



**Collections**

Harsha Vardhan  
 UI Expert  
 .NET Expert  
 Module Lead  
 Corporate Instructor

**Introducing Collections** **Next: List Collection**

**What**

- Collections are the standard-way to store and manipulate group of elements (primitive values or objects).
- Collections are internally objects of specific 'collection classes' such as List, Dictionary, SortedList etc.

**How**

'List' collection

```
List<type> referenceVariable = new List<type>();
```

Collection	
[0]	value0
[1]	value1
[2]	value2
[3]	value3
[4]	value4
[5]	value5
[6]	value6

**Understanding Collections** **Next: 'List' Collection**

**💡**

- Collections can store unlimited elements.
- You can add, remove elements at any time.
- You need not specify the size (no. of elements) while creating collection.
- You can search, sort, copy collections using various built-in methods.

**Agenda**

- 1 **Understanding Collections**  
What is collection and how is it better than array
- 2 **List**  
Understanding List collection and its methods
- 3 **Dictionary**  
Understanding Dictionary collection and its methods
- 4 **SortedList**  
Understanding SortedList collection and its methods
- 5 **Hashtable**  
Understanding Hashtable collection and its methods

**'List' Collection** **Next: Features of 'List' class**

**'list' collection**

```
List<type> referenceVariable = new List<type>();
```

Collection	
[0]	value0
[1]	value1
[2]	value2
[3]	value3
[4]	value4
[5]	value5
[6]	value6

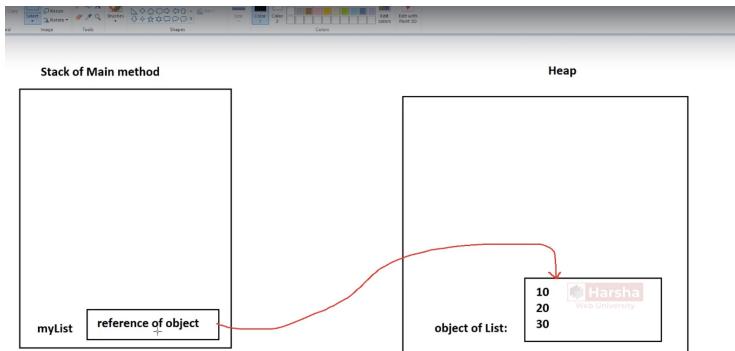
```

using System;
using System.Collections.Generic;

namespace ListExample
{
    class Program
    {
        static void Main()
        {
            //create reference variable for List class
            List<int> myList;

            //create object of List class
            new List<int>() { 10, 20, 30 };
        }
    }
}

```



### Features of 'List' class

**💡**

- It is dynamically sized. You can add, remove elements at any time.
- It allows duplicate values.
- It is index-based. You need to access elements by using zero-based index.
- It is not sorted by default. The elements are stored in the same order, how they are initialized.
- It uses arrays internally; that means, recreates array when the element is added / removed.
- The 'Capacity' property holds the number of elements that can be stored in the internal array of the List. If you add more elements, the internal array will resized to the 'Count' of elements.

### Properties & Methods of 'List' class

**Properties**

- Count
- Capacity
- Add(T)
- AddRange(IEnumerable<T>)
- Insert(int, T)
- InsertRange(int, IEnumerable<T>)
- Remove(T)
- RemoveAt(int)
- RemoveRange(int, int)
- RemoveAll(Predicate<T>)
- Clear()
- IndexOf(T)
- BinarySearch(T)

**Methods**

- Sort()
- Reverse()
- ToArray()
- ForEach(Action<T>)
- Exists(Predicate<T>)
- Find(Predicate<T>)
- FindIndex(Predicate<T>)
- FindLast(Predicate<T>)
- FindLastIndex(Predicate<T>)
- FindAll(Predicate<T>)
- ConvertAll(T)

Next: IndexOf()

**Properties & Methods of 'List' class**

**Properties**

- > Count
- > Capacity

**Methods**

- > Add(T)
- > AddRange(IEnumerable<T>)
- > Insert(int, T)
- > InsertRange(int, IEnumerable<T>)
- > Remove(T)
- > RemoveAt(int)
- > RemoveRange(int, int)
- > RemoveAll(Predicate<T>)
- > Clear()
- > IndexOf(T)
- > BinarySearch(T)
- > Contains(T)
- > Sort()
- > Reverse()
- > ToArray()
- > ForEach(Action<T>)
- > Exists(Predicate<T>)
- > Find(Predicate<T>)
- > FindIndex(Predicate<T>)
- > FindLast(Predicate<T>)
- > FindLastIndex(Predicate<T>)
- > FindAll(Predicate<T>)
- > ConvertAll(T)

**Next: IndexOf()**

**List - Add( ) method**

**Add()** > This method adds a new element to the collection.

Collection [0] value0 [1] value1 [2] value2 [3] value3 [4] value4

Collection [0] value0 [1] value1 [2] value2 [3] value3 [4] value4 [5] newValue

**List - Add() method**

**void List.Add(T newValue)**

**Next: AddRange()**

**List - AddRange( ) method**

**AddRange()** > This method adds a new set of elements to the collection.

Collection [0] value0 [1] value1 [2] value2

Collection [0] value0 [1] value1 [2] value2 [3] 10 [4] 20

**List - AddRange() method**

**void List.AddRange(IEnumerable<T> newValue)**

**List - AddRange() - Example**

**List.AddRange(new List<int>() { newValue1, newValue2 })**

**Next: Insert()**

**Properties & Methods of 'List' class**

**Properties**

- > Count
- > Capacity

**Methods**

- > Add(T)
- > AddRange(IEnumerable<T>)
- > **Insert(int, T)**
- > InsertRange(int, IEnumerable<T>)
- > Remove(T)
- > RemoveAt(int)
- > RemoveRange(int, int)
- > RemoveAll(Predicate<T>)
- > Clear()
- > IndexOf(T)
- > BinarySearch(T)
- > Contains(T)
- > Sort()
- > Reverse()
- > ToArray()
- > ForEach(Action<T>)
- > Exists(Predicate<T>)
- > Find(Predicate<T>)
- > FindIndex(Predicate<T>)
- > FindLast(Predicate<T>)
- > FindLastIndex(Predicate<T>)
- > FindAll(Predicate<T>)
- > ConvertAll(T)

**Next: IndexOf()**

**List - Insert( ) method** **Next: InsertRange()**

This method adds a new element to the collection at the specified index.

