Unit – 3.2 : OOP and C#

Non-Generic Collections:

- **A *collection* —***sometimes called a container* — is simply an object that groups multiple elements into a single unit.
- A *collection* is a group of objects.
- Collections are used to store, retrieve, manipulate, and communicate aggregate data.
- The .Net Framework contains a large number of interfaces and classes that define and implement various types of collections.
- The principal benefit of collection is that they standardize the way group of objects are handled by programs.
- A collection class refers to a class that represents a collection of similar objects.
- The .Net Framework supports following general type of collections :

❖ **Non-Generic Collection classes:**

ArrayList
SortedList
Queue
Stack
Hashtable
BitArray

❖ **Generic Collection classes**

Generic List
Generic Stack
Gzeneric Queue
Generic LinkList
Generic HashSet
Dictionary

Non-Generic Collection Classes Overview

Collection Classes have the following properties

- Non- Generic Collection classes are defined as part of the System.Collections namespace.
- Most collection classes derive from the interfaces **ICollection**, **IComparer**, **IEnumerable**, **IList**, **IDictionary**, and **IDictionaryEnumerator** and their generic equivalents.
- Using generic collection classes provides increased type-safety and in some cases can provide better performance, especially when storing value types.
- The **System.Collections** namespace contains interfaces and classes that define various collections of objects, such as lists, queues, bit arrays, hash tables and dictionaries.

Class	**Description**
ArrayList	Implements the IList interface using an array whose size is dynamically increased as required.
BitArray	Manages a compact array of bit values, which are represented as Booleans, where **true** indicates that the bit is on (1) and **false** indicates the bit is off (0).

Hashtable	Represents a collection of key/value pairs that are organized based on the hash code of the key.
Queue	Represents a first-in, first-out collection of objects.
SortedList	Represents a collection of key/value pairs that are sorted by the keys and are accessible by key and by index.
Stack	Represents a simple last-in-first-out (LIFO) non-generic collection of objects.

ArrayList :

- The ArrayList class supports dynamic array which can grow or sort as needed.
- In C#, standard arrays are of fixed length, which cannot be changed during program execution.
- It means that how many elements an array will hold must be known in advance.
- In some situations in which it is not known until the runtime how large an array will be, the ArrayList is used.
- An ArrayList is a variable-length array of object references that can dynamically increase or decrease in size.
- An ArrayList is created with an initial size.
- When this size is exceeded, the collection is automatically enlarged.
- When objects are removed, the array can be shrunk.
- An ArrayList has the following constructor :

  public ArrayList( )
  public ArrayList(ICollection c)
  public ArrayList(int capacity)
- The first constructor builds an empty ArrayList with an initial capacity of zero.
- The second constructor builds an ArrayList that is initialized with the elements specified by c and has an initial capacity equal to the number of elements.
- The third constructor builds an ArrayList that has the specified initial capacity.
- The capacity is the size of array which grows automatically as elements are added to an ArrayList.

ArrayList defines several methods described as follows :

Name	**Description**
Add(value)	Used to add element in array.
CopyTo(Array A)	Used to copy the ArrayList value into specific array variable. Array variable must be one dimensional.
IndexOf(value)	Returns the index of the first occurrence of value, if not found it will return -1.
LastIndexOf(value)	Returns the last index of the last occurrence of value, if not found it will return -1.
Reverse( )	Reverses the content of collection in ArrayList.
Remove(value)	Removes the element specified by value in argument from ArrayList.
RemoveAt(int index)	Removes the value located at specified index.
Sort( )	Sorts the collection in to ascending order.

ArrayList defines several properties described as follows :

Name	Description
Capacity	Used to get or set the capacity of ArrayList items.
Count	Used to get the no. of items inside ArrayList. Counting of items.
Item	Used to get particular item by giving Index value of item.

Example : 1 : ArrayList Example

```
using System ;
using System.Collections ;
class Demo
{
    public static void Main()
    {
        ArrayList A = new ArrayList();
        A.Add("A");
        A.Add("S");
        A.Add("P");
        Console.WriteLine("Number of Elements :" + A.Count);
        A.Remove("A");
        foreach (string str in A)
            Console.WriteLine(str + "  ");
        Console.ReadKey();
    }
}
```

Output :

```
Number of Elements :3
S
P
```

Example : 2 : Constructors and Methdos

```
using System;
using System.Collections;

class Class1
{
    static void Main(string[] args)
    {
            ArrayList al1 = new ArrayList();
            ArrayList al2 = new ArrayList( 10 );

            int[] intArr = new int[10];
            ArrayList al3 = new ArrayList( intArr );

            ArrayList myArrayList = new ArrayList();
            myArrayList.Capacity = 10;

            int[] intArr2 = {2, 3, 4, 5};
            myArrayList.Add( 1 );
            myArrayList.AddRange( intArr2 );
            Console.WriteLine( myArrayList.Capacity );

            myArrayList.Remove( 1 );
            myArrayList.RemoveRange( 1, 2 );

            myArrayList.Insert( 1, 3 );
            myArrayList.InsertRange( 0, intArr );

            foreach( object elem in myArrayList )
            {
                Console.WriteLine( elem );
            }
     }
}
```

Output :

```
10
0
0
0
0
0
0
0
0
0
0
2
3
5
```

Example : 3 : Get the number of elements

```
using System;
using System.Collections;

class MainClass
{
    public static void Main()
    {
        ArrayList al = new ArrayList();
        Console.WriteLine("Initial number of elements: " + al.Count);
        Console.WriteLine("Adding 6 elements");
        // Add elements to the array list
        al.Add('C');
        al.Add('A');
        al.Add('E');
        al.Add('B');
        al.Add('D');
        al.Add('F');
        Console.WriteLine("Number of elements: " + al.Count);
    }
}
```

Output :

```
Initial number of elements: 0
Adding 6 elements
Number of elements: 6
```

Example : 4 : Remove elements from ArrayList

```
using System;
using System.Collections;

class MainClass
{
    public static void Main()
    {
        ArrayList al = new ArrayList();
        Console.WriteLine("Adding 6 elements");
        // Add elements to the array list
        al.Add('C');
        al.Add('A');
        al.Add('E');
        al.Add('B');
        al.Add('D');
        al.Add('F');
        Console.WriteLine("Removing 2 elements");
        // Remove elements from the array list.
        al.Remove('F');
        al.Remove('A');
        Console.WriteLine("Number of elements: " + al.Count);
    }
}
```

Output :

```
Adding 6 elements
Removing 2 elements
Number of elements: 4
```

For more details refer:
http://msdn.microsoft.com/en-US/library/system.collections.arraylist_methods%28v=vs.80%29.aspx

SortedList :

- SortedList creates a collection that stores a key/value pairs in sorted order, based on the value of the keys.
- A **SortedList** object internally maintains two arrays to store the elements of the list; that is, one array for the keys and another array for the associated values. Each element is a key/value pair that can be accessed as a DictionaryEntry object. A key cannot be a null reference , but a value can be.
- SortedList has two constructors as shown below:

  public SortedList ( )

  public SortedList (int capacity)
- The first constructor builds an empty collection with an initial capacity of zero.
- The second constructor builds an empty SortedList that has the initial capacity specified by "capacity".
- The capacity of SortedList grows automatically as needed when elements are added to the list.
- When the current capacity is exceeded, the capacity is increased.

