[**Interview Questions in ASP.NET,C#.NET,SQL Server,.NET Framework**](http://www.aspdotnet-suresh.com/2010/05/interview-questions-in-aspnetcnetsql.html)

**Difference:**

**=> Session VS Cookies VS** **LocalStorage VS SessionStorage**LocalStorage is not session base, Need to delete via JS or manually

SessionStorage is session base, will delete after tab/browser close

Cookies expiry depend of what set,

**=> StringBuilder VS string**

The main difference is system.string is immutable and system.stringbuilder is a mutable. Append keyword is used in string builder but not in system.string.

Immutable means once we created we cannot modified. Suppose if we want give new value to old value simply it will discarded the old value and it will create new instance in memory to hold the new value.

**=> StringBuffer and StringBuilder Class**

|  |  |  |
| --- | --- | --- |
| No | StringBuffer | StringBuilder |
| 1 | [StringBuffer class](http://www.c-sharpcorner.com/UploadFile/9a9e6f/learn-stringbuffer-class-in-java-lecture-1/) is synchronized | [StringBuilder class](http://www.c-sharpcorner.com/Blogs/48859/stringbuilder-class-in-java.aspx) is not synchronized |
| 2 | It is thread-safe | It is not thread-safe |
| 3 | It is slower than StringBuilder | It is faster than StringBuffer |
| 4 | It is available from Java 1.0 | It is available from Java 5.0 |

**Synchronization, thread safety & speed**

* **StringBuffer** is **synchronized**, that's why it is also **thread-safe**. In other words, two or more threads cannot call the methods of StringBuffer simultaneously.  
  In a synchronized environment, a single thread can perform a certain operation rather than disturbing any other thread that makes StringBuffer **slower** because it is synchronized.
* **StringBuilder** is not **synchronized** and that's why it is **not thread-safe**. That means that two or more threads can call the methods of StringBuilder simultaneously. Since it is non-synchronized and not thread-safe, it can perform **faster** because there is no overhead of acquiring and releasing of locks associated with synchronized methods (StringBuffer).

**Note:** The only similarity between the two is that they both are mutable.

**=> Differences between Array and ArrayList**

|  |  |  |
| --- | --- | --- |
| **Sr** | **Array** | **ArrayList** |
| 1 | Array is strongly typed. This means that an array can store only specific type of items\elements | ArrayList can store any type of items\elements. |
| 2 | In arrays we can store only one datatype either int, string, char etc… | In arraylist we can store all the datatype values |
| 3 | Array can’t accept null | ArrayList collection accepts null |
| 4 | Arrays belong to System.Array namespace using System; | Arraylist belongs to System.Collection namespaces using System.Collections; |
| 5 | Example - int[] intArray=new int[]{2}; intArray[0] = 1; intArray[2] = 2; | Example - ArrayList Arrlst = new ArrayList(); Arrlst.Add("Sagar"); Arrlst.Add(1); Arrlst.Add(null); |

**=> Array list and Hash table?**

1) Hash table store data as name, value pair. While in array only value is store.

2) To access value from hash table, you need to pass name. While in array, to access value, you need to pass index number.

3) you can store different type of data in hash table, say int, string etc. while in array you can store only similar type of data.

**=> What is difference between array and array list?**

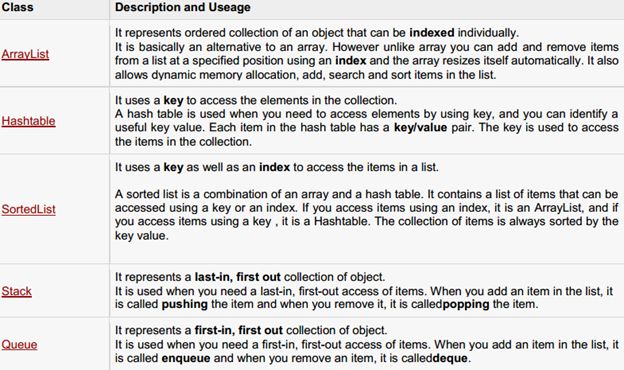
| **Array** | **ArrayList** |
| --- | --- |
| Array is strongly typed. This means that an array can store only specific type of items\elements. | ArrayList can store any type of items\elements. |
| Array stores fixed number of elements. Size of an Array must be specified at the time of initialization. | ArrayList grows automatically and you don't need to specify size. |
| No need to cast elements of an array while retriving because it is strongly type and stores specific type of items only. | Items of ArrayList need to be cast to appropriate data type while retriving. |
| Use static helper class Array to perform different tasks on the array. | ArrayList itself includes various utility methods for various tasks. |

http://www.aspdotnet-suresh.com/2013/09/difference-bw-array-and-arraylist-in-csharp-example.html

|  |  |
| --- | --- |
| **Arrays** | **ArrayLists** |
| These are strong type collection and allow to store fixed length | Array Lists are not strong type collection and size will increase or decrease dynamically |
| In arrays we can store only one datatype either int, string, char etc… | In arraylist we can store all the datatype values |
| Arrays belong to System.Array namespace | Arraylist belongs to System.Collection namespaces |

**=> Array list VS Hash table or Array List, Hash Table, Sorted List, Stack And Queue**

Ref: https://www.c-sharpcorner.com/UploadFile/57a357/overview-of-collection-array-list-hash-table-sorted-list/



## **ArrayList**

ArrayList represents an ordered collection of a specified object that can be indexed individually. It allows dynamic memory allocation, adding, searching and sorting items in the list.  
  
The following are the properties of the ArrayList class:

1. Capacity: Gets or sets the number of elements that the ArrayList can contain.
2. Count: Returns the number of elements present in the ArrayList.
3. IsFixedSize: Returns a value indicating whether the ArrayList has a fixed size.
4. IsReadOnly: Returns a value indicating whether the ArrayList is read-only.
5. Item: Gets or sets the element at the specified index position.