Collection	[0]	[1]	[2]	[3]	[4]
	value0	value1	value2	value3	value4

Collection	[0]	[1]	[2]	[3]	[4]	[5]
	value0	value1	value2	newValue	value3	value4

**List - Insert() method**

```
void List.Insert(int index, T newValue)
```

**List - Insert() - Example**

```
List.Insert(3, newValue)
```

**List - InsertRange( ) method** **Next: Remove()**

This method adds a new set of elements to the collection at the specified index.

Collection	[0]	[1]	[2]	[3]
	value0	value1	value2	value3

Collection	[0]	[1]	[2]	[3]	[4]
	value0	value1	newValue1	newValue2	value3

**List - InsertRange() method**

```
void List.InsertRange(int index, IEnumerable<T> newValue)
```

**List - InsertRange() - Example**

```
List.InsertRange(2, new List<int> { newValue1, newValue2 })
```

**Properties & Methods of 'List' class** **Next: IndexOf()**

**Properties**

- Count
- Capacity

**Methods**

- Add(T)
- AddRange(IEnumerable<T>)
- Insert(int, T)
- InsertRange(int, IEnumerable<T>)
- Remove(T)
- RemoveAt(int)
- RemoveRange(int, int)
- RemoveAll(Predicate<T>)
- Clear()
- IndexOf(T)
- BinarySearch(T)
- Contains(T)
- Sort()
- Reverse()
- ToArray()
- ForEach(Action<T>)
- Exists(Predicate<T>)
- Find(Predicate<T>)
- FindIndex(Predicate<T>)
- FindLast(Predicate<T>)
- FindLastIndex(Predicate<T>)
- FindAll(Predicate<T>)
- ConvertAll(T)

**List - Remove( ) method** **Next: RemoveAt()**

This method removes the specified element from the collection.

Collection	[0]	[1]	[2]	[3]	[4]
	value0	value1	value2	value3	value4

Collection	[0]	[1]	[2]	[3]
	value0	value1	value3	value4

**List - Remove() method**

```
void List.Remove(T newValue)
```

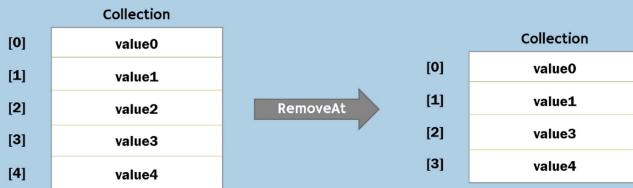
**List - Remove() - Example**

```
List.Remove(value2)
```

## removeRange, RemoveAll, Clear

Next: RemoveRange()

- This method removes an element from the collection at the specified index.



List - RemoveAt() method

```
void List.RemoveAt(int index)
```

List - RemoveAt() - Example

```
List.RemoveAt(2)
```

## removeRange, RemoveAll, Clear

Next: RemoveAll()

- This method removes specified count of elements starting from the specified startIndex.



List - RemoveRange() method

```
void List.RemoveRange(int index, int count)
```

List - RemoveRange() - Example

```
List.RemoveRange(1, 2)
```

## List - RemoveAll() method

Next: Clear()

- This method removes all the elements that are matching with the given condition.
- You can write your condition in the lambda expression of Predicate type.



List - RemoveAll() method

```
void List.RemoveAll(value => condition)
```

List - RemoveAll() - Example

```
List.RemoveAll( n => n >= 30 )
```

## List - Clear() method

Next: IndexOf()

- This methods removes all elements in the collection.



List - Clear() method

```
void List.Clear()
```

List - Clear() - Example

```
List.Clear()
```

Properties & Methods of 'List' class	
<b>Properties</b>	> Count > Capacity
<b>Methods</b>	> Add(T) > AddRange(IEnumerable<T>) > Insert(int, T) > InsertRange(int, IEnumerable<T>) > Remove(T) > RemoveAt(int) > RemoveRange(int, int) > RemoveAll(Predicate<T>) > Clear() > IndexOf(T) <span style="border: 1px solid red; padding: 2px;">Index Of</span> > BinarySearch(T) > Contains(T)
	> Sort() > Reverse() > ToArray() > ForEach(Action<T>) > Exists(Predicate<T>) > Find(Predicate<T>) > FindIndex(Predicate<T>) > FindLast(Predicate<T>) <span style="background-color: #e0e0e0;">Harsha</span> > FindLastIndex(Predicate<T>) > FindAll(Predicate<T>) > ConvertAll(T)

Next: [IndexOf\(\)](#)

This method searches the collection for the given value.

- If the value is found, it returns its index.
- If the value is not found, it returns -1.

List - IndexOf( ) method	
<b>Collection</b>	
[0]	10
[1]	20
[2]	30
[3]	40
<b>List - IndexOf() method</b>	<b>List - IndexOf() - Example</b>
<code>int List.IndexOf(T value, int startIndex)</code>	<code>List.IndexOf(20)</code>

💡

The "IndexOf" method performs linear search. That means it searches all the elements of the collection, until the search value is found. When the search value is found in the collection, it stops searching and returns its index.

The linear search has good performance, if the collection is small. But if the collection is larger, Binary search is recommended to improve the performance.

**Parameters**

- value:** This parameter represents the actual value that is to be searched.
- startIndex:** This parameter represents the start index, from where the search should be started.

This method searches the collection for the given value.

List - BinarySearch() method	
<b>Collection</b>	
[0]	10
[1]	20
[2]	30
[3]	40
[4]	50
[5]	60
<code>int List.BinarySearch(T value)</code>	

**contains** `List.BinarySearch( ) method` [Next: Contains\(\)](#)



- > The "Binary Search" requires a collection, which is already sorted.
- > On unsorted collections, binary search is not possible.
- > It directly goes to the middle of the collection (collection size / 2), and checks that item is less than / greater than the search value.
- > If that item is greater than the search value, it searches only in the first half of the collection.
- > If that item is less than the search value, it searches only in the second half of the array.

  
**Parameters**

- > **value:** This parameter represents the actual value that is to be searched.

**contains** `list - BinarySearch( ) method` [Next: Contains\(\)](#)

- > This method searches the collection for the given value.
- > If the value is found, it returns its index.
- > If the value is not found, it returns -1.

Collection	
[0]	10
[1]	20
[2]	30
[3]	40
[4]	50
[5]	60

`BinarySearch(20)` → 1  
`BinarySearch(70)` → -1

**List - BinarySearch() method**  
`int List.BinarySearch(T value)`

**List - BinarySearch() - Example**  
`List.BinarySearch(20)`

**contains** `list - Contains( ) method` [Next: Sort\(\)](#)

- > This method searches the specified element and returns 'true', if it is found; but returns 'false', if it is not found.