SortedList defines several methods described as follows :

Name	**Description**
ContainsKey(key)	Returns true if key is in a SortedList otherwise it returns false.
ContainsValue(value)	Returns true if value is in a SortedList, otherwise it returns false.
GetKey(int index)	Returns the value of the key located at index specified in argument.
IndexOfKey(key)	Returns the index of the key specified by key, if not found then returns -1.
IndexOfValue(value)	Returns the index of the value specified by value, if not found then returns -1.
GetByIndex(int index)	Returns the value located at index specified by index.

Example : 1 : SortedList Demo.

```
using System ;
using System.Collections ;
class Demo
{
    public static void Main()
    {
        SortedList S = new SortedList();
        S.Add("c", "pointer");
        S.Add("basic", "class");
        S.Add("java", "Applet");
        S["J2EE"] = "JDBC";
        for (int i = 0; i < S.Count; i++)
            Console.WriteLine(S.GetByIndex(i));
        Console.ReadKey();
    }
}
```

Output :

```
class
pointer
JDBC
Applet
```

Example : 2 : Add value to SortedList and get contents by integer indexes

```
using System;
using System.Collections;

class MainClass
{
    public static void Main()
    {
        SortedList sl = new SortedList();
        sl.Add("a", "A");
        sl.Add("b", "B");
        sl.Add("c", "C");
        sl.Add("d", "D");
        // Display list using integer indexes.
        Console.WriteLine("Contents by integer indexes.");
        for (int i = 0; i < sl.Count; i++)
            Console.WriteLine(sl.GetByIndex(i));
    }
}
```

Output :

```
Contents by integer indexes.
A
B
C
D
```

Example : 3 : Add element to SortedList by using the indexer.

```
using System;
using System.Collections;

class MainClass
{
    public static void Main()
    {
        SortedList sl = new SortedList();
        sl.Add("a", "A");
        sl.Add("b", "B");
        sl.Add("c", "C");
        sl.Add("d", "D");
        // add by using the indexer.
        sl["e"] = "E";
        // Display list using integer indexes.
        Console.WriteLine("Contents by integer indexes.");
        for (int i = 0; i < sl.Count; i++)
            Console.WriteLine(sl.GetByIndex(i));
    }
}
```

Output :

```
Contents by integer indexes.
A
B
C
D
E
```

Example : 4 : Get value by key indexer

```
using System;
using System.Collections;

class MainClass
{
    public static void Main()
    {
        SortedList mySortedList = new SortedList();
        mySortedList.Add("NY", "New York");
        mySortedList.Add("FL", "Florida");
        mySortedList.Add("AL", "Alabama");
        mySortedList.Add("WY", "Wyoming");
        mySortedList.Add("CA", "California");
        string myState = (string)mySortedList["CA"];
        Console.WriteLine("myState = " + myState);
        // display the keys for mySortedList using the Keys property
        foreach (string myKey in mySortedList.Keys)
        {
            Console.WriteLine("myKey = " + myKey);
        }
        // display the values for mySortedList using the Values property
        foreach (string myValue in mySortedList.Values)
        {
            Console.WriteLine("myValue = " + myValue);
        }
    }
}
```

Output :

```
myState = California
myKey = AL
myKey = CA
myKey = FL
myKey = NY
myKey = WY
myValue = Alabama
myValue = California
myValue = Florida
myValue = New York
myValue = Wyoming
```

Example : 5 : Get the key list using the GetKeyList() method

```
using System;
using System.Collections;

class MainClass
{

    public static void Main()
    {
        SortedList mySortedList = new SortedList();

        mySortedList.Add("NY", "New York");
        mySortedList.Add("FL", "Florida");
        mySortedList.Add("AL", "Alabama");
        mySortedList.Add("WY", "Wyoming");
        mySortedList.Add("CA", "California");

        foreach (string myKey in mySortedList.Keys)
        {
            Console.WriteLine("myKey = " + myKey);
        }

        foreach (string myValue in mySortedList.Values)
        {
            Console.WriteLine("myValue = " + myValue);
        }
        Console.WriteLine("Getting the key list");
        IList myKeyList = mySortedList.GetKeyList();
        foreach (string myKey in myKeyList)
        {
            Console.WriteLine("myKey = " + myKey);
        }
    }
}
```

Output :

```
myKey = AL
myKey = CA
myKey = FL
myKey = NY
myKey = WY
myValue = Alabama
myValue = California
myValue = Florida
myValue = New York
myValue = Wyoming
Getting the key list
myKey = AL
myKey = CA
myKey = FL
myKey = NY
myKey = WY
```

Example : 6 : Get the index of the element with a value using the IndexOfValue() method

```
using System;
using System.Collections;

class MainClass
{

    public static void Main()
    {
        SortedList mySortedList = new SortedList();

        mySortedList.Add("NY", "New York");
        mySortedList.Add("FL", "Florida");
        mySortedList.Add("AL", "Alabama");
        mySortedList.Add("WY", "Wyoming");
        mySortedList.Add("CA", "California");

        foreach (string myKey in mySortedList.Keys)
        {
            Console.WriteLine("myKey = " + myKey);
        }

        foreach (string myValue in mySortedList.Values)
        {
            Console.WriteLine("myValue = " + myValue);
        }
        int myIndex = mySortedList.IndexOfValue("New York");
        Console.WriteLine("The index of New York is " + myIndex);
    }
}
```

Output :

```
myKey = AL
myKey = CA
myKey = FL
myKey = NY
myKey = WY
myValue = Alabama
myValue = California
myValue = Florida
myValue = New York
myValue = Wyoming
The index of New York is 3
```

Queue :

- It is also an another type of data structure which is FIFO(first in first out) type.
- The first item put in a queue is the first item retrieved.
- The collection class that supports a queue is called “Queue”.
- Queue is a dynamic collection that grows as needed to accommodate the elements it must store.
- Queue works like First In First Out method and the item added first in the Queue is first get out from Queue. We can Enqueue (add) items in Queue and we can Dequeue (remove from Queue) or we can Peek (that is get the reference of first item added in Queue) the item from Queue.