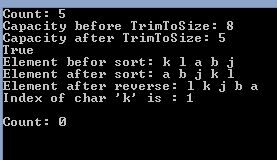
The following are the methods of the ArrayList class,

1. public virtual int add(object value); Inserts the object at the end of the ArrayList.
2. public virtual void AddRange(Icollection c); Adds the elements of a collection to the end of the ArrayList.
3. public virtual void Clear(); Removes all the elements from the ArrayList. Does not affect the capacity of the ArrayList.
4. public virtual bool Contains(object item); Determines whether an element is present or not in the ArrayList.
5. public virtual ArrayList GetRange( int index, int count ); Returns an ArrayList that represents a subset of the elements in the source ArrayList.
6. public virtual int IndexOf(object); Returns the index of the first occurrence of a value in the ArrayList or in a portion of it.
7. public virtual void Insert(int index, object value); Inserts an element into the ArrayList at the specified index.
8. public virtual void InsertRange(int index, ICollection c); Inserts the element of a collection into the ArrayList at the specified index.
9. Public virtual void Remove(object obj); Removes the first occurrence of a specified object from the ArrayList.
10. public virtual void RemoveAt(int index); Removes the element at the given specified index of the ArrayList.
11. public virtual void RemoveRange(int index,int count); Removes a range of elements from the ArrayList.
12. public virtual void Reverse(); Reverses the order of the elements in the ArrayList.
13. public virtual void Sort(); Sorts all the elements in the ArrayList.
14. public virtual void TrimToSize(); Sets the capacity to the actual number of elements present in the ArrayList. Basically the capacity is not increased one by one. Capacity is just doubled each time whenever the size reaches the threshold. So the TrimToSize() method sets the capacity to the exact the size of the ArrayList.

The following is the example,

1. **using** System;
2. **using** System.Collections;
4. **namespace** Collection\_Example
5. {
6. **class** Program
7. {
8. **static** **void** Main(**string**[] args)
9. {
10. ArrayList al = **new** ArrayList();
11. ArrayList al1 = **new** ArrayList();
12. // Adding object into the ArrayList
13. al1.Add('a');
14. al1.Add('b');
15. al.Add('k');
16. al.Add('l');
17. al.Add('j');
18. // Adding Arraylist at specific position into the ArrayList
19. al.InsertRange(2,al1);
20. //Get the Capacity and number of element present in the ArrayList
21. // Note that Capacity and Count are not equal
22. Console.WriteLine("Count: {0}", al.Count);
23. Console.WriteLine("Capacity before TrimToSize: {0} ", al.Capacity);
24. al.TrimToSize();
25. Console.WriteLine("Capacity after TrimToSize: {0} ", al.Capacity);
26. Console.WriteLine(al.Contains('b'));
27. Console.Write("Element befor sort: ");
28. **foreach** (**object** obj **in** al)
29. Console.Write(obj + " ");
30. Console.Write("\nElement after sort: ");
31. al.Sort();
32. **foreach** (**object** obj **in** al)
33. Console.Write(obj + " ");
34. al.Reverse();
35. Console.Write("\nElement after reverse: ");
36. **foreach** (**object** obj **in** al)
37. Console.Write(obj + " ");
38. Console.WriteLine("\nIndex of char 'k' is : {0}", al.IndexOf('k'));
39. // clear the ArrayList
40. al.Clear();
41. Console.WriteLine("\nCount: {0}", al.Count);
42. Console.ReadKey();
43. }
44. }
45. }

**Output**



## **HashTable**

The Hashtable class represents a collection of key-and-value pairs organized based on the hash code of the key. It uses the key to access the elements in the collection. Hash table is used when you need to access elements using a key. Each item in the hash table has a key/value pair. The key is used to access the items in the collection.  
  
The following are the properties of HashTable class,

1. Count: Returns the number of elements present in the Hashtable.
2. IsFixedSize: Returns a value indicating whether the Hashtable has a fixed size.
3. IsReadOnly: Returns a value indicating whether the Hashtable is read-only.
4. Item: Gets or sets the value with the specific key.
5. Keys: Returns an ICollection containing the keys in the HashTable.
6. Values: Returns an ICollection containing the values in the HashTable.

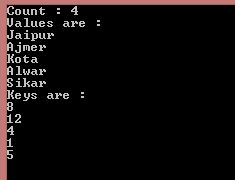
The following are the methods of the HashTable Class,

1. public virtual void add(object key,object value); Adds a value with the specified key into the HashTable.
2. public virtual void Clear(); Clears the HashTable.
3. public virtual bool ContainsKey(object key); Determines whether the HashTable contains a specific key, if Yes then returns true otherwise it returns false.
4. public virtual bool ContainsValue(object key); Determines whether the HashTable contains a specific value, if Yes then returns true otherwise it returns false.
5. public virtual void Remove(object key); Removes an element with the specific key from the HashTable.

The following is the example:

1. **using** System;
2. **using** System.Collections;
4. **namespace** Collection\_Example
5. {
6. **class** Program
7. {
8. **static** **void** Main(**string**[] args)
9. {
10. Hashtable ht = **new** Hashtable();
11. //Adding item into HashTable
12. ht.Add(1, "Alwar");
13. ht.Add(12, "Ajmer");
14. ht.Add(8, "Jaipur");
15. ht.Add(4, "Kota");
16. Console.WriteLine("Count : {0}", ht.Count);
17. **if** (ht.ContainsValue("Sikar"))
18. Console.WriteLine("Sikar is already exist in the HashTable");
19. **else**
20. ht.Add(5, "Sikar");
22. //Get a collection of values
23. Console.WriteLine("Values are :");
24. ICollection values = ht.Values;
25. **foreach** (**string** str **in** values)
26. Console.WriteLine(str);
27. //Get a collection of Keys
28. Console.WriteLine("Keys are :");
29. ICollection keys = ht.Keys;
30. **foreach** (**int** i **in** keys)
31. Console.WriteLine(i);
32. ht.Remove(3);
33. ht.Clear();
34. Console.ReadKey();
35. }
36. }
37. }

**Output**



## **SortedList**

SortedList represents a collection of key-value pairs sorted by the keys and are accessible by key and by index. 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. Note that SortedList is always sorted by the key value.  
  