Collection	
[0]	10
[1]	20
[2]	30
[3]	40
[4]	50

`Contains(30)` → true  
`Contains(60)` → false

**List - Contains() method**  
`bool List.Contains(T value)`

**Contains() - Example**  
`List.Contains(30)`

**contains** `List - Sort( ) method` [Next: Reverse\(\)](#)

- > This method sorts the collection in ascending order.

Collection	
[0]	100
[1]	950
[2]	345
[3]	778
[4]	20

`Sort` →

Collection	
[0]	20
[1]	100
[2]	345
[3]	778
[4]	950

**List - Sort() method**  
`void List.Sort()`

**List - Sort() - Example**  
`List.Sort()`

**List - Reverse( ) method** → Next: [ToArray\(\)](#)

> This method reverses the collection.

Collection	
[0]	100
[1]	950
[2]	345
[3]	778
[4]	20

Reverse

Collection	
[0]	20
[1]	778
[2]	345
[3]	950
[4]	100

**List - Reverse() method**

```
void List.Reverse()
```

**List - Reverse() - Example**

```
List.Reverse()
```

**List - ToArray( ) method** → Next: [ForEach\(\)](#)

> This method converts the collection into an array with same elements.

Collection	
[0]	10
[1]	20
[2]	30
[3]	40
[4]	50

ToArray

array	
[0]	10
[1]	20
[2]	30
[3]	40
[4]	50

**List - ToArray() method**

```
T[] List.ToArray()
```

**List - ToArray() - Example**

```
List.ToArray()
```

**List - ForEach( ) method** → Next: [Exists\(\)](#)

> This method executes the lambda expression once per each element.

Collection	
[0]	10
[1]	20
[2]	30
[3]	40

**List - ForEach() method**

```
void List.ForEach( Action<T> )
```

**List - ForEach() - Example**

```
List.ForEach( n => { Console.WriteLine(n); } )
```

Action is a pre-defined delegate that returns void and takes an argument. It accept lamda expression or annynomous function.

**List - ForEach( ) method** → Next: [Exists\(\)](#)

> This method executes the lambda expression once per each element.

Collection	
[0]	10
[1]	20
[2]	30
[3]	40

10 → Execute lambda expression  
 20 → Execute lambda expression  
 30 → Execute lambda expression  
 40 → Execute lambda expression

**List - ForEach() method**

```
void List.ForEach( Action<T> )
```

**List - ForEach() - Example**

```
List.ForEach( n => { Console.WriteLine(n); } )
```

## Exist, Find, FindIndex, FindLast, FindLastIndex, FindAll

all these methods are provided by List Class.

### List - Exists() method

FindLast, FindLastIndex, FindAll      Next: Find()

This method executes the lambda expression once per each element.  
It returns true, if at least one element matches with the given condition; but returns false, if no element matches with the given condition.

Collection

[0]	10
[1]	20
[2]	30

List - Exists() method      List - Exists() - Example

`bool List.Exists( Predicate<T> )`      `List.Exists( n => n > 15 )`

Harsha  
Wadu University

FindLast, FindLastIndex, FindAll      Next: Find()

This method executes the lambda expression once per each element.  
It returns true, if at least one element matches with the given condition; but returns false, if no element matches with the given condition.

Collection

[0]	10
[1]	20
[2]	30

Execute lambda expression      Execute lambda expression      Execute lambda expression

true

List - Exists() method      List - Exists() - Example

`bool List.Exists( Predicate<T> )`      `List.Exists( n => n > 15 )`

Harsha  
Wadu University

Find, Exists, Find, FindIndex, FindLast, FindLastIndex, FindAll      Next: FindIndex()

This method executes the lambda expression once per each element.  
It returns the first matching element, if at least one element matches with the given condition; but returns the default value, if no element matches with the given condition.

Collection

[0]	10
[1]	20
[2]	30

List - Find() method      List - Find() - Example

`T List.Find( Predicate<T> )`      `List.Find( n => n > 15 )`

Harsha  
Wadu University

**List - FindAll() method** Next: **FindIndex()**

- This method executes the lambda expression once per each element.
- It returns the first matching element, if at least one element matches with the given condition; but returns the default value, if no element matches with the given condition.

Collection			
[0]	10	10	Execute lambda expression
[1]	20	20	Execute lambda expression
[2]	30	30	Execute lambda expression

20

**List - Find() method** **List - Find() - Example**

**T List.Find( Predicate<T> )**

**List.Find( n => n > 15 )**

**List - FindIndex() method** Next: **FindLast()**

- This method executes the lambda expression once per each element.
- It returns index of the first matching element, if at least one element matches with the given condition; but returns -1, if no element matches with the given condition.

Collection			
[0]	10	10	Execute lambda expression
[1]	20	20	Execute lambda expression
[2]	30	30	Execute lambda expression

1

**List - FindIndex() method** **List - FindIndex() - Example**

**int List.FindIndex( Predicate<T> )**

**List.FindIndex( n => n > 15 )**

**List - FindLast() method** Next: **FindLastIndex()**

- This method executes the lambda expression once per each element.
- It returns index of the last matching element, if at least one element matches with the given condition; but returns -1, if no element matches with the given condition.

Collection			
[0]	10	10	Execute lambda expression
[1]	20	20	Execute lambda expression
[2]	30	30	Execute lambda expression

30

**List - FindLast() method** **List - FindLast() - Example**

**T List.FindLast( Predicate<T> )**

**List.FindLast( n => n > 15 )**

**List - FindLastIndex() method** Next: **FindAll()**

- This method executes the lambda expression once per each element.
- It returns index of the last matching element, if at least one element matches with the given condition; but returns -1, if no element matches with the given condition.

Collection			
[0]	10	10	Execute lambda expression
[1]	20	20	Execute lambda expression
[2]	30	30	Execute lambda expression

2

**List - FindLastIndex() method** **List - FindLastIndex() - Example**

**int List.FindLastIndex( Predicate<T> )**

**List.FindLastIndex( n => n > 15 )**

indLast, FindLastIndex, FindAll ] method Next: ConvertAll()

- This method executes the lambda expression once per each element.
- It returns all matching elements as a collection, if there are one or more matching elements; but returns empty collection if no matching elements.