- Queue has several constructors as shown below:
  public Queue( )
  public Queue(ICollection c)
  public Queue(int capacity , float growfact)
- The third constructor always specifies a growth factor in “growfact” which must be between 1.0 and 10.0, by default it is 2.0.

Queue defines several methods described as follows :

Name	**Description**
Clear( )	Clears all elements in queue and set count to ‘0’.
Contains(value)	Returns true if value is in a Queue, otherwise it returns false.
TrimToSize( )	Set the capacity to the count.
Enqueue(value)	Adds the value into the end of the Queue.
Dequeue()	Return the object and Remove the value from the front of Queue.
Peek( )	Return the object at the front of the invoking Queue but does not remove original value.

Queue defines a property described as follows :

Name	**Description**
Count	Used to get or set the total number of elements in Queue.

Example : 1 : Queue Demo

```
using System ;
using System.Collections ;
class Demo
{
    public static void Main()
    {
        Queue Q = new Queue();
        Q.Enqueue(18);
        Q.Enqueue(73);
        Q.Enqueue(35);
        Q.Enqueue(50);
        foreach (int i in Q)
            Console.WriteLine(i);
        Q.Dequeue(); // remove element 18
        Console.WriteLine("Now top element is : " + Q.Peek());
        Console.ReadKey();
    }
}
```

Output :

```
18
73
35
50
Now top element is : 73
```

Example : 2 : Queue with Clear method

```
using System;
using System.Collections;
class Demo
{
    public static void Main()
    {
        Queue Q = new Queue();
        Q.Enqueue(18);
        Q.Enqueue(73);
        Q.Enqueue(35);
        Q.Enqueue(50);
        foreach (int i in Q)
            Console.WriteLine(i);
        Q.Clear(); // clear all elements
        Console.WriteLine("After Queue clear");
        foreach (int i in Q)
            Console.WriteLine(i);
        Console.ReadKey();
    }
}
```

Output :

```
18
73
35
50
After Queue clear
```

Example : 3 : Creating a list from a queue

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

public class MainClass
{
    public static void Main()
    {
        Queue<string> q = new Queue<string>();
        q.Enqueue("A");
        q.Enqueue("B");
        q.Enqueue("C");
        q.Enqueue("D");
        List<string> stringList = new List<string>(q);
        foreach (string str in stringList)
            Console.WriteLine(str);
    }
}
```

Output :

```
A
B
C
D
```

Stack :

- Stack follows the push-pop operations, that is we can Push Items into Stack and Pop it later.
- Stack follows the Last In First Out (LIFO) system. That is we can push the items into a stack and get it in reverse order. Stack returns the last item first.
- The real life example is Stack of Plats on the table.
- The first plat put down is the last one to be picked up.
- The Stack defines following constructors:

  public Stack( )

  public Stack(int capacity)

Stack defines several methods described as follows :

Name	**Description**
Clear( )	Clears all elements in Stack and set count to '0'.
Contains(value)	Returns true if value is in a Stack, otherwise it returns false.
Peek( )	Return the object at the front of the invoking Stack but does not remove original value.
Push(object)	Adds the element in Stack on the top of Stack.
Pop()	Returns the value from the top of Stack and remove it from Stack.

Stack defines a property described as follows :

Name	**Description**
Count	Used to get or set the total number of elements in Stack.

Example : 1 : Stack Demo

```
using System;
using System.Collections;
class Demo
{
    public static void Main()
    {
        Stack St = new Stack();
        St.Push("Rajkot");
        St.Push("Mumbai");
        St.Push("Delhi");
        St.Push("Pune");
        int cnt = St.Count;
        for (int i = 0; i < cnt; i++)
            Console.WriteLine(St.Pop());
        Console.ReadKey();
    }
}
```

Output :

```
Pune
Delhi
Mumbai
Rajkot
```

Example : 2 : Push and pop value

```
using System;
using System.Collections;

class StackDemo
{
    public static void Main()
    {
        Stack st = new Stack();

        st.Push(1);
        st.Push(2);

        foreach (int i in st)
            Console.Write(i + " ");

        st.Push(1);

        Console.Write("stack: ");
        foreach (int i in st)
            Console.Write(i + " ");

        Console.WriteLine();

        Console.Write("Pop -> ");
        int a = (int)st.Pop();
        Console.WriteLine(a);

        Console.Write("stack: ");
        foreach (int i in st)
            Console.Write(i + " ");

        Console.WriteLine();
    }
}
```

Output :

```
2 1 stack: 1 2 1
Pop -> 1
stack: 2 1
```

Example : 3 : Clear a stack

```
using System;
using System.Collections;

class MainClass
{
    static void Main(string[] args)
    {
        Stack a = new Stack(10);
        int x = 0;

        a.Push(x);
        x++;
        a.Push(x);
        foreach (int y in a)
        {
            Console.WriteLine(y);
        }
        a.Pop();
        a.Clear();
    }
}
```

Output :

```
1
0
```

Example : 4 : Pop and Peek

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

public class Tester
{
    static void Main()
    {
        Stack<Int32> intStack = new Stack<Int32>();
        for (int i = 0; i < 8; i++)
        {
            intStack.Push(i * 5);
        }
        PrintValues(intStack);
        Console.WriteLine("\n(Pop)\t{0}", intStack.Pop());
        PrintValues(intStack);
        Console.WriteLine("\n(Pop)\t{0}", intStack.Pop());
        PrintValues(intStack);
        Console.WriteLine("\n(Peek) \t{0}", intStack.Peek());
        PrintValues(intStack);
        int[] targetArray = new int[12];
        for (int i = 0; i < targetArray.Length; i++)
        {
            targetArray[i] = i * 100 + 100;
        }
        PrintValues(targetArray);
        intStack.CopyTo(targetArray, 6);
        PrintValues(targetArray);
    }

    public static void PrintValues(IEnumerable<Int32> myCollection)
    {
        IEnumerator<Int32> enumerator = myCollection.GetEnumerator();
        while (enumerator.MoveNext())
            Console.Write("{0} ", enumerator.Current);
    }
}
```

Output :

```
35 30 25 20 15 10 5 0
(Pop)   35
30 25 20 15 10 5 0
(Pop)   30
25 20 15 10 5 0
(Peek)  25
25 20 15 10 5 0 100 200 300 400 500 600 700 800 900 1000 1100 1200 100 200 300
400 500 600 25 20 15 10 5 0
```

Hashtable :

- HashTable stores a Key Value pair type collection of data. We can retrieve items from hashTable to provide the key. Both key and value are Objects.
- HashTable stores information using a mechanism called “hashing”.
- In “hashing”, the informational content of a key is used to determine a unique value, called “hash code”.
- The “hash code” is used then as index at which the data associated with the key is stored in the table.
- HashTable defines the following constructor :

    public Hashtable( )

    public Hashtable(int capacity)

- The first form constructs a default HashTable.
- The second form initializes the capacity of HashTable to capacity.