The following are the properties of the SortedList class,

1. Capacity: Gets or sets the number of elements that the SortedList can contain.
2. Count: Returns the number of elements present in the SortedList.
3. IsFixedSize: Returns a value indicating whether the SortedList has a fixed size.
4. IsReadOnly: Returns a value indicating whether the SortedList is read-only.
5. Item: Gets or sets the value associated with a specific key in the SortedList.
6. Keys: Returns an ICollection containing the keys in the SortedList.
7. Values: Returns an ICollection containing the values in the SortedList.

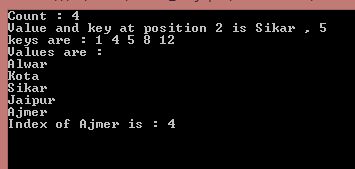
**Method of the SortedList Class**

1. public virtual void add(object key , object value); Adds a value with the specified key into the SortedList.
2. public virtual void Clear(); Clears the SortedList.
3. public virtual bool ContainsKey(object key); Determines whether the SortedList contains a specific key, if Yes then returns true otherwise it returns false.
4. public virtual bool ContainsValue(object key); Determines whether the SortedList contains a specific value, if Yes then returns true otherwise it returns false.
5. public virtual object GetByIndex(int index); It returns the value at the specified index of the SortedList.
6. public virtual object GetKey(int index); Returns the key at the specified index of the SortedList.
7. public virtual IList GetKeyList(); Gets the keys in the SortedList.
8. public virtual IList GetValueList(); Gets the values in the SortedList.
9. public virtual int IndexOfKey(object key); Gets the index of the specified key in the SortedList.
10. public virtual int IndexOfValue(object value); Gets the index of the first occurrence of the specified value in the SortedList.
11. public virtual void Remove(object key); Remove an element with the specific key from the SortedList.
12. public virtual void RemoveAt(int index); Removes the element at the specified index of SortedList.

The following is the example,

1. **using** System;
2. **using** System.Collections;
4. **namespace** Collection\_Example
5. {
6. **class** Program
7. {
8. **static** **void** Main(**string**[] args)
9. {
10. SortedList sl = **new** SortedList();
11. //Adding item into HashTable
12. sl.Add(1, "Alwar");
13. sl.Add(12, "Ajmer");
14. sl.Add(8, "Jaipur");
15. sl.Add(4, "Kota");
16. Console.WriteLine("Count : {0}", sl.Count);
17. **if** (sl.ContainsValue("Sikar"))
18. Console.WriteLine("Sikar is already exist in the SortedList");
19. **else**
20. sl.Add(5, "Sikar");
21. Console.WriteLine("Value and key at position 2 is {0} , {1}", sl.GetByIndex(2), sl.GetKey(2));
22. Console.Write("keys are : ");
23. **foreach** (**int** i **in** sl.Keys)
24. Console.Write(i + " ");
25. sl.Remove(3);
26. //Geting the keys and value from SortedList
27. IList keys = sl.GetKeyList();
28. IList values = sl.GetValueList();
29. Console.WriteLine("\nValues are :");
30. **foreach** (**object** obj **in** values)
31. Console.WriteLine(obj);
32. Console.WriteLine("Index of Ajmer is : {0}", sl.IndexOfValue("Ajmer"));
33. // Remove an element at specified index
34. sl.RemoveAt(2);
35. Console.ReadKey();
36. }
37. }
38. }

**Output**



## **Stack**

A Stack is a Last In First Out (LIFO) collection of objects. A Stack is used when you need last-in, first-out access to the objects. That means accessing the last inserting item. A Stack basically consists of two operations, Push and Pop. When you insert an element into the stack, it is called pushing the item and when you extract the item from the stack, it is called popping the item. Both Push and Pop are done at the top of the stack. To use the Stack data type in C# first you need to use the System.Collections namespace.  
  
The following are the properties of Stack:

1. Count: Returns the number of elements in the stack.

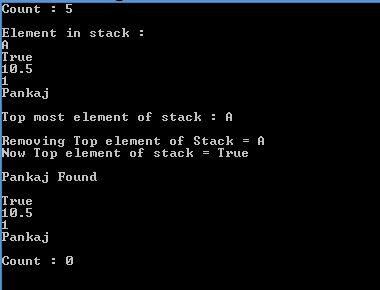
The following are the methods of stack:

1. public virtual void Push(object obj); Simply inserts an object at the top of the stack.
2. public virtual object Pop(object obj); Simply removes and returns the object from the top of the stack.
3. public virtual void Clear(); Clears the stack. Removes all the elements from the stack.
4. public virtual object Peek(); Returns the objects from the top of the stack (without removing).
5. public virtual object[] ToArray(); Copies the stack into an object array.
6. public virtual bool Contains(object obj); Checks whether an element exists in the stack. Returns True when an item exists in the stack otherwise it returns False.