Collection

[0]	10
[1]	20
[2]	30

Execute lambda expression

Execute lambda expression

Execute lambda expression

[0]	20
[1]	30

List - FindAll() method List - FindAll() - Example

List<T> List.FindAll( Predicate<T> )

List.FindAll( n => n > 15 )

24:36

List - ConvertAll( ) method Next: Dictionary

- This method executes the lambda expression once per each element.
- It adds each returned element into a new collection and returns the same at last; thus it converts all elements from the input collection as output collection.

Input Collection

[0]	10
[1]	20
[2]	30

Execute lambda expression

Execute lambda expression

Execute lambda expression

Output Collection

[0]	10
[1]	20
[2]	30

List - ConvertAll() method List - ConvertAll() - Example

List<TOutput> List.ConvertAll( Converter<TInput, TOutput> )

List.ConvertAll( n => Convert.ToDouble(n) )

9:07

'Dictionary' Collection Next: Features of 'Dictionary' class

- Dictionary collection contains a group of elements of key/value pairs.
- Full Path: System.Collections.Generic.Dictionary
- The "Dictionary" class is a generic class; so you need to specify data type of the key and data type of the value while creating object.
- You can set / get the value based on the key.
- The key can't be null or duplicate.

Dictionary Collection

[key 0]	value0
[key 1]	value1
[key 2]	value2
[key 3]	value3
[key 4]	value4
[key 5]	value5
[key 6]	value6

'Dictionary' collection

Dictionary< TKey, TValue > referenceVariable = new Dictionary< TKey, TValue >();

Group of key-value pair is called as Dictionary.

Features of 'Dictionary' class Next: Methods of Dictionary class

- It is dynamically sized. You can add, remove elements (key/value pairs) at any time.
- Key can't be null or duplicate; but value can be null or duplicate.
- It is not index-based. You need to access elements by using key.
- It is not sorted by default. The elements are stored in the same order, how they are initialized.

💡

<

**Properties & Methods of 'Dictionary' class** | Harsh

**Properties**

- > **Count** : Returns count of elements.
- > **[TKey]** : Returns value based on specified key.
- > **Keys** : Returns a collection of key (without values).
- > **Values** : Returns a collection of values (without keys).

**Methods**

- > **void Add(TKey, TValue)** : Adds an element (key/value pair).
- > **bool Remove(TKey)** : Removes an element based on specified key.
- > **bool ContainsKey(TKey)** : Determines whether the specified key exists.
- > **bool ContainsValue(TValue)** : Determines whether the specified value exists.
- > **void Clear()** : Removes all elements.

'SortedList' Collection | Harsh

Next: Features of 'SortedList' class

> SortedList collection contains a group of elements of key/value pairs.

> Full Path: System.Collections.Generic.SortedList

**SortedList Collection**

[key 0]	value0
[key 1]	value1
[key 2]	value2
[key 3]	value3
[key 4]	value4
[key 5]	value5
[key 6]	value6

'SortedList' collection

```
SortedList< TKey, TValue> referenceVariable = new SortedList< TKey, TValue>();
```

SortedList and Dictionary are similar.

When you will use SortedList over Dictionary?

In case of large amount of data, ex: millions of records, use sortedList as it keeps all the record in sorted order whereas dictionary keeps data as they are stored.

Dictionary performs Linear Search.

Searching process is slower but adding process is faster in Dictionary.  
Searching process is faster but adding process is slower in SortedList.

'SortedList' Collection | Harsh

Next: Features of 'SortedList' class

> SortedList collection contains a group of elements of key/value pairs.

> Full Path: System.Collections.Generic.SortedList

< The "SortedList" class is a generic class; so you need to specify data type of the key and data type of the value while creating object.

> You can set / get the value based on the key.

> The key can't be null or duplicate.

**SortedList Collection**

[key 0]	value0
[key 1]	value1
[key 2]	value2
[key 3]	value3
[key 4]	value4
[key 5]	value5
[key 6]	value6

'SortedList' collection

```
SortedList< TKey, TValue> referenceVariable = new SortedList< TKey, TValue>();
```



- > It is dynamically sized. You can add, remove elements (key/value pairs) at any time.
- > Key can't be null or duplicate; but value can be null or duplicate.
- > It is not index-based. You need to access elements by using key.
- > It is sorted by default. The elements are stored in the sorted ascending order, according to the key.
- > Each operation of adding element, removing element or any other operation might be slower than Dictionary, because internally it resorts the data based on key.

**Properties**

- > Count : Returns count of elements.
- > [TKey] : Returns value based on specified key.
- > Keys : Returns a collection of key (without values).
- > Values : Returns a collection of values (without keys).

**Methods**

- > void Add(TKey, TValue) : Adds an element (key/value pair).
- > bool Remove(TKey) : Removes an element based on specified key.
- > bool ContainsKey(TKey) : Determines whether the specified key exists.
- > bool ContainsValue(TValue) : Determines whether the specified value exists.
- > int IndexOfKey(TKey) : Returns index of the specified key.
- > int IndexOfValue(TValue) : Returns index of the specified value.
- > void Clear() : Removes all elements.

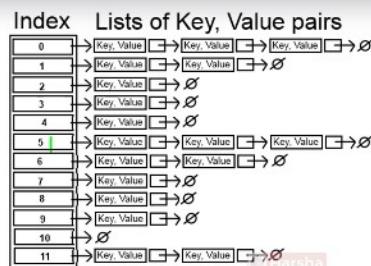
> Hashtable collection contains a group of elements of key/value pairs stored at respective indexes.

> Full Path:  
System.Collections.Hashtable

> Process of adding an element:

- I. Generate index based on the key. Ex: index = hash code % size of dictionary

2. Add the element (key and value) next to the linked list at the generated index.



**'Hashtable' collection**

```
Hashtable referenceVariable = new Hashtable();
```

In case of string, system will generate the hash code, in case of number, same number will be used as a hash code.

HashTable is by default linked-list.

HashTable is not a generic class; it is represent in System.Collection namespace. you can store any data type for key and any data value for value.

198. Hashtable

Features of 'Hashtable' class

Next: Methods of Hashtable class

 It is dynamically sized. You can add, remove elements (key/value pairs) at any time.

Key can't be null or duplicate; but value can be null or duplicate.

The "Hashtable" class is not a generic class.

You can set / get the value based on the key.

It is not index-based. You need to access elements by using key.



Properties & Methods of 'Hashtable' class

Properties

- Count : Returns count of elements.
- [TKey] : Returns value based on specified key.
- Keys : Returns a collection of key (without values).
- Values : Returns a collection of values (without keys).

Methods

- void Add(object key, object value) : Adds an element (key/value pair).
- void Remove(object key) : Removes an element based on specified key.
- bool ContainsKey(object key) : Determines whether the specified key exists.
- bool ContainsValue(object value) : Determines whether the specified value exists.
- void Clear() : Removes all elements.