The methods that defines by HashTable is shown below :

Name	**Description**
Add( )	Used To add a pair of value in HashTable
Remove( )	Removes the specified key and corresponding value in HashTable.
ContainsKey(object o)	Check if a specified key exist or not in HashTable.
ContainsValue(object o)	Check the specified Value exist in HashTable
Equals(HashTable ht)	It returns true if both objects are same otherwise it returns false.

The HashTable supports for following properties.

Name	**Description**
Count	Used To get the no. of items inside Hashtable. Counting of items.
Keys	Gives collection of all the keyvalues given for HashTable items.
Values	Gives collection of all the values given for HashTable Items.

Example : 1 : HashTable Demo

```
using System;
using System.Collections;
class Demo
{
    public static void Main()
    {
        Hashtable Ht = new Hashtable();
        Ht.Add("R", "Rajkot");
        Ht.Add("M", "Mumbai");
        Ht.Add("D", "Delhi");
        Ht.Remove("M");

        ICollection c = Ht.Keys;
        foreach (string s in c)
            Console.WriteLine(s + "  " + Ht[s]);

        Console.ReadKey();
    }
}
```

Output :

```
R  Rajkot
D  Delhi
```

Example : 2 : Add elements to the table and Use the keys to obtain the values

```
using System;
using System.Collections;

class HashtableDemo
{
    public static void Main()
    {
        Hashtable ht = new Hashtable();

        ht.Add("a", "A");
        ht.Add("b", "B");
        ht.Add("c", "C");
        ht.Add("e", "E");

        // Get a collection of the keys.
        ICollection c = ht.Keys;

        foreach (string str in c)
            Console.WriteLine(str + ": " + ht[str]);
    }
}
```

Output :

```
a: A
b: B
c: C
e: E
```

Example : 3 : Add key-value pair to Hashtable by using the indexer

```
using System;
using System.Collections;

class HashtableDemo
{
    public static void Main()
    {
        Hashtable ht = new Hashtable();

        ht.Add("a", "A");
        ht.Add("b", "B");
        ht.Add("c", "C");
        ht.Add("e", "E");

        ht["f"] = "F";

        // Get a collection of the keys.
        ICollection c = ht.Keys;

        foreach (string str in c)
            Console.WriteLine(str + ": " + ht[str]);
    }
}
```

Output :

```
a: A
b: B
c: C
e: E
f: F
```

Example : 4 : Use foreach statement to loop through all keys in a hashtable

```
using System;
using System.Collections;

class MainClass
{
    public static void Main()
    {
        Hashtable hash = new Hashtable();
        hash.Add("A", "1");
        hash.Add("B", "2");
        hash.Add("C", "3");
        hash.Add("D", "4");
        hash.Add("E", "5");

        foreach (string firstName in hash.Keys)
        {
            Console.WriteLine("{0} {1}", firstName, hash[firstName]);
        }
    }
}
```

Output :

```
A 1
B 2
C 3
D 4
E 5
```

Example : 5 : Clear all key/value pairs in a Hashtable

```
using System;
using System.Collections;

class MainClass
{
    static void Main(string[] args)
    {
        Hashtable a = new Hashtable(10);

        a.Add(100, "Arrays");
        a.Add(200, "Classes");

        a.Clear();
    }
}
```

Output :

Example : 6 : Remove key/value pairs from Hashtable

```
using System;
using System.Collections;

class MainClass
{
    static void Main(string[] args)
    {
        Hashtable a = new Hashtable(10);

        a.Add(100, "A");
        a.Add(200, "C");

        a.Remove(100);
        a.Remove(200);

    }
}
```

Output :

BitArray :

- The BitArray class supports a collection of bits.
- It stores bits rather than objects.
- Each element of bits becomes a bit in the collection.
- So, each bit in the collection corresponds to an element of bits.
- BitArray defines following constructors :

  public BitArray(type[ ] bits)

  public BitArray(int size)

  public BitArray(int size, bool b)

- The first constructor defines the each element of array bits of specified type become elements of BitArray.
- The second constructor defines BitArray of specific size.
- The third constructor creates a BitArray with specified size and all elements are initialized to Boolean value “b”.

The BitArray supports following methods :

Name	**Description**
And(BitArray b)	ANDs the bits of the invoking object with specified by b and return a BitArray that contains the result.
Or(BitArray b)	ORs the bits of the invoking object with specified by b and return a BitArray that contains the result.
Not( )	Performs a bitwise, logical NOT on the invoking collection and returns a BitArray that contains the result.
Xor(BitArray b)	XORs the bits of the invoking object with specified by b and return a BitArray that contains the result.
Get(int idx)	Returns the value of the bit at the index specified by “idx”.
Set(int idx, bool v)	Sets the bit at the index specified by “idx” to “b”.
SetAll(bool v)	Sets all bits to “v”.

The BitArray supports for following properties.

Name	Description
Count	Used To get the no. of items inside BitArray. Counting of items.

Example : 1 : BitArray Demo

```
using System;
using System.Collections;
class Demo
{
    public static void Main()
    {
        BitArray b1 = new BitArray(4, true);
        BitArray b2 = new BitArray(4, true);
        BitArray b3 = new BitArray(4);
        b3 = b1.And(b2);
        for (int i = 0; i < b3.Count; i++)
            Console.WriteLine(b3[i]);
        Console.ReadKey();
    }
}
```

Output :

```
True
True
True
True
```

Example : 2 : BitArray: Not()

```
using System;
using System.Collections;

class MainClass
{
    public static void Main()
    {
        BitArray ba = new BitArray(8);
        byte[] b = { 67 };
        BitArray ba2 = new BitArray(b);

        Console.WriteLine("Original contents of ba:");

        for (int i = 0; i < ba.Count; i++)
            Console.Write("{0, -6} ", ba[i]);

        ba = ba.Not();

        Console.WriteLine("\nContents of ba after Not:");
        for (int i = 0; i < ba.Count; i++)
            Console.Write("{0, -6} ", ba[i]);
    }
}
```

Output :

```
Original contents of ba:
False  False  False  False  False  False  False  False
Contents of ba after Not:
True   True   True   True   True   True   True   True
```