The following is the example,

1. **using** System;
2. **using** System.Collections;
4. **namespace** Test\_Cdac
5. {
6. **class** Program
7. {
8. **static** **void** Main(**string**[] args)
9. {
10. // Declaring a stack
11. Stack st = **new** Stack();
12. // Inserting an element at the top of stack i.e. Push operation
13. st.Push("Pankaj");
14. st.Push(1);
15. st.Push(10.5);
16. st.Push(**true**);
17. st.Push('A');
18. //Get the number of elements contained in the stack
19. Console.WriteLine("Count : {0}", st.Count);
20. Console.WriteLine();
21. //Printing all the element of stack
22. Console.WriteLine("Element in stack : ");
23. **foreach** (**object** obj **in** st)
24. Console.WriteLine(obj);
25. Console.WriteLine();
26. //Returns the topmost element of the stack without removing
27. Console.WriteLine("Top most element of stack : {0}", st.Peek());
28. Console.WriteLine();
29. //Removes and Returns the topmost element of the stack i.e. Pop operation
30. **object** TopElement = st.Pop();
31. Console.WriteLine("Removing Top element of Stack = {0}\nNow Top element of stack = {1}\n", TopElement, st.Peek());
32. //Determines whether an element present or not in the stack
33. **if** (st.Contains("Pankaj"))
34. Console.WriteLine("Pankaj Found");
35. **else**
36. Console.WriteLine("Pankaj Not found");
37. //Copies the stack to a new Array(object)
38. Object[] ob=st.ToArray();
39. Console.WriteLine();
40. **foreach** (**object** obj **in** ob)
41. Console.WriteLine(obj);
42. //Removes all the element from stack
43. st.Clear();
44. Console.WriteLine();
45. Console.WriteLine("Count : {0}", st.Count);
46. Console.ReadKey();
47. }
48. }
49. }

**Output**



## **Queue**

A Queue is a First-In-First-Out (FIFO) collection of objects. Queue is used when you need first-in, first-out access to objects. That means accessing the first inserting item. A Queue basically consists of two operations, Enqueue and Dequeue. When you insert an element into a Queue, it is called Enqueue and when you extract an item from the Queue, it is called Dequeue. The Enqueue operation is done at the end of the queue and the Dequeue operation is done at end of the queue. To use the Queue data type in C# you need to use the System.Collections namespace.  
  
The following is the property of Queue,

1. Count: Returns the number of elements in the Queue.

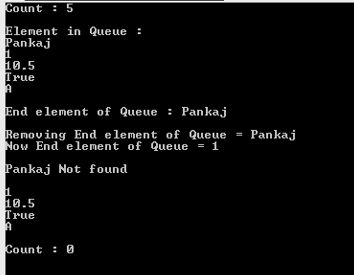
The following are the methods of Queue,

1. public virtual void Enqueue(object obj); Simply inserts an object at the end of the queue.
2. public virtual object Dequeue(object obj); Simply removes and returns the object from the front of the queue.
3. public virtual void Clear(); Clears the queue. This method removes all the elements from the queue.
4. public virtual object Peek(); This method returns the object from the front of the queue (without removing).
5. public virtual object[] ToArray(); Copies the queue into an object array.
6. public virtual bool Contains(object obj); Checks whether an element exists in the queue. Returns True when the item exists in the queue otherwise it returns False.
7. public virtual void TrimToSize(); Sets the capacity to the actual number of elements present in the Queue. Basically the capacity is not increased one by one. Capacity is just doubled each time whenever the size reaches the threshold. So the TrimToSize() method sets the capacity to the exact the size of the queue.

The following is the example,

1. **using** System;
2. **using** System.Collections;
4. **namespace** Teq\_Cdac
5. {
6. **class** Program
7. {
8. **static** **void** Main(**string**[] args)
9. {
10. // Declaring a Queue
11. Queue q = **new** Queue();
12. // Adds an element at the end of Queue i.e. Enqueue operation
13. q.Enqueue("Pankaj");
14. q.Enqueue(1);
15. q.Enqueue(10.5);
16. q.Enqueue(**true**);
17. q.Enqueue('A');
18. //Get the number of elements present in the Queue
19. Console.WriteLine("Count : {0}", q.Count);
20. Console.WriteLine();
21. //Printing all the element of Queue
22. Console.WriteLine("Element in Queue : ");
23. **foreach** (**object** obj **in** q)
24. Console.WriteLine(obj);
25. Console.WriteLine();
26. //Returns the end of the Queue without removing
27. Console.WriteLine("End element of Queue : {0}", q.Peek());
28. Console.WriteLine();
29. //Removes and Returns the end element of the Queue i.e. Dequeue operation
30. **object** TopElement = q.Dequeue();
31. Console.WriteLine("Removing End element of Queue = {0}\nNow End element of Queue = {1}\n", TopElement, q.Peek());
32. //Determines whether an element present or not in the Queue
33. **if** (q.Contains("Pankaj"))
34. Console.WriteLine("Pankaj Found");
35. **else**
36. Console.WriteLine("Pankaj Not found");
37. //Copies the qack to a new Array(object)
38. Object[] ob=q.ToArray();
39. Console.WriteLine();
40. **foreach** (**object** obj **in** ob)
41. Console.WriteLine(obj);
42. //Trim the Queue
43. q.TrimToSize();
44. //Removes all the element from Queue
45. q.Clear();
46. Console.WriteLine();
47. Console.WriteLine("Count : {0}", q.Count);
48. Console.ReadKey();
49. }
50. }
51. }

**Output**



**=> Structure VS Class**

|  |  |
| --- | --- |
| Class | Structure |
| Classes are of reference types. | Structs are of value types. |
| All the reference types are allocated on heap memory. | All the value types are allocated on stack memory. |
| Allocation of large reference type is cheaper than allocation of large value type. | Allocation and de-allocation is cheaper in value type as compare to reference type. |
| Class has limitless features. | Struct has limited features. |
| Class is generally used in large programs. | Struct are used in small programs. |
| Classes can contain constructor or destructor. | Structure does not contain parameter less constructor or destructor, but can contain Parameterized constructor or static constructor. |
| Classes used new keyword for creating instances. | Struct can create an instance, with or without new keyword. |
| A Class can inherit from another class. | A Struct is not allowed to inherit from another struct or class. |
| The data member of a class can be protected. | The data member of struct can’t be protected. |
| Function member of the class can be virtual or abstract. | Function member of the struct cannot be virtual or abstract. |
| Two variable of class can contain the reference of the same object and any operation on one variable can affect another variable. | Each variable in struct contains its own copy of data(except in ref and out parameter variable) and any operation on one variable can not effect another variable. |

**=> Abstract VS Interface**

**=> WCF and Web service and WebAPI**

<https://www.dotnettricks.com/learn/webapi/difference-between-wcf-and-web-api-and-wcf-rest-and-web-service>

## **Web Service**

* It is based on SOAP and return data in XML form.
* It support only HTTP protocol.
* It is not open source but can be consumed by any client that understands xml.
* It can be hosted only on IIS.