## HashSet

Try to imagine a hash-table without keys, that becomes hashset.

Duplicate values are not allowed in hash-set.  
But duplicate index can be generated for two value.

In hash-table, the index will be generated based on the hash code of the key right and you have unique key here.

But in hash-set, the index will be calculated based on the hash code of the value instead of the hashtable.

'HashSet' Collection

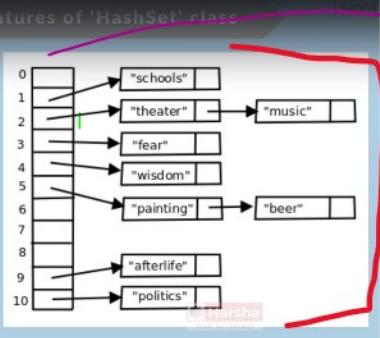
Next: Features of 'HashSet' class

HashSet collection contains a group of elements of unique values stored at respective indexes.

Full Path:  
System.Collections.Generic.HashSet

Process of adding an element:

- Generate index based on the value. Ex: index = hash code % count
- Add the element (value) next to the linked list at the generated index.


 A diagram illustrating the internal structure of a HashSet. It shows an array of 11 slots, indexed from 0 to 10. Each slot contains a reference to a node in a linked list. The nodes store key-value pairs. For example, index 0 points to a node with "schools" as the key and "1" as the value. Index 1 points to a node with "theater" and "2". Index 2 points to a node with "fear" and "3". Index 3 points to a node with "wisdom" and "4". Index 4 points to a node with "painting" and "5". Index 5 points to a node with "beer" and "6". Index 6 points to a node with "afterlife" and "7". Index 7 points to a node with "politics" and "8". Index 8 points to a node with "politics" and "9". Index 9 points to a node with "politics" and "10". Index 10 is empty. Handwritten annotations with red arrows point from the text "array" to the array structure and from "Linked list" to the linked list structure of the nodes.

'HashSet' collection

```
HashSet<T> referenceVariable = new HashSet<T>();
```

always capacity will be maintained as prime number. The default capacity is 17. Capacity will automatically be incremented if you keep adding elements.

Here the capacity means the size of the internal array .





here the hashCode of the string values will be automatically generated based on the predefined implementation of the getHashCode method of System.Object class.

using System;

```
Class Object{
    getHashCode();
}

class String extends Object{

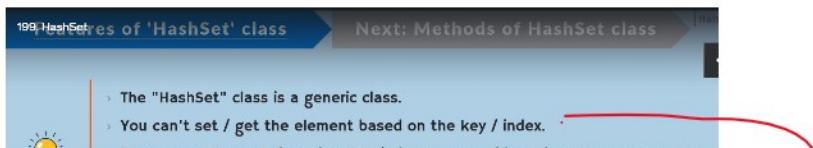
//overridden
public override getHashCode() // calculates the hashcode based on the
characters.
}
```

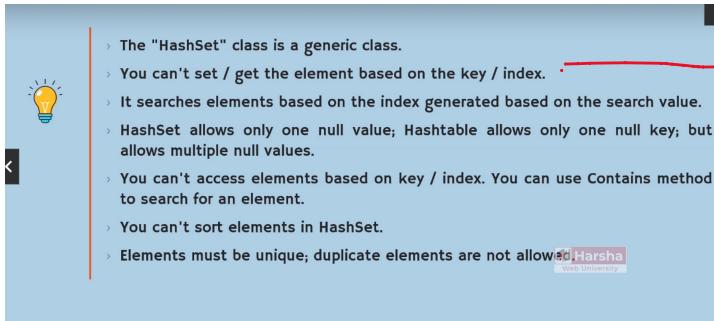
In case of numerical type, the same number will be treated as hash-code.

> Generate index based on the value. Ex:  
 $\text{index} = \text{hash code \% count}$

suppose you are looking for the particular value called **painting** so based on the value called painting an index will be generated at that moment and it goes to the particular index directly instead of searching all the previous elements in the form of linear search so that is the reason it's extremely fast while retrieving that means while checking whether the element exists or not that is the benefit of hash set see in case if you store the same values in the form of a list suppose you are searching for a particular element you need to perform linear search that means you have to set all the elements top to bottom but it's not the case in case of hash set so checking and determining whether the particular element exists or not is extremely faster in case of hash set so sometimes rather than storing the elements in the form of a collection you want to make sure whether this particular element or object already exists in the collection or not if that is your main intention then you prefer using hash set over the list to class.

So, for retrieving purpose, hash-set is more useful.





because it takes value and calculates index based on hash code

**Properties & Methods of 'HashSet' class** → **Next: ArrayList**

Properties	Methods
> <b>Count</b> : Returns count of elements.	
	> <b>void Add(T value)</b> : Adds an element (key/value pair). > <b>void Remove(T value)</b> : Removes an element based on specified key. > <b>void RemoveWhere(Predicate&lt;T&gt; predicate)</b> : Remove elements that matches with condition. > <b>bool Contains (T value)</b> : Determines whether the specified value exists. > <b>void Clear()</b> : Removes all elements. > <b>void UnionWith(IEnumerable&lt;T&gt; other)</b> : Unions the hashset and specified collection. > <b>void IntersectWith(IEnumerable&lt;T&gt; other)</b> : Intersects the hashset and specified collection.

so in real time applications when you don't want to store the keys but mainly you want the collection to determine whether this particular value exists or not  
for that cases we can store those set of elements in the **hash set collection** but alternatively you want to get the exact element  
based on the key for example employee id unique key in that case prefer using **hash table or dictionary**

but you want to **store a plain set of elements with or without unique key** in that case prefer using **list class** but in case if you want to store key and

in case if you are performing adding and removing process less number of times but mainly you are trying to retrieve elements by using for each loop or based on the key in that case prefer using **sorted list**

ArrayList

'ArrayList' Collection

Next: Features of 'ArrayList' class

| Harsha

- > ArrayList collection contains a group of elements of any type.
- > Full Path: System.Collections.ArrayList
- > The "ArrayList" class is a not a generic class; so you need not specify data type value while creating object.

