Interview Questions

1. **Overview of Collection, Array List, Hash Table, Sorted List, Stack and Queue**

**Collections**  
In many applications we need to create and manage groups of related objects. There are two ways to do it, either by creating an array of objects or by creating a collection of objects.  
  
Arrays are most useful for creating a fixed number of strongly typed objects.  
  
Collections provide a more flexible way to work with groups of objects and the group of objects you work with can grow and shrink dynamically as the needs of the application changes. It also allows access to a list of items using an index. C# collections classes are defined as part of the System.Collections. So first we need to use the **System.Collections** namespace.

**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:

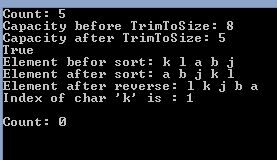
1. **Capacity:**Gets or sets the number of elements that the ArrayList can contain.
2. **Count:**Returns the number of elements present in the ArrayList.
3. **IsFixedSize:**Returns a value indicating whether the ArrayList has a fixed size.
4. **IsReadOnly:**Returns a value indicating whether the ArrayList 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:

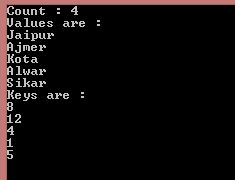
1. **Count:** Returns the number of elements present in the Hashtable.
2. **IsFixedSize:** Returns a value indicating whether the Hashtable has a fixed size.
3. **IsReadOnly:** Returns a value indicating whether the Hashtable is read-only.
4. **Item:** Gets or sets the value with the specific key.
5. **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:

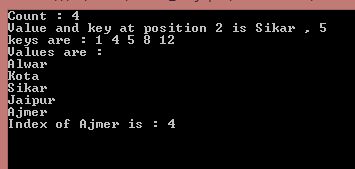
1. **Capacity:** Gets or sets the number of elements that the SortedList can contain.
2. **Count:** Returns the number of elements present in the SortedList.
3. **IsFixedSize:**Returns a value indicating whether the SortedList has a fixed size.
4. **IsReadOnly:**Returns a value indicating whether the 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:

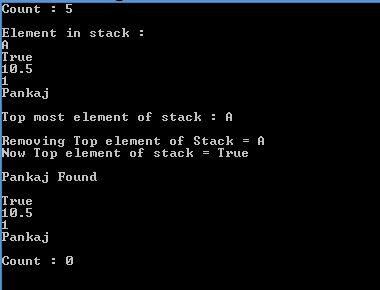
1. **Count:**Returns the number of elements in the stack.

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:

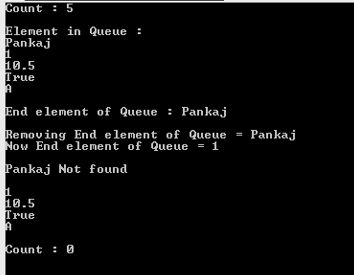
1. **Count:** Returns the number of elements in the Queue.

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**  
  


1. **The basic difference between generic and non-generic collections:**

**Non-Generic collections** - These are the collections that can hold elements of different data types. It holds all elements as object type.  
So it includes overhead of type conversions.  
  
**Generic collections -** These are the collections that can hold data of same type and we can decide what type of data that collections can hold.  
Some advantages of generic collections - Type Safe, Secure, reduced overhead of type conversions.

# Difference Between IEnumerable, ICollection and IList Interface

IEnumerable, ICollection and IList are interfaces in the .Net Framework used the most often. IEnumerable is the base of the ICollection and IList interfaces (and many other). All these interfaces provide various functionalities and are useful in various cases.  
  
**IEnumerable Interface**  
  
IEnumerable interface is used when we want to iterate among our classes using a foreach loop. The IEnumerable interface has one method, GetEnumerator, that returns an IEnumerator interface that helps us to iterate among the class using the foreach loop. The IEnumerator interface implements two the methods MoveNext() and Reset() and it also has one property called Current that returns the current element in the list.  
  
I have created a class StoreData for holding an integer type of data and this class implements the IEnumerable interface. Internally I have used a linked list for holding the data.

1. **class** StoreData : IEnumerable
2. {
3. LinkedList<**int**> items = **new** LinkedList<**int**>();
4. **public** **void** Add(**int** i)
5. {
6. items.AddLast(i);
7. }
8. **public** IEnumerator GetEnumerator()
9. {
10. **foreach** (var item **in** items)
11. {
12. yield **return** item;
13. }
14. }
15. }

In the preceding code I have created a custom storage list in which I can store integers (practically we can store any type of data depending on our requirements). We can use the preceding list as:

1. **static** **void** Main(**string**[] args)
2. {
3. StoreData list = **new** StoreData();
4. list.Add(1);
5. list.Add(2);
6. list.Add(3);
7. list.Add(4);
9. **foreach** (var item **in** list)
10. {
11. Console.WriteLine(item);
12. }
14. Console.ReadLine();
15. }

The preceding code will display all the values using a foreach loop. Instead of a foreach loop we can also use the following code:

1. IEnumerator enumerator = list.GetEnumerator();
3. **while** (enumerator.MoveNext())
4. {
5. Console.WriteLine(enumerator.Current);
6. }

Behind the scenes the foreach loop works as in the preceding code. The GetEnumerator() method is available with a list object since it implements the IEnumerable interface. Then by using the MoveNext() method and the Current property of the StoreData class we can display the data.   
  