## **WCF**

* It is also based on SOAP and return data in XML form.
* It is the evolution of the web service(ASMX) and support various protocols like TCP, HTTP, HTTPS, Named Pipes, MSMQ.
* The main issue with WCF is, its tedious and extensive configuration.
* It is not open source but can be consumed by any client that understands xml.
* It can be hosted with in the applicaion or on IIS or using window service.

## **WCF Rest**

* To use WCF as WCF Rest service you have to enable webHttpBindings.
* It support HTTP GET and POST verbs by [WebGet] and [WebInvoke] attributes respectively.
* To enable other HTTP verbs you have to do some configuration in IIS to accept request of that particular verb on .svc files
* Passing data through parameters using a WebGet needs configuration. The UriTemplate must be specified.
* It support XML, JSON and ATOM data format.

## **Web API**

* This is the new framework for building HTTP services with easy and simple way.
* Web API is open source an ideal platform for building REST-ful services over the .NET Framework.
* Unlike WCF Rest service, it use the full feature of HTTP (like URIs, request/response headers, caching, versioning, various content formats)
* It also supports the MVC features such as routing, controllers, action results, filter, model binders, IOC container or dependency injection, unit testing that makes it more simple and robust.
* It can be hosted with in the application or on IIS.
* It is light weight architecture and good for devices which have limited bandwidth like smart phones.
* Responses are formatted by Web API’s MediaTypeFormatter into JSON, XML or whatever format you want to add as a MediaTypeFormatter.

To whom choose between WCF or WEB API

* Choose WCF when you want to create a service that should support special scenarios such as one way messaging, message queues, duplex communication etc.
* Choose WCF when you want to create a service that can use fast transport channels when available, such as TCP, Named Pipes, or maybe even UDP (in WCF 4.5), and you also want to support HTTP when all other transport channels are unavailable.
* Choose Web API when you want to create a resource-oriented services over HTTP that can use the full features of HTTP (like URIs, request/response headers, caching, versioning, various content formats).
* Choose Web API when you want to expose your service to a broad range of clients including browsers, mobiles, iphone and tablets.

**=> Ref Vs Out**

|  |  |
| --- | --- |
| **Ref** | **Out** |
| The parameter or argument must be initialized first before it is passed to ref. | It is not compulsory to initialize a parameter or argument before it is passed to an out. |
| It is not required to assign or initialize the value of a parameter (which is passed by ref) before returning to the calling method. | A called method is required to assign or initialize a value of a parameter (which is passed to an out) before returning to the calling method. |
| Passing a parameter value by Ref is useful when the called method is also needed to modify the pass parameter. | Declaring a parameter to an out method is useful when multiple values need to be returned from a function or method. |
| It is not compulsory to initialize a parameter value before using it in a calling method. | A parameter value must be initialized within the calling method before its use. |
| When we use REF, data can be passed bi-directionally. | When we use OUT data is passed only in a unidirectional way (from the called method to the caller method). |
| Both ref and out are treated differently at run time and they are treated the same at compile time. | |
| Properties are not variables, therefore it cannot be passed as an out or ref parameter. | |

**=> 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.

<http://www.c-sharpcorner.com/UploadFile/c210df/difference-between-const-readonly-and-static-readonly-in-C-Sharp/>

## **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.

A const field is a compile-time constant. A constant field or local variable can be initialized with a constant expression which must be fully evaluated at compile time.

You can apply const keyword to built-in value types (byte, short, int, long, char, float, double, decimal, bool), enum, a string literal, or a reference type which can be assigned with the value null.

Constants can be marked as public, private, protected, internal, or protected internal access modifiers.

Use the const modifier when you sure that the value a field or local variable would not be changed.

## **ReadOnly**

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

Explicitly, you can specify a readonly field as static since like constant by default it is not static. Readonly keyword can be applied to a value type and reference type (which initialized by using the new keyword) both. Also, the delegate and event could not be readonly.

Use the readonly modifier when you want to make a field constant at run time.

## **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 tie 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.

### Key points about Static keyword

1. If the static keyword is applied to a class, all the members of the class must be static.
2. Static methods can only access static members of the same class. Static properties are used to get or set the value of static fields of a class.
3. A static constructor can't be parameterized. Access modifiers cannot be applied on Static constructor, it is always a public default constructor which is used to initialize static fields of the class.

**=> IEnumerable and IQueryable?**

## **IEnumerable**

1. IEnumerable exists in System.Collections Namespace.
2. IEnumerable can move forward only over a collection, it can’t move backward and between the items.
3. IEnumerable is best to query data from in-memory collections like List, Array, etc.
4. While query data from a database, IEnumerable execute a select query on the server side, load data in-memory on a client-side and then filter data.
5. IEnumerable is suitable for LINQ to Object and LINQ to XML queries.
6. IEnumerable supports deferred execution.
7. IEnumerable doesn’t support custom query.
8. IEnumerable doesn’t support lazy loading. Hence not suitable for paging like scenarios.
9. Extension methods support by IEnumerable takes functional objects

## **IQueryable**

1. IQueryable exists in System. Linq Namespace.
2. IQueryable can move forward only over a collection, it can’t move backward and between the items.
3. IQueryable is best to query data from out-memory (like remote database, service) collections.
4. While query data from a database, IQueryable execute the select query on the server side with all filters.
5. IQueryable is suitable for[**LINQ to SQL**](https://www.dotnettricks.com/learn/entityframework/difference-between-linq-to-sql-and-entity-framework)queries.
6. IQueryable supports deferred execution.
7. IQueryable supports custom query using CreateQuery and Execute methods.
8. IQueryable support lazy loading. Hence it is suitable for paging like scenarios.
9. Extension methods support by IQueryable takes expression objects means expression tree.