Example : 3 : BitArray: Xor()

```
using System;
using System.Collections;

class MainClass
{
    public static void Main()
    {
        BitArray ba = new BitArray(8);
        byte[] b = { 67 };
        BitArray ba2 = new BitArray(b);

        Console.WriteLine("Original contents of ba:");

        for (int i = 0; i < ba.Count; i++)
            Console.Write("{0, -6} ", ba[i]);

        ba = ba.Not();

        BitArray ba3 = ba.Xor(ba2);

        Console.WriteLine("\nResult of ba XOR ba2:");
        for (int i = 0; i < ba3.Count; i++)
            Console.Write("{0, -6} ", ba3[i]);
    }
}
```

Output :

```
Original contents of ba:
False  False  False  False  False  False  False  False
Result of ba XOR ba2:
False  False  True   True   True   True   False  True
```

Example : 4 : Set the four elements of the BitArray and display the elements of the BitArray

```
using System;
using System.Collections;

class MainClass
{
    public static void Main()
    {
        BitArray myBitArray = new BitArray(4);

        Console.WriteLine("myBitArray.Length = " + myBitArray.Length);
        myBitArray[0] = false;
        myBitArray[1] = true;
        myBitArray[2] = true;
        myBitArray[3] = false;

        for (int i = 0; i < myBitArray.Count; i++)
        {
            Console.WriteLine("myBitArray[" + i + "] = " + myBitArray[i]);
        }
    }
}
```

Output :

```
myBitArray.Length = 4
myBitArray[0] = False
```

```
myBitArray[1] = True
myBitArray[2] = True
myBitArray[3] = False
```

Example : 5 : Use copy constructor in BitArray

```
using System;
using System.Collections;

class MainClass
{
    public static void DisplayBitArray(string arrayListName, BitArray myBitArray)
    {
        for (int i = 0; i < myBitArray.Count; i++)
        {
            Console.WriteLine(arrayListName + "[" + i + "] = " + myBitArray[i]);
        }
    }

    public static void Main()
    {
        BitArray myBitArray = new BitArray(4);
        myBitArray[0] = false;
        myBitArray[1] = true;
        myBitArray[2] = true;
        myBitArray[3] = false;
        DisplayBitArray("myBitArray", myBitArray);

        BitArray anotherBitArray = new BitArray(myBitArray);
        DisplayBitArray("anotherBitArray", myBitArray);

    }

}
```

Output :

```
myBitArray[0] = False
myBitArray[1] = True
myBitArray[2] = True
myBitArray[3] = False
anotherBitArray[0] = False
anotherBitArray[1] = True
anotherBitArray[2] = True
anotherBitArray[3] = False
```

Example : 6 : Use the Or() method to perform an OR operation on the elements in BitArray and another BitArray

```
using System;
using System.Collections;

class MainClass
{
    public static void DisplayBitArray(string arrayListName, BitArray myBitArray)
    {
        for (int i = 0; i < myBitArray.Count; i++)
        {
            Console.WriteLine(arrayListName + "[" + i + "] = " + myBitArray[i]);
        }
    }

    public static void Main()
    {
        BitArray myBitArray = new BitArray(4);
        myBitArray[0] = false;
        myBitArray[1] = true;
        myBitArray[2] = true;
        myBitArray[3] = false;
        DisplayBitArray("myBitArray", myBitArray);

        BitArray anotherBitArray = new BitArray(myBitArray);
        DisplayBitArray("anotherBitArray", myBitArray);

        myBitArray.Not();
        DisplayBitArray("myBitArray", myBitArray);

        myBitArray.Or(anotherBitArray);
        DisplayBitArray("myBitArray", myBitArray);


    }

}
```

Output :

```
myBitArray[0] = False
myBitArray[1] = True
myBitArray[2] = True
myBitArray[3] = False
anotherBitArray[0] = False
anotherBitArray[1] = True
anotherBitArray[2] = True
anotherBitArray[3] = False
myBitArray[0] = True
myBitArray[1] = False
myBitArray[2] = False
myBitArray[3] = True
myBitArray[0] = True
myBitArray[1] = True
myBitArray[2] = True
myBitArray[3] = True
```

Interfaces

Interface	**Description**
ICollection	Defines size, enumerators, and synchronization methods for all nongeneric collections.
IComparer	Exposes a method that compares two objects.
IDictionary	Represents a nongeneric collection of key/value pairs.
IDictionaryEnumerator	Enumerates the elements of a nongeneric dictionary.
IEnumerable	Exposes the enumerator, which supports a simple iteration over a non-generic collection.
IEnumerator	Supports a simple iteration over a nongeneric collection.

IEqualityComparer	Defines methods to support the comparison of objects for equality.
IHashCodeProvider	Supplies a hash code for an object, using a custom hash function.
IList	Represents a non-generic collection of objects that can be individually accessed by index. (Implemented by ArrayList collection)

Generic Collections :

- Generic collections are declared in System.Collections.Generics namespace.
- It is collection of Class and Interface.
- The generic collection is same as non generic collection with the exception that a generic collection is type safe.
- Therefore if we want to store a data that is mix type, then non generic collection is used.
- Generic Collection Classes
  - List<T>
  - Stack<T>
  - Queue<T>
  - HashSet<T>
  - LinkedList<T>
  - Dictionary<T>

Generic List<T> :

- The list <T> class implements a generic dynamic array and it is similar to the non generic array list class.
- List <T> implements ICollection, ICollection<T>, IList, IList<T> interfaces.
- Generic List defines following constructor :

  public List( )

  public List(int capacity)
- The first constructor builds an empty List with default initial capacity.
- The second constructor builds an array list that has the specified initial capacity.
- The capacity is the underlying array that is used to store the elements.
- The capacity grows automatically as elements are added to a List<T>.

The generic List supports following methods :

Name	**Description**
AndRange(ICollection C)	Adds the element at the end of the list.
IndexOf(<T> value)	Returns the first occurrence of value in the collection, returns -1 if value not found.
LastIndexOf(<T> value)	Returns the last occurrence of value in the collection, returns -1 if value not found.
Reverse( )	Reverses the content of the invoking collection.
Sort( )	Sorts the collection into ascending order.

Example : 1 : Generic List<T> Demo

```
using System;
using System.Collections.Generic;
class Demo
{
    public static void Main()
    {
        List<char> L1 = new List<char>();
        L1.Add('x');
        L1.Add('y');
        L1.Add('z');
        Console.WriteLine("Element in List : " + L1.Count);
        for (int i = 0; i < L1.Count; i++)
            Console.WriteLine(L1[i]);
        Console.ReadKey();
    }
}
```

Output :

```
Element in List : 3
x
y
z
```

Generic Stack<T> :

- Stack<T> is the generic equivalent of the non-generic Stack class.
- Stack<T> supports a first-in, last-out stack.
- It implements the ICollection interface.
- Stack<T> is a dynamic collection that grows as needed to accommodate the elements it must store.
- It defines following constructors :

  public Stack( )

  public Stack(int capacity)

Stack<T> supports following methods :

Name	**Description**
Peek( )	Returns the element on the top of stack, but does not remove it.
Pop( )	Returns the element on the top of stack, removing it in the process.
Push(<T> value)	Pushes value onto the stack.
ToArray( )	Returns an array that contains copies of the elements of the invoking stack.
Sum( )	Returns the sum of all elements of stack.