ArrayList Collection	
[0]	value0
[1]	value1
[2]	value2
[3]	value3
[4]	value4
[5]	value5
[6]	value6

'ArrayList' collection

```
ArrayList referenceVariable = new ArrayList();
```

| Harsha

So when will you use the array list over list whenever you want to store a plain set of elements of a particular type that means when all the values are of same type then you will use the list as usual as we have demonstrated in the previous lectures but whenever you don't know that means you are not sure what type of values that you want to store in the collection

then you will use the array list for example you want to store some numbers string date time and objects of various custom classes so like this you want to store various types of elements in the same collection then you will use the error list over list sometimes you are not sure what type of value does a method returns for example the written type of the method is object so it may written a number or string or object of different custom classes so you are not sure about what type of value does that method

returns that method may be developed by other developers or sometimes by third party developers in that cases you will be using error list because in case of error list the elements can be of any type so you will not specify any data type while creating object of error list class that means error list is not a generic class so you will not specify any data type while creating object for the same since it is not a generic class it is present under the namespace called system.collections but not system.collections.generic

*Whenever we are not sure what to store or we want to store various types of elements in the same collection, then use ArrayList*

Features of 'ArrayList' class

Next: Methods of ArrayList class

| Harsha



- > It is dynamically sized. You can add, remove elements at any time.
- > It is index-based. You need to access elements by using the zero-based index.
- > It is not sorted by default. The elements are stored in the same order, how they are initialized.
- > You don't specify data type of elements for ArrayList. So you can store any type of elements in ArrayList.
- > Each element is treated as 'System.Object' type while adding, searching and retrieving elements.

| Harsha  
Web University

Properties and methods are most similar to List class.

**Properties & Methods of 'ArrayList' class**      Next: Stack

<b>Properties</b> <ul style="list-style-type: none"> <li>&gt; Count</li> <li>&gt; Capacity</li> </ul>	<ul style="list-style-type: none"> <li>&gt; Add(object)</li> <li>&gt; AddRange(ICollection)</li> <li>&gt; Insert(int, object)</li> <li>&gt; InsertRange(int, ICollection)</li> <li>&gt; Remove(object)</li> <li>&gt; RemoveAt(int)</li> <li>&gt; RemoveRange(int, int)</li> <li>&gt; Clear()</li> <li>&gt; IndexOf(object)</li> <li>&gt; BinarySearch(object)</li> <li>&gt; Contains(object)</li> <li>&gt; Sort()</li> <li>&gt; Reverse()</li> <li>&gt; ToArray()</li> </ul>
---	--

Harsha  
Web University

**'Stack' Collection**      Next: Features of 'Stack' class

<ul style="list-style-type: none"> <li>&gt; Stack collection contains a group of elements based on LIFO (Last-In-First-Out) based collection.</li> <li>&gt; Full Path: System.Collections.Generic.Stack</li> <li>&gt; It stores the elements in bottom-to-up approach.           <ul style="list-style-type: none"> <li>&gt; Firstly added item at bottom.</li> <li>&gt; Lastly added item at top.</li> </ul> </li> </ul>	
---	--

'Stack' collection

```
Stack<T> referenceVariable = new Stack<T>();
```

Harsha  
Web University

**Features of 'Stack' class**      Next: Methods of Stack class

<ul style="list-style-type: none"> <li>&gt; It is based on LIFO (Last-In-First-Out).</li> <li>&gt; The "Stack" class is a generic class; so you need to specify data type of elements while creating object.</li> <li>&gt; You can't access elements based on index.</li> <li>&gt; You can add elements using 'Push' method; remove elements using 'Pop' method; access last element using 'Peek' method.</li> <li>&gt; It is not index-based. You need to access elements by either using pop, peek or foreach loop.</li> </ul>	
--	--

**Properties & Methods of 'Stack' class**      Next: Queue

<b>Properties</b> <ul style="list-style-type: none"> <li>&gt; Count : Returns count of elements.</li> </ul>	
<b>Methods</b> <ul style="list-style-type: none"> <li>&gt; void Push(T) : Adds an element at the top of the stack.</li> <li>&gt; T Pop() : Removes and returns the element at top of stack.</li> <li>&gt; T Peek() : Returns the element at the top of the stack.</li> <li>&gt; bool Contains(T) : Determines whether the specified element exists.</li> <li>&gt; T[] ToArray() : Converts the stack as an array.</li> <li>&gt; void Clear() : Removes all elements.</li> </ul>	

Harsha  
Web University

**'Queue' Collection** ➔ **Next: Features of 'Queue' class**

- Queue collection contains a group of elements based on FIFO (First-In-First-Out) based collection.
- Full Path: System.Collections.Generic.Queue
- It stores the elements in front-to-rear approach.
- Firstly added item at front.
- Lastly added item at rear.

```
Queue<T> referenceVariable = new Queue<T>();
```

**Features of 'Queue' class** ➔ **Next: Methods of Queue class**

- It is based on FIFO (First-In-First-Out).
- The "Queue" class is a generic class.
- You can't access elements based on index.
- You can add elements using 'Enqueue' method; remove elements using 'Dequeue' method; access last element using 'Peek' method.
- It is not index-based. You need to access elements by either using Dequeue, Peek or foreach loop.

**Properties & Methods of 'Queue' class** ➔ **Next: Collection of Objects**

<b>Properties</b>	<ul style="list-style-type: none"> <li>&gt; <b>Count</b> : Returns count of elements.</li> </ul>
<b>Methods</b>	<ul style="list-style-type: none"> <li>&gt; <b>void Enqueue(T)</b> : Adds an element at the top of the queue.</li> <li>&gt; <b>T Dequeue()</b> : Removes and returns the element at top of queue.</li> <li>&gt; <b>T Peek()</b> : Returns the element at the top of the queue.</li> <li>&gt; <b>bool Contains(T)</b> : Determines whether the specified element exists.</li> <li>&gt; <b>T[] ToArray()</b> : Converts the queue as an array.</li> <li>&gt; <b>void Clear()</b> : Removes all elements.</li> </ul>

**Collection of Objects** ➔ **Next: Object Relations**

- 'Collection of objects' is an collection object, where each element stores a reference to some other object.
- Used to store details of groups of people or things.

```
List<ClassName> referenceVariable = new List<ClassName>();
referenceVariable.Add(object1);
referenceVariable.Add(object2);
referenceVariable.Add(object3);
...
```

**Object Relations** → **Next: Collection Hierarchy**

- An object can contain a field that stores references to one or more objects.

**One-to-One Relation**

```
class Student
{
    Branch branch; refer
}
```

The diagram shows a class box for 'Student' containing a field 'Branch branch;'. A green arrow labeled 'refer' points from the field name to a gray box labeled 'object of Branch class'.

**Object Relations** → **Next: Collection Hierarchy**

- An object can contain a field that stores references to one or more objects.