**=> IEnumerable VS Collection VS List**

<https://medium.com/developers-arena/ienumerable-vs-icollection-vs-ilist-vs-iqueryable-in-c-2101351453db>

**=> Is vs As**

* The **is** operator is used to check if the run-time type of an object is compatible with the given type or not whereas **as** operator is used to perform conversion between compatible reference types or Nullable types.
* The **is** operator is of boolean type whereas **as** operator is not of boolean type.
* The **is** operator returns true if the given object is of the same type whereas **as** operator returns the object when they are compatible with the given type.
* The **is** operator returns false if the given object is not of the same type whereas **as** operator return null if the conversion is not possible.
* The **is** operator is used for only reference, boxing, and unboxing conversions whereas **as** operator is used only for nullable, reference and boxing conversions

**=> What is Language Integrated Query (LINQ)?**

LINQ is known as **Language Integrated Query** and it is introduced in .NET 3.5 and Visual Studio 2008. The beauty of LINQ is it provides the ability to .NET languages(like [C#](https://www.geeksforgeeks.org/csharp-programming-language/), VB.NET, etc.) to generate queries to retrieve data from the data source. For example, a program may get information from the student records or accessing employee records, etc. In, past years, such type of data is stored in a separate database from the application, and you need to learn different types of query language to access such type of data like SQL, XML, etc. And also you cannot create a query using C# language or any other .NET language.

To overcome such type of problems Microsoft developed LINQ. It attaches one, more power to the C# or .NET languages to generate a query for any LINQ compatible data source. And the best part is the syntax used to create a query is the same no matter which type of data source is used means the syntax of creating a query data in a relational database is same as that used to create query data stored in an array there is no need to use SQL or any other non-.NET language mechanism. You can also use LINQ with SQL, with XML files, with ADO.NET, with web services, and with any other database.

In C#, LINQ is present in System.Linq namespace. It provides different type of classes and methods which supports LINQ queries. In this namespace:

* Enumerable class holds standard query operators that operate on object which executes IEnumerable<T>.
* Queryable class holds standard query operators that operate on object which executes IQueryable<T>

**=> What is Sealed Class in C#? or How to stop inherintace in C#**

Sealed classes are used to restrict the inheritance feature of object oriented programming. Once a class is defined as a **sealed class,** this class cannot be inherited.

In C#, the sealed modifier is used to declare a class as **sealed**. In Visual Basic .NET, **NotInheritable** keyword serves the purpose of sealed. If a class is derived from a sealed class, compiler throws an error.  
  
If you have ever noticed, structs are sealed. You cannot derive a class from a struct.   
  
The following class definition defines a sealed class in C#:

1. // Sealed class
2. sealed **class** SealedClass
3. {
4. }

**=> What are the differences between value type and reference type?**

Value type contain variable and reference type are notcontaining value directly in its memory.

Memory is allocated in managed heap in reference type and in value type memory allocated in stack. Reference type ex-class value type-struct, enumeration

**=> What is boxing and unboxing concepts in .net?**Boxing is a process of converting value type into reference type  
Unboxing is a process of converting reference type to value type

**There two types.**

Value Type

Reference Type

**Value Types:**

A value type’s variables contain the value itself. I mean it doesn’t contain a pointer to the object. It does not store the data into heap memory. Different value types are

* Integral Types (sbyte, byte, short, ushort, int, uint, long, ulong), bool type, char type,   
  Floating point types(flaot,double) and the decimal types. They are all aliases of the .NET System Types.
* Struct Types
* Enumeration Types

A variable that contains reference or address of the actual data.

**Reference Type:**  
Reference types are allocated on the managed heap. Different reference   
types are

* The Object Type
* The class Type
* Interfaces
* Delegates
* The string type
* Arrays.

**=> What Partial class?**

Instead of defining an entire class, you can split the definition into multiple classes by using partial class keyword. When the application compiled, c# compiler will group all the partial classes together and treat them as a single class. There are a couple of good

reasons to use partial classes. Programmers can work on different parts of classes without needing to share same physical file

Ex:

Public partial class employee

{

Public void somefunction()

{

}  
}

Public partial class employee

{

Public void function ()

{

}

}

**=> What are generics in C#?**

<https://www.c-sharpcorner.com/UploadFile/84c85b/using-generics-with-C-Sharp>

**=> What is IL code, CLR, CTS, GAC & GC?**https://www.geeksforgeeks.org/common-language-runtime-clr-in-c-sharp/

**=> What is the difference between Virtual method and Abstract method?**

|  |  |
| --- | --- |
| **Virtual Method** | **Abstract Method** |
| It can be declared inside abstract as well as non abstract class. | It can be declared only inside abstract class. |
| **Virtual method** can have a method body. | **Abstract methods** have only the signature. It cannot have method body. |
| It may be overridden. | It must be overridden. |
| Class containing **virtual method** can be instantiated. | Class containing **abstract method** cannot be instantiated. It can only be inherited. |
| If you feel that the derived class may or may not override the base class method, then you will define the base class method as **virtual**. | If you want to enforce that derived class must override the base class method then you will define the base class method as **abstract**. |

**=> What is the difference between dynamic type variables and object type variables?**

**Object**

Each object in C# is derived from **object type** , either directly or indirectly. It is compile time variable and require boxing and unboxing for conversion and it makes it slow. You can change value type to reference type and vice versa.

Object type variables require to cast object variable to original type before using it. It means values of any types can be stored in object type variable. But **type conversion** (un-boxing) is required to get original type when the value of variable is retrieved in order to use it. It's actually increasing the overhead when we do this both operation. Allows to perform operation of given type once it gets cast any user defined or **primitive data type** . Also, Object type is useful when it doesn't have more information about the data type.