The Stack<T> supports for following properties.

Name	Description
Count	Used To get the no. of items inside Stack. Counting of items.

Example : 1 : Generic Stack<T> Demo

```
using System;
using System.Collections.Generic;
class Demo
{
    public static void Main()
    {
        Stack<int> S = new Stack<int>();
        S.Push(10);
        S.Push(20);
        S.Push(30);
        Console.WriteLine("No. of elements :" + S.Count);
        int cnt = S.Count;
        for (int i = 0; i < cnt; i++)
        {
            Console.WriteLine(S.Pop());
        }
        Console.ReadKey();
    }
}
```

Output :

```
No. of elements :3
30
20
10
```

Generic Queue<T> :

- Queue<T> is the generic equivalent of the non-generic Queue class.
- It supports a first-in, first-out list.
- Queue<T> implements the ICollection interface.
- Queue<T> is a dynamic collection that grows as needed to accommodate the elements it must store.
- It defines following constructor :

  public Queue( )

  public Queue(int capacity)

Queue<T> supports following methods :

Name	**Description**
Peek( )	Returns the object at the front of the invoking Queue, but does not remove it.
Enqueue(<T> value)	Adds value to the end of the queue.
Dequeue(<T> value)	Returns the object at the front of the invoking queue. The object is removed in the process.
ToArray( )	Returns an array that contains copies of the elements of the invoking queue.

Queue<T> supports following properties :

Name	**Description**
Count	Used To get the no. of items inside Queue. Counting of items.

Example : 1 : Generic Queue<T> Demo

```
using System;
using System.Collections.Generic;
class Demo
{
    public static void Main()
    {
        Queue<double> Q = new Queue<double>();
        Q.Enqueue(10.6);
        Q.Enqueue(20.0);
        Q.Enqueue(30.14);
        Console.WriteLine("Queue contents :");
        while (Q.Count > 0)
            Console.WriteLine(Q.Dequeue());
        Console.ReadKey();
    }
}
```

Output :

```
Queue contents :
10.6
20
30.14
```

Generic HashSet<T> :

- HashSet<T> is a new collection added to the .Net Framework.
- It supports a collection that implements a set.
- It uses a hash table for storage.
- HashSet<T> implements the ICollection<T> interface.
- **HashSet<T> implements a set in which all elements are unique. In other words, duplicates are not allowed.**
- HashSet<T> defines a full complement of set operations, such as intersection, union and difference.
- The commonly used constructors defined by HashSet<T> are :

  public HashSet( )

The list of methods supported by HashSet<T> :

Name	**Description**
ExceptWidth(<T> set2)	Removes the elements in set2 form the invoking set.
IntersectWith(<T> set2)	Removes from the invoking set those elements not common to both the invoking set and set2.
IsSubsetOf(<T> set2)	Returns true if the invoking set is a subset of set2.
IsSupersetOf(<T> set2)	Returns true if the invoking set is a superset of set2.
SetEquals(<T> set2)	Returns true if the invoking set is equivalent to set2.
UnionWith(<T> set2)	Adds the elements from set2 to the invoking set. Duplicates are not included. Creates the union of two sets.

Example : 1 : Generic Queue<T> Demo

```
using System;
using System.Collections.Generic;
class Demo
{
    public static void Main()
    {
        HashSet<char> H1 = new HashSet<char>();
        HashSet<char> H2 = new HashSet<char>();
        H1.Add('A');
        H1.Add('B');
        H1.Add('C');
        Console.WriteLine("Elements of H1 :");
        foreach (char ch in H1)
            Console.WriteLine(ch);
        H1.UnionWith(H2);
        Console.WriteLine("Elements of H1 After Union:");
```

```
            foreach (char ch in H1)
                Console.WriteLine(ch);
            Console.ReadKey();
        }
    }
```

Output :

```
Elements of H1 :
A
B
C
Elements of H1 After Union:
A
B
C
```

Generic LinkedList<T> :

- Represents a doubly linked list.
- LinkedList<T> is a general-purpose linked list. It supports enumerators and implements the ICollection interface, consistent with other collection classes in the .NET Framework.
- LinkedList<T> provides separate nodes of type LinkedListNode<T>, so insertion and removal are O(1) operations.
- Each node in a LinkedList<T> object is of the type LinkedListNode<T>. Because the LinkedList<T> is doubly linked, each node points forward to the Next node and backward to the Previous node.
- Lists that contain reference types perform better when a node and its value are created at the same time. LinkedList<T> accepts null as a valid Value property for reference types and allows duplicate values.
- If the LinkedList<T> is empty, the First and Last properties contain null.
- It defines following constructor :

  LinkedList<T>()

  Initializes a new instance of the LinkedList<T> class that is empty.

  LinkedList<T>(IEnumerable<T>)

  Initializes a new instance of the LinkedList<T> class that contains elements copied from the specified IEnumerable and has sufficient capacity to accommodate the number of elements copied.

LinkedList<T> supports following methods :

Name	**Description**
AddFirst(T)	Adds a new node containing the specified value at the start of the LinkedList<T>.
AddLast(T)	Adds a new node containing the specified value at the end of the LinkedList<T>.
Clear	Removes all nodes from the LinkedList<T>.
Contains	Determines whether a value is in the LinkedList<T>.
CopyTo	Copies the entire LinkedList<T> to a compatible one-dimensional Array, starting at the specified index of the target array.
Find	Finds the first node that contains the specified value.
FindLast	Finds the last node that contains the specified value.
Remove(T)	Removes the first occurrence of the specified value from the LinkedList<T>.

Remove(LinkedListNode<T>)	Removes the specified node from the LinkedList<T>.
RemoveFirst	Removes the node at the start of the LinkedList<T>.
RemoveLast	Removes the node at the end of the LinkedList<T>.

LinkedList<T> supports following properties :

Name	**Description**
Count	Gets the number of nodes actually contained in the LinkedList<T>.
First	Gets the first node of the LinkedList<T>.
Last	Gets the last node of the LinkedList<T>.