**One-to-One Relation**

```
class Student
{
    Branch branch; refer
}
```

**One-to-Many Relation**

```
class Student
{
    List<Examination> examinations; refer
}
```

The diagram shows a class box for 'Student' containing a field 'List<Examination> examinations;'. A green arrow labeled 'refer' points from the field name to a gray box labeled 'object of List<Examination>'. Below this, there are two smaller boxes: 'object of Examination' (with multiplicity 0..1) and 'object of Examination' (with multiplicity 1..n).

**Object Relations** → **Next: Collection Hierarchy**

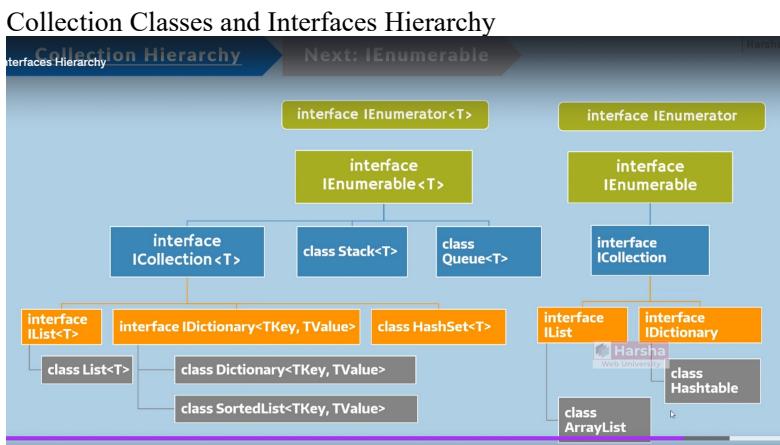
**Many-to-One Relation**

```
object of Employee class
Department department refer
```

```
object of Employee class
Department department refer
```

```
object of Employee class
Department department refer
```

The diagram shows three separate class boxes for 'Employee' containing a field 'Department department;'. Each has a green arrow labeled 'refer' pointing to a large gray box labeled 'object of Department'.



From developer point of view, remembering this hierarchical structure is very important not only for understanding purposes but also creating reference variable for the parent interface

instead of child class reference.

<https://learn.microsoft.com/en-us/dotnet/csharp/>

## IEnumerable and IEnumerator

The screenshot shows two code snippets side-by-side. The top snippet is titled 'System.Collections.Generic.IEnumerable<T>' and defines a generic interface with one method: 'IEnumerator<T> GetEnumerator();'. The bottom snippet is titled 'System.Collections.IEnumerable' and defines a non-generic interface with the same method. Both snippets are enclosed in orange boxes. At the top of the page, there is a breadcrumb navigation bar with 'IEnumeration' and 'Next: IEnumerator'.

```
System.Collections.Generic.IEnumerable<T>
public interface I Enumerable<out T> : I Enumerable
{
    IEnumerator<T> GetEnumerator();
}

System.Collections.IEnumerable
public interface I Enumerable
{
    IEnumerator GetEnumerator();
}
```

Actually **IEnumerable** and **IEnumerator** are **not comparable both are two individual interfaces** and one is not the alternative of other each has its own purpose and usage but for a developer it is necessary to understand both of these interfaces and when and how to use each one so this lecture is about how to work with the **IEnumerable** and how to work with the **Ienumerator**

as you can see both of these interfaces are the predefined interfaces some what related to collections but not

only limited to collections both of these are for arrays also see first we will consider understanding

about **IEnumerable** then we can talk about **IEnumerator** first of all there is an interface called **IEnumerable** under the **system.collections** namespace it has only one method called **enumerator** which has written type called **IEnumerator** the fundamental purpose of the **IEnumerable** is that to represent a group of elements

it can be an array it can be any type of collection such as list or array list and even it can be collection of characters

or anything else but the problem is that the **default IEnumerable interface** is not generic interface so it is not

possible to specify the data type that means it don't have any generic type here.

than so that is the reason in order to provide the ability to have generic collection classes there is an

alternative in **memorable of T** which is the generic version of the **ia numerable** and which is the child of **IEnumerable**

interface that means this one so the **system.connections.generic.IEnumerable of T** is the child of

**system.collections.IEnumerable** in other words the generic **IEnumerable** is the child of non-generic **IEnumerable**.

in other words the generic **IEnumerable** is the child of non-generic **IEnumerable** even the generic interface **IEnumerable**

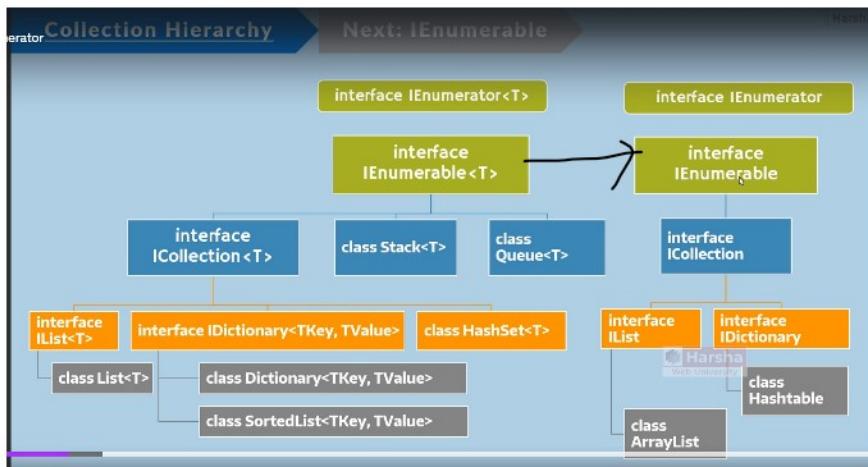
also contains a method called **get enumerator** much like non-genetic one but alternatively its written type is

the generic type that is **IEnumerator of team** so overall this generic version of **IEnumerable** is meant for

representation of a specific data type because of which you can specify any data type such as

int string or any other custom class while working with collections such as list, dictionary etc so all the other collection interfaces and classes

are directly or indirectly the children of IEnumerable as shown in the previous diagram so as you can see ICollection, IDictionary, IList and the classes such as hash set stack queue list dictionary etc are the children of IEnumerable which is ultimately the child of IEnumerable which is the non-generic version.