**Note:** Every collection inside the .Net Framework implements the IEnumerable interface.  
  
**Key points of IEnumerable Interface**  
1. It provides read-only access to collections. We cannot change any item inside an IEnumerable List. It provides a sort of encapsulation in cases where we don't want our list to be changed.  
  
2. If we are dealing with some SQL queries dynamically then it also provides lazy evaluation. That means the queries will not be executed until we explicitly need them.  
  
**ICollection Interface**  
  
The ICollection interface is inherited from the IEnumerable interface which means that any class that implements the ICollection interface can also be enumerated using a foreach loop. In the IEnumerable interface we don't know how many elements there are in the collection whereas the ICollection interface gives us this extra property for getting the count of items in the collection. The ICollection interface contains the following:

* Count Property
* IsSynchronized Property
* SyncRoot Property
* CopyTo Method

The Count property is used for maintaining the count of elements in the list whereas the IsSysnchronized and SyncRoot properties help to make the collection thread-safe. The CopyTo method copies the entire collection into an array.   
  
The generic version of this interface provides Add and Remove methods also.   
  
**IList Interface**  
  
The IList interface implements both ICollection and IEnumerable interfaces. This interface allows us to add items to and remove items from the collection. It also provides support for accessing the items from the index. This interface has more power than the preceding two interfaces.  
  
The IList interface contains the following:

1. IsFixedSize Property
2. IsReadOnly Property
3. Indexer
4. Add Method
5. Clear Method
6. Contains Method
7. Indexof Method
8. Insert Method
9. Remove Method
10. RemoveAt Method

The IList interface has one indexer by which we can access any element by its position and can insert an element and remove an element at any position.

# Difference between IEnumerable and IQueryable

**IEnumerable**

1. IEnumerable exists in the System.Collections namespace.
2. IEnumerable is suitable for querying data from in-memory collections like List, Array and so on.
3. While querying data from the database, IEnumerable executes "select query" on the server-side, loads data in-memory on the client-side and then filters the data.
4. IEnumerable is beneficial for LINQ to Object and LINQ to XML queries.

**IQueryable**

1. IQueryable exists in the System.Linq Namespace.
2. IQueryable is suitable for querying data from out-memory (like remote database, service) collections.
3. While querying data from a database, IQueryable executes a "select query" on server-side with all filters.
4. IQueryable is beneficial for LINQ to SQL queries.

# Difference between Parse and ConvertTo and TryParse

* If you've got a string, and you expect it to always be an integer (say, if some web service is handing you an integer in string format), you'd use [**Int32.Parse()**](http://msdn.microsoft.com/en-us/library/system.int32.parse.aspx).
* If you're collecting input from a user, you'd generally use [**Int32.TryParse()**](http://msdn.microsoft.com/en-us/library/system.int32.tryparse.aspx), since it allows you more fine-grained control over the situation when the user enters in invalid input.
* [**Convert.ToInt32()**](http://msdn.microsoft.com/en-us/library/System.Convert.ToInt32.aspx) takes an object as its argument, and I believe it invokes Int32.TryParse() when it finds that the object taken as the argument is a string.
* Convert.ToInt32() also does not throw ArgumentNullException when it's argument is null the way Int32.Parse() does. That also means that Convert.ToInt32() is probably a wee bit slower than Int32.Parse(), though in practice, unless you're doing a very large number of iterations in a loop, you'll never notice it.

# What is Http Handler and HTTP Modules?

**HTTP Handler**

HTTP Handler is the process which runs in response to a HTTP request. So whenever user requests a file it is processed by the handler based on the extension. So, custom http handlers are created when you need to special handling based on the file name extension. Let's consider an example to create RSS for a site. So, create a handler that generates RSS-formatted XML. Now bind the .rss extension to the custom handler.

**HTTP Modules**

HTTP Modules are plugged into the life cycle of a request. So when a request is processed it is passed through all the modules in the pipeline of the request. So generally http modules are used for:

***Security***: For authenticating a request before the request is handled.

***Statistics and Logging*:** Since modules are called for every request they can be used for gathering statistics and for logging information.

***Custom header*:**  Since response can be modified, one can add custom header information to the response.