Example : 1 : Generic LinkedList<T> Demo

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

public class MainClass
{
    public static void Main()
    {
        LinkedList<int> list = new LinkedList<int>();

        list.AddFirst(10);
        list.AddLast(15);
        list.AddLast(3);
        list.AddLast(99);
        list.AddBefore(list.Last, 25);

        LinkedListNode<int> node = list.First;
        while (node != null)
        {
            Console.WriteLine(node.Value);
            node = node.Next;
        }
    }
}
```

Output :

```
10
15
3
25
99
```

Example : 2 : Display the linked list by using a foreach loop

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

class MainClass
{
    public static void Main()
    {
        // Create an linked list.
        LinkedList<char> ll = new LinkedList<char>();

        Console.WriteLine("Adding 5 elements.");
        // Add elements to the linked list
        ll.AddFirst('A');
        ll.AddFirst('B');
        ll.AddFirst('C');
        ll.AddFirst('D');
        ll.AddFirst('E');

        Console.WriteLine("Number of elements: " + ll.Count);
        Console.Write("Display contents with foreach loop: ");
        foreach (char ch in ll)
            Console.Write(ch + " ");

        Console.WriteLine("\n");

    }
}
```

Output :

```
Adding 5 elements.
Number of elements: 5
Display contents with foreach loop: E D C B A
```

Example : 3 : Remove elements from the linked list

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

class MainClass
{
    public static void Main()
    {
        // Create an linked list.
        LinkedList<char> ll = new LinkedList<char>();

        Console.WriteLine("Adding 5 elements.");
        // Add elements to the linked list
        ll.AddFirst('A');
        ll.AddFirst('B');
        ll.AddFirst('C');
        ll.AddFirst('D');
        ll.AddFirst('E');
        Console.WriteLine("Removing 2 elements.");
        ll.Remove('C');
        ll.Remove('A');
        Console.WriteLine("Number of elements: " + ll.Count);
    }
}
```

Output :

```
Adding 5 elements.
Removing 2 elements.
Number of elements: 3
```

Generic Dictionary<TKey, TValue> :

- Represents a collection of keys and values.
- The Dictionary<TKey, TValue> generic class provides a mapping from a set of keys to a set of values.

- Each addition to the dictionary consists of a value and its associated key. Retrieving a value by using its key is very fast, because the Dictionary<TKey, TValue> class is implemented as a hash table.
- It defines following constructor :

  Dictionary<TKey, TValue>()

  Initializes a new instance of the Dictionary<TKey, TValue> class that is empty, has the default initial capacity, and uses the default equality comparer for the key type.

  Dictionary<TKey, TValue>(IDictionary<TKey, TValue>)

  Initializes a new instance of the Dictionary<TKey, TValue> class that contains elements copied from the specified IDictionary<TKey, TValue> and uses the default equality comparer for the key type.

  Dictionary<TKey, TValue>(Int32)

  Initializes a new instance of the Dictionary<TKey, TValue> class that is empty, has the specified initial capacity, and uses the default equality comparer for the key type.

Dictionary< TKey, TValue > supports following methods :

Name	**Description**
Add	Adds the specified key and value to the dictionary.
Clear	Removes all keys and values from the Dictionary<TKey, TValue>.
Remove	Removes the value with the specified key from the Dictionary<TKey, TValue>.
TryGetValue	Gets the value associated with the specified key.

Dictionary< TKey, TValue > supports following properties :

Name	**Description**
Count	Gets the number of key/value pairs contained in the Dictionary<TKey, TValue>.
Item	Gets or sets the value associated with the specified key.
Keys	Gets a collection containing the keys in the Dictionary<TKey, TValue>.
Values	Gets a collection containing the values in the Dictionary<TKey, TValue>.

Example : 1 : Add to dictionary

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

public class Tester
{
    static void Main()
    {
        Dictionary<string, string> Dictionary = new Dictionary<string, string>();
        Dictionary.Add("1", "J");
        Dictionary.Add("2", "S");
        Dictionary.Add("3", "D");
        Dictionary.Add("4", "A");
        Console.WriteLine(Dictionary["1"]);
    }
}
```

Output :

```
J
```

Example : 2 : Use the keys to obtain the values from a generic Dictionary

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

class MainClass
{
    public static void Main()
    {
        Dictionary<string, double> dict = new Dictionary<string, double>();

        // Add elements to the collection.
        dict.Add("A", 7);
        dict.Add("B", 5);
        dict.Add("C", 4);
        dict.Add("D", 9);

        // Get a collection of the keys (names).
        ICollection<string> c = dict.Keys;

        foreach (string str in c)
            Console.WriteLine("{0}, Salary: {1:C}", str, dict[str]);
    }
}
```

Output :

```
A, Salary: $7.00
B, Salary: $5.00
C, Salary: $4.00
D, Salary: $9.00
```

Generics

- The term generics means parameterized types.
- Parameterized types are important because they enable you to create classes, interfaces, methods, and delegates in which the type of data operated on is specified as a parameter.
- Using generics, it is possible to create a single class, for example, that automatically works with different types of data.
- A class, interface, method, or delegate that operates on a parameterized type is called generic, as in generic class or generic method.
- Use generic types to maximize code reuse, type safety, and performance.
- The most common use of generics is to create collection classes.
- The .NET Framework class library contains several new generic collection classes in the System.Collections.Generic namespace. These should be used whenever possible in place of classes such as ArrayList in the System.Collections namespace.
- You can create your own generic interfaces, classes, methods, events and delegates.
- Generic classes may be constrained to enable access to methods on particular data types.
- Information on the types used in a generic data type may be obtained at run-time by means of reflection.

Generic Type Parameters

- In a generic type or method definition, a type parameters is a placeholder for a specific type that a client specifies when they instantiate a variable of the generic type.
- A generic class, such as GenericList<T>, cannot be used as-is because it is not really a type; it is more like a blueprint for a type.
- To use GenericList<T>, client code must declare and instantiate a constructed type by specifying a type argument inside the angle brackets.
- The type argument for this particular class can be any type recognized by the compiler.
- Any number of constructed type instances can be created, each one using a different type argument, as follows:

```
GenericList<float> list1 = new GenericList<float>();
GenericList<ExampleClass> list2 = new GenericList<ExampleClass>();
GenericList<ExampleStruct> list3 = new GenericList<ExampleStruct>();
```

Example : 1 : Life without generics

```
using System;

class NonGen
{
    object ob;
    public NonGen(object o)
    {
        ob = o;
    }
    public object getob()
    {
        return ob;
    }
    public void showType()
    {
        Console.WriteLine("Type of ob is " + ob.GetType());
    }
}
class MainClass
{
    public static void Main()
    {
        NonGen iOb = new NonGen(102);
        iOb.showType();

        int v = (int)iOb.getob();
        Console.WriteLine("value: " + v);

