that it prevents the loss of information, even in the case of system failure.

**Difference between ExecuteScalar, ExecuteReader and ExecuteNonQuery**

**ExecuteScalar()** only returns the value from the first column of the first row of your query.  
**ExecuteReader()** returns an object that can iterate over the entire result set.  
**ExecuteNonQuery()** does not return data at all: only the number of rows affected by an insert, update, or delete.

**ExecuteNonQuery():**

1. will work with Action Queries only (Create,Alter,Drop,Insert,Update,Delete).
2. Returns the count of rows effected by the Query.
3. Return type is int
4. Return value is optional and can be assigned to an integer variable.

**ExecuteReader():**

1. will work with Action and Non-Action Queries (Select)
2. Returns the collection of rows selected by the Query.
3. Return type is DataReader.
4. Return value is compulsory and should be assigned to an another object DataReader.

**ExecuteScalar():**

1. will work with Non-Action Queries that contain aggregate functions.
2. Return the first row and first column value of the query result.
3. Return type is object.
4. Return value is compulsory and should be assigned to a variable of required type.

**Serialization**

Serialization is the process of converting an object into a stream of bytes.

**Cookies**

Cookies are a State Management Technique that can store the values of control after a post-back. Cookies can store user-specific Information on the Client's machine like when the user last visited your site. Cookies are also known by many names, such as HTTP Cookies, Browser Cookies, Web Cookies, Session Cookies and so on. Basically cookies are a small text file sent by the web server and saved by the Web Browser on the Client's Machine.  
  
Basically Cookies are the following 2 types:

1. **Persistent Cookies:** Persistent Cookies are Permanent Cookies stored as a text file in the hard disk of the computer.
2. **Non-Persistent Cookies:** Non-Persistent cookies are temporary. They are also called in-memory cookies and session-based cookies. These cookies are active as long as the browser remains active, in other words if the browser is closed then the cookies automatically expire.

**Limitation of the Cookies**

1. The size of cookies is limited to 4096 bytes.
2. A total of 20 cookies can be used in a single website.
3. Default expiration time of a cookie is 30min.

# Connected V/S 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 disconnection 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 |

**Differences between Array and ArrayList**

* Array stores the values or elements of same data type but arraylist stores values of different datatypes.
* Arrays will use the fixed length but arraylist does not uses fixed length like array.

**Cookie-less Session**  
  
By default a session uses a cookie in the background. To enable a cookie-less session, we need to change some configuration in the Web.Config file. Follow these steps:

1. Open Web.Config file
2. Add a <sessionState> tag under <system.web> tag
3. Add an attribute "cookieless" in the <sessionState> tag and set its value to "AutoDetect" like below:

<sessionState cookieless="AutoDetect" regenerateExpiredSessionId="true"/>

The possible values for "cookieless" attribute are:

* AutoDetect : Session uses background cookie if cookies are enabled. If cookies are disabled, then the URL is used to store session information.
* UseCookie: Session always use background cookie. This is default.
* UseDeviceProfile: Session uses background cookie if browser supports cookies else URL is used.
* UseUri: Session always use URL.

**What are Globalization and localization? How to implement them?**

Globalization is the concept of developing the application in more than one language while the Localization is used for a particular language. Like if we develop the application in more than one language we need to create the resource files (.resx) by using System. Globalization and when we open the application in a particular language, then the localizations used to convert that application to the selected language.

**What is Strong Name?**

Strong Name (SN) is used to make the dll as the unique not by its name but by its version as:  
SN -k fileName.dll  
Now it will have the unique name with respect to the version. This assembly when placed in the GAC, it will treat as the unique with its version number and other details. 2 assemblies with the same name can exist in the GAC but both will have different version. The CLR takes the latest version assembly while running the application.

**What is assembly, GAC? Where they are physically located?**

Assembly is the collection of classes, namespaces, methods, properties which may be developed in different language and packed as a dll. So we can say that dll is also called as assembly.  
There are 3 types of assemblies-   
- Private Assembly, Shared Assembly, and Satellite Assembly.  
GAC (Global Assembly Cache)- When the assembly is  required by more than one project or application, we need to make the assembly with strong name and keep it in GAC or in Assembly folder by installing the assembly with the GACUtil command.  
To make the assembly with strong name:  
SN -k MyDll.dll  
And to install it in GAC:  
GacUtil -i MyDll.dll  
GAC assemblies are physically stored in Assembly folder in the system.

**Strong typing and Weak Typing**

**Strong typing:** It checks the type of variables as soon as possible, usually at compile time. It prevents mixing operations between mismatched types.

**Weak Typing:** While weak typing is delaying checking the types of the system as late as possible, usually to run-time. In this you can mix types without an explicit conversion.

**What is a Delegate?**

Delegate is a type which holds the method(s) reference in an object. It is also referred to as a type safe function pointer.

A delegate is like a pointer to a function. It is a reference type data type and it holds the reference of a method. All the delegates are implicitly derived from System.Delegate class.

A function can have one or more parameters of different data types, but what if you want to pass a function itself as a parameter, it is Delegate.

Advantages:-

Encapsulating the method's call from caller

Effective use of delegate improves the performance of application

Used to call a method asynchronously

Example:-

using System;

delegate int NumberChanger(int n);

namespace DelegateAppl

{

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 nc1 = new NumberChanger(AddNum);

NumberChanger nc2 = new NumberChanger(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();

}

}

}

Result:-

Value of Num: 35

Value of Num: 175

**What are Endpoints?**

The collection of Address, Binding and Contract is called as End Point. In Sort,  
EndPoint = A+B+C  
Address (Where)-  It means where the service is hosted. URL of the service shows the address.  
Binding (How)- How to connect to the service, is defined by the Binding. It basically has the definition of the communication channel to communicate to the WCF service  
Contract (what)- It means what the service contains for the client. What all the methods are implemented in the WCF service is implemented in the Contract.