## **Dynamic**

Dynamic is run time variable and not require **boxing** and **unboxing** . You can assign and value to dynamic and also can change value type stored in same. All errors on dynamic can be discovered at run time only. We an also say that dynamic is a run time object which can hold any type of data.

**=> What is deep or shallow copy concept in C#?**

Shallow copies duplicate as little as possible. A shallow copy of a collection is a copy of the collection structure, not the elements. With a shallow copy, two collections now share the individual elements.

Deep copies duplicate everything. A deep copy of a collection is two collections with all of the elements in the original collection duplicated.

**=> What Are Access Modifiers In C#**

|  |  |
| --- | --- |
| Modifier | Description |
| public | There are no restrictions on accessing public members. |
| private | Access is limited to within the class definition. This is the default access modifier type if none is formally specified |
| protected | Access is limited to within the class definition and any class that inherits from the class |
| internal | Access is limited exclusively to classes defined within the current project assembly |
| protected internal | Access is limited to the current assembly and types derived from the containing class. All members in current project and all members in derived class can access the variables. |
| private protected | Access is limited to the containing class or types derived from the containing class within the current assembly. |

**=> What is the use of 'using' statement in C#?**

C# and .NET provide resource management for managed objects through the garbage collector - You do not have to explicitly allocate and release memory for managed objects. Clean-up operations for any unmanaged resources should be performed in the destructor in C#.

To allow the programmer to explicitly perform these clean-up activities, objects can provide a Dispose method that can be invoked when the object is no longer needed. The C# using statement defines a boundary for the object outside of which, the object is automatically destroyed. The using statement in C# is exited when the end of the "using" statement block or the execution exits the "using" statement block indirectly, for example - an exception is thrown.

The "using" statement allows you to specify multiple resources in a single statement. The object could also be created outside the "using" statement. The objects specified within the using block must implement the IDisposable interface. The framework invokes the Dispose method of objects specified within the "using" statement when the block is exited.

**=> What are extension methods in C#?**

In C#, the **extension method** concept allows you to add new methods in the existing class or in the structure without modifying the source code of the original type and you do not require any kind of special permission from the original type and there is no need to re-compile the original type. It is introduced in C# 3.0.

Let us discuss this concept with the help of an example. Suppose you have a class or a structure which contains three methods and you want to add two new methods in this class or structure, you did not have the source code of the class/structure, or do not have permissions from the class/structure, or the class is a sealed class, but you still want to add new methods in it, then you can use the concept extension method to add the new method in the existing class/structure. Now you create a new class which is static and contain the two methods which you want to add in the existing class, now bind this class with the existing class. After binding you will see the existing class can access the two new added methods. As shown in the below program.

**=> What is the difference between the dispose and finalize methods in C#?**

Methods dispose() and finalize() are the methods of C# which are invoked to free the unmanaged resources held by an object. The dispose() method is defined inside the interface IDisposable whereas, the method finalize() is defined inside the class object. The main difference between dispose() and finalize() is that the method **dispose**() has to be explicitly invoked by the user whereas, the method **finalize()** is invoked by the garbage collector, just before the object is destroyed.

**=> What is Delegates ?Types?**

**=> What is Events?**

**=> What is the Constructor Chaining in C#?**

**Constructor chaining** is an approach where a constructor calls another constructor in the same class or base class. It is very useful when we have a class that defines multiple constructors. Before knowing how to use constructor chaining in C#. Let’s first see why we need constructor chaining.

**=> Give a brief explanation on Thread Pooling in C#**

**=> What does the "volatile" keyword in C# mean?**

https://www.c-sharpcorner.com/UploadFile/1d42da/volatile-keyword-in-C-Sharp-threading/

**=> SOLID Principles In C#?**

**=> Design patterns C#?**

**=> What is Hashset in C#?**

https://www.c-sharpcorner.com/article/working-with-hashset-in-c-sharp/

**=> How interface is diffrent then class apart from the use of multiple inheritance?**

**=> Tools for code review in C#?**

[StyleCo](http://stylecop.codeplex.com/)

**=> Can I Create a method with same name in base class with different signature? Do I need to do anything**

**=> Can function return multiple values in C#?**

You can return multiple values from a function using either a dictionary, a tuple, **or a list**.

**=> What is dynamic type?**

n C# 4.0, a new type is introduced that is known as a dynamic type. It is used to avoid the compile-time type checking. The compiler does not check the type of the dynamic type variable at compile time, instead of this, the compiler gets the type at the run time. The dynamic type variable is created using dynamic keyword.

**Example:**

dynamic value = 123;

**=> Reflector in C#**

**=> Singleton?**

**=> What are the steps for the execution of an MVC project?**

The steps for the execution of an MVC project includes:

* Receive the first request for the application
* Perform routing
* Create an MVC request handler
* Create Controller
* Execute Controller
* Invoke action
* Execute Result

**=> What is the difference between Temp data, View data, and View Bag?**

**View data:**

ViewData is a dictionary object to pass the data from Controller to View where data is passed in the form of key-value pair. And typecasting is required to read the data in View if the data is complex and we need to ensure null check to avoid null exceptions. The scope of ViewData is similar to ViewBag and it is restricted to the current request and the value of ViewData will become null while redirecting

**View Bag:**

ViewBag is a dynamic object to pass the data from Controller to View. And, this will pass the data as a property of object ViewBag. And we have no need to typecast to read the data or for null checking. The scope of ViewBag is permitted to the current request and the value of ViewBag will become null while redirecting.

**Temp data:**

TempData is a dictionary object to pass the data from one action to other action in the same Controller or different Controllers. Usually, TempData object will be stored in a session object. Tempdata is also required to typecast and for null checking before reading data from it. TempData scope is limited to the next request and if we want Tempdata to be available even further, we should use Keep and peek.