        Console.WriteLine();

        NonGen strOb = new NonGen("Non-Generics Test");
        strOb.showType();

        String str = (string)strOb.getob();
        Console.WriteLine("value: " + str);
    }
}
```

Output :

```
Type of ob is System.Int32
value: 102

Type of ob is System.String
value: Non-Generics Test
```

Generics Types

Generic Classes
Generic Interface
Generic Methods
Generic Delegates

Generic Classes

- 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 where operations such as adding and removing items from the collection are performed in much the same way regardless of the type of data being stored.

Example : 1 : A simple generic class

```
using System;

class Gen<T>
{
    T ob;
    public Gen(T o)
    {
        ob = o;
    }
    public T getob()
    {
        return ob;
    }
    public void showType()
    {
        Console.WriteLine("Type of T is " + typeof(T));
    }
}
class MainClass
{
    public static void Main()
    {
        Gen<int> iOb = new Gen<int>(102);
        iOb.showType();

        int v = iOb.getob();
        Console.WriteLine("value: " + v);

        Console.WriteLine();

        Gen<string> strOb = new Gen<string>("Generics add power.");
        strOb.showType();
        string str = strOb.getob();
        Console.WriteLine("value: " + str);
    }
}
```

Output :

```
Type of T is System.Int32
value: 102

Type of T is System.String
value: Generics add power.
```

Example : 2 : A simple generic class with two type parameters: T and V

```
using System;

class TwoGen<T, V>
{
    T ob1;
    V ob2;
    public TwoGen(T o1, V o2)
    {
        ob1 = o1;
        ob2 = o2;
    }
    public void showTypes()
    {
        Console.WriteLine("Type of T is " + typeof(T));
        Console.WriteLine("Type of V is " + typeof(V));
    }
    public T getT()
    {
        return ob1;
    }
    public V getV()
    {
        return ob2;
    }
}
class MainClass
{
    public static void Main()
    {

        TwoGen<int, string> tgObj = new TwoGen<int, string>(1, "A");

        tgObj.showTypes();

        int v = tgObj.getT();
        Console.WriteLine("value: " + v);

        string str = tgObj.getV();
        Console.WriteLine("value: " + str);
    }
}
```

Output :

```
Type of T is System.Int32
Type of V is System.String
value: 1
value: A
```

Generic Interface

- It is often useful to define interfaces either for generic collection classes, or for the generic classes that represent items in the collection.
- With generic classes it is preferable to use generic interfaces, such as IComparable<T> rather than IComparable, in order to avoid boxing and unboxing operations on value types.
- When an interface is specified as a constraint on a type parameter, only types that implement the interface can be used.

Example : 1 : Generic Interface

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

interface GenericInterface<T>
{
    T getValue(T tValue);
}
class MyClass<T> : GenericInterface<T>
{
    public T getValue(T tValue)
    {
        return tValue;
    }
}
class MainClass
{
    static void Main()
    {
        MyClass<int> intObject = new MyClass<int>();
        MyClass<string> stringObject = new MyClass<string>();

        Console.WriteLine("{0}", intObject.getValue(5));
        Console.WriteLine("{0}", stringObject.getValue("Hi there."));
    }
}
```

Output :

```
5
Hi there.
```

Generic Methods

- A generic method is a method that is declared with type parameters, as follows:

```
static void Swap<T>(ref T lhs, ref T rhs)
{
    T temp;
    temp = lhs;
    lhs = rhs;
    rhs = temp;
}
public static void TestSwap()
{
    int a = 1;
    int b = 2;

    Swap<int>(ref a, ref b);
    System.Console.WriteLine(a + " " + b);
}
```

- You can also omit the type argument and the compiler will infer it. The following call to Swap is equivalent to the previous call:

```
Swap(ref a, ref b);
```

- The same rules for type inference apply to static methods as well as instance methods.
- **The compiler is able to infer the type parameters based on the method arguments you pass in; it cannot infer the type parameters solely from a constraint or return value.**
- Therefore **type inference does not work with methods that have no parameters.**
- Type inference takes place at compile time before the compiler attempts to resolve any overloaded method signatures.
- The compiler applies type inference logic to all generic methods that share the same name.
- In the overload resolution step, the compiler includes only those generic methods on which type inference succeeded.

Example : 1 : Generic Method

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

class GenericMethodDemo
{
    static List<T> MakeList<T>(T first, T second)
    {
        List<T> list = new List<T>();
        list.Add(first);
        list.Add(second);
        return list;
    }
    static void Main()
    {
        List<string> list = MakeList<string>("Line 1", "Line 2");
        foreach (string x in list)
        {
            Console.WriteLine(x);
        }
    }
}
```

Output :

```
Line 1
Line 2
```

Example : 2 : Generic Method

```
using System;

class ArrayUtils
{
    public static bool isBigger<T>(T[] src, T[] target)
    {
        // See if target array is big enough.
        if (target.Length < src.Length + 1)
            return false;
        return true;
    }
}
class GenMethDemo
{
    public static void Main()
    {
        int[] nums = { 1, 2, 3 };
        int[] nums2 = new int[4];

        // Operate on an int array.
        bool b = ArrayUtils.isBigger(nums, nums2);

        Console.WriteLine(b);

        string[] strs = { "Generics", "are", "powerful." };
        string[] strs2 = new string[4];

        b = ArrayUtils.isBigger(strs, strs2);
        Console.WriteLine(b);
    }
}
```

Output :

```
True
True
```

Example : 3 : Generic Method with two different data type parameters

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

class GenericMethodDemo
{
    static void Test<T,V>(T lhs, V rhs)
    {
        Console.WriteLine(lhs);
        Console.WriteLine(rhs);
    }
    public static void Main()
    {
        int a = 1;
        float b = 2.1f;

        Test<int,float>(a, b);
    }

}
```

Output:

```
1
2.1
```

Generic Delegates

- A delegate can define its own type parameters. Code that references the generic delegate can specify the type argument to create a closed constructed type, just like when instantiating a generic class or calling a generic method, as shown in the following example:

```
public delegate void Del<T>(T item);
public static void Notify(int i) { }

Del<int> m1 = new Del<int>(Notify);
```

Example : 1 : A generic delegate.

```
using System;

delegate T SomeOp<T>(T v);

class MainClass
{
    static int sum(int v)
    {
        return v;
    }
    static string reflect(string str)
    {
        return str;
    }
    public static void Main()
    {
        SomeOp<int> intDel = sum;
        Console.WriteLine(intDel(3));

        SomeOp<string> strDel = reflect;
        Console.WriteLine(strDel("Hello"));
    }
}
```

Output :

```
3
Hello
```