**=> What are the types of results in MVC?**

In MVC, there are twelve types of results in where “ActionResult” class is the main class while the 11 are their sub-types:

* ViewResult
* PartialViewResult
* EmptyResult
* RedirectResult
* RedirectToRouteResult
* JsonResult
* JavaScriptResult
* ContentResult
* FileContentResult
* FileStreamResult
* FilePathResult

**=> What is the order of the filters that get executed, if multiple filters are implemented?**

The filter order would be like:

* Authorization filters
* Action filters
* Response filters
* Exception filters

**=> Can you create a web application with both webforms and MVC?**

Yes. You need to include the below MVC assembly references in the web forms application to create a hybrid application.

**=> What is the difference between authorization and authentication?**

Authentication means, Checking whether User is valid or not by accepting certain credentials like username and password.Authorization means - Checking what all rights user have, Like he have read access to so and so option, Read and write to so and so...Like that,record

**=> RenderSection RenderBody RenderPage?**

https://www.dotnettricks.com/learn/mvc/layouts-renderbody-rendersection-and-renderpage-in-aspnet-mvc

**=> RenderPartial,RenderAction, Partial and Action**

<https://www.dotnettricks.com/learn/mvc/renderpartial-vs-renderaction-vs-partial-vs-action-in-mvc-razor>

**=> What are different ways of rendering a Partial View in ASP.NET MVC?**

**Html.RenderPartial:**

This method result will be directly written to the HTTP response stream means it used the same TextWriter object as used in the current webpage/template.

This method returns void.

Simple to use and no need to create any action.

RenderPartial method is useful when the displaying data in the partial view is already in the corresponding view model.

For example: In a blog to show comments of an article, you can use RenderPartial method since an article information with comments are already populated in the view model.

*@{Html.RenderPartial("\_Comments");}*

This method is faster than Partial method since its result is directly written to the response stream which makes it fast.

**Html.RenderAction:**

This method result will be directly written to the HTTP response stream means it used the same TextWriter object as used in the current webpage/template.

For this method, we need to create a child action for the rendering the partial view.

RenderAction method is useful when the displaying data in the partial view is independent from corresponding view model. For example: In a blog to show category list on each and every page, we would like to use RenderAction method since the list of category is populated by the different model.

*@{Html.RenderAction("Category","Home");}*

This method is the best choice when you want to cache a partial view.

This method is faster than Action method since its result is directly written to the HTTP response stream which makes it fast.

**Html.Partial:**

Renders the partial view as an HTML-encoded string.   
This method result can be stored in a variable, since it returns string type value.   
Simple to use and no need to create any action.   
 Like RenderPartial method, Partial method is also useful when the displaying data in the partial view is already in the corresponding view model. For example: In a blog to show comments of an article, you can use Partial method since an article information with comments are already populated in the view model

*@Html.Partial("\_Comments")*

**Html.Action:**

This method result will be directly written to the HTTP response stream means it used the same TextWriter object as used in the current webpage/template.   
For this method, we need to create a child action for the rendering the partial view.   
RenderAction method is useful when the displaying data in the partial view is independent from corresponding view model. For example: In a blog to show category list on each and every page, we would like to use RenderAction method since the list of category is populated by the different model.

*@{Html.RenderAction("Category","Home");}*   
| This method is the best choice when you want to cache a partial view.   
 This method is faster than Action method since its result is directly written to the HTTP response stream which makes it fast.

**=> What is Remote Validation?**

Remote validation is **the process where we validate specific data posting data to a server without posting the entire form data to the server**.

**=> MVC page life cygle and working**

**=> What is cross site scripting (XSS) attacks?**

**=> What are ASP.NET MVC Filters and Attributes?**

ASP.NET MVC provides a simple way to inject your piece of code or logic either before or after an action is executed. This is achieved by decorating the controllers or actions with ASP.NET MVC attributes or custom attributes. An attribute or custom attribute implements the ASP.NET MVC filters (filter interface) and can contain your piece of code or logic. You can make your own custom filters or attributes either by implementing ASP.NET MVC filter interface or by inheriting and overriding methods of ASP.NET MVC filter attribute class if available. Typically, Filters are used to perform the following common functionalities in your ASP.NET MVC application.

1. Custom Authentication 2. Custom Authorization (User based or Role based)

3. Error handling or logging

4. User Activity Logging

5. Data Caching

6. Data Compression

**=> What are ASP.NET MVC Filters and Attributes?**

1. Authentication Filters - This filter is introduced with ASP.NET MVC5. The IAuthenticationFilter interface is used to create CustomAuthentication filter. The definition of this interface is given below

2. Authorization Filters - The ASP.NET MVC Authorize filter attribute implements the IAuthorizationFilter interface. The definition of this interface is given below

3. Action Filters - Action filters are executed before or after an action is executed. The IActionFilter interface is used to create an Action Filter which provides two methods OnActionExecuting and OnActionExecuted which will be executed before or after an action is executed respectively.

4. Result Filters - Result filters are executed before or after generating the result for an action. The Action Result type can be ViewResult, PartialViewResult, RedirectToRouteResult, RedirectResult, ContentResult, JsonResult, FileResult and EmptyResult which derives from the ActionResult class. Result filters are called after the Action filters. The IResultFilter interface is used to create a Result Filter which provides two methods OnResultExecuting and OnResultExecuted which will be executed before or after generating the result for an action respectively.

5. Exception Filters - Exception filters are executed when exception occurs during the actions execution or filters execution. The IExceptionFilter interface is used to create an Exception Filter which provides OnException method which will be executed when exception occurs during the actions execution or filters execution.

**=> When Exception filters are executed in ASP.NET MVC?**

**And What is the order of execution of filters in ASP.NET MVC?**

Exception filters are executed if there is an unhandled exception thrown during the execution of the ASP.NET MVC pipeline

All ASP.NET MVC filter are executed in an order.

The correct order of execution is given below:

1. Authentication filters

2. Authorization filters

3. Action filters

4. Result filters