### 205. IEnumerable and IEnumerator

**IEnumerable** Next: IEnumerator

- The IEnumerable interface represents a group of elements.
- It is the parent interface of all types of collections.
- It is the parent of ICollection interface, which is implemented by other interfaces such as IList, IDictionary etc.

```
System.Collections.Generic.IEnumerable<T>
public interface IEnumerable<out T> : IEnumerable
{
    IEnumerator<T> GetGetEnumerator();
}

System.Collections.IEnumerable
public interface IEnumerable
{
    IEnumerator GetGetEnumerator();
}
```

### Working with IEnumerable

**IEnumerable** Next: IEnumerator

```
System.Collections.Generic.IEnumerable<T>
IEnumerable<T> referenceVariable = new List<T>();

System.Collections.IEnumerable
IEnumerable referenceVariable = new ArrayList();
```

### IEnumerator

**IEnumerator** Next: Iterators

- The IEnumerator interface is meant for readonly and sequential navigation of group of elements.
- IEnumerator is used by foreach loop internally.

**IEnumerator** > **Next, Iterators**

- The **IEnumerator** interface is meant for readonly and sequential navigation of group of elements.
- IEnumerator** is used by **foreach loop internally**.
- IEnumerable** interface has a method called **GetEnumerator** that returns an instance of **IEnumerator**.
- IEnumerator** by default starts with first element; **MoveNext()** method reads the next element; and the "Current" property returns the current element based on the current position.

```
System.Collections.Generic.IEnumerator<T>

public interface Ienumerator<out T> : IDisposable, Ienumerator
{
    T Current { get; }
}
```

```
System.Collections.IEnumerator

public interface Ienumerator
{
    object Current { get; }
    bool MoveNext();
    void Reset();
}
```

do you know the for each loop works based on **IEnumerator** the for each loop by default starts from the beginning of the group that means beginning of the collection and goes through each and every element in a sequential order right internally it works based on a numerator actually the **IEnumerator** has methods like move next and reset and it has a property called current so internally when you use the for each loop for any kind of collection or array it by default calls the reset method first and every time it calls the move next method in order to go to the next number and essence the value of current property into the variable that is declared in the for each loop that means internally how does the foreach loop works is that it works based on **IEnumerator** in other words if **IEnumerator** is not there for each loop will not work so whenever you use the for each loop first it calls the **reset method** that means the record pointer points to the beginning of the collection or array and then it immediately calls the move next method it returns true if any one element is found but returns false if no element is found so because of the first call of the move next method the record pointer points to the first element of the collection that means element at 0 and that particular element at 0 position will be assigned into the current property automatically and the value of the current property will be assigned automatically into the variable of for each loop this is how the item variable gets the value from the particular collection so at the beginning a reset method is being called and every time more next method will be called for each iteration and every time the next value that means the subsequent value will be assigned into the corresponding item variable which is declared here of course the variable name is not fixed though so if you call the more next method second time it points to the next element and that element first gets assigned into the current property of the a numerator and that current property will be assigned into this particular item variable so all this process is just internal process how does it work behind the scenes so every time when you use the for each loop the a numerator works so as per the developer point of view he may not notice the internal process but the a numerator is the fact how the for loop works.

```

static void Main()
{
    //create a collection
    I Enumerable<string> messages;
    messages = new List<string>() {"How are you", "Have a great day", "Thanks for meeting" };

    //foreach
    Console.WriteLine("I Enumerable:");
    foreach (string item in messages)
    {
        Console.WriteLine(item);
    }
}

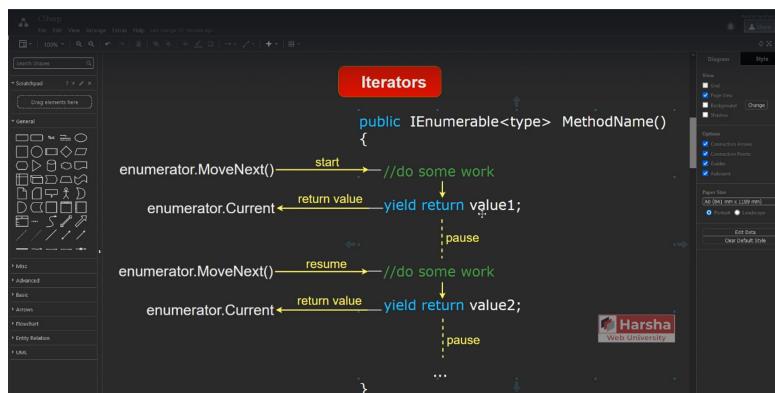
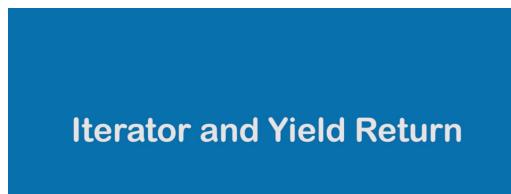
```



This is how item variable get the value from the collection.

IEnumerable is made for making foreach work.

## Iterator and Yield Return



The Iterator is a special type of method in C#

Generally when you call the method the complete body of the method executes and at last it returns a value optionally but iterator's return type will be innumerable of particular type

For example int or string or any other particular class so how does this iterator method executes it will do some work and return the value and pass the execution and later the execution can be resumed

again it will do some work that means execute some set of statements and then return the value and also pass the execution of that method in this way the execution model of this iterator

method is run and pass mode it will run some code and pass execution again run some code pass execution and this process repeats so this is the execution model of the iterator method so in the real-world applications when you want to do some larger task but as a step-by-step approach means do the first step and then pause again do the second step and then pause like this you can resume the execution of the method at any point of time wherever required or needed if you want this type of execution model you will use the **iterator methods**.

**yield return statement** returns a value but continues the execution of that method.

**Iterators** Next: Custom Collections

**What**

- > Iterator is a method which `yield return` statement returns elements one-by-one.
- > Iterator is used to iterate over a set of elements.
- > The `yield return` statement returns an element; but pauses execution of the method.

Iterator

```
public IEnumerable<type> MethodName()
{
    for (int i = 0; i < 10; i++)
    {
        yield return value;
    }
}
```

IEnumerator<T>

value0
value1
Harsha
value2
...

**Iterators** Next: Custom Collections

- > Iterator can be a method or get accessor of a property.
- > Iterator can't be constructor or destructor.
- > Iterator can't contain ref or out parameters.
- > Multiple `yield` statements can be used within the same iterator.
- > Iterators are generally implemented in custom collections.
- > While using the iterator, you must import the `System.Collections.Generic` namespace.

Harsha

## Custom Collections

suppose you have created a collection of objects maybe collection of customers or employees whatever required

suppose you would like to add a specific set of methods that are applicable for that particular type of collection

so these additional methods might manipulate these collection of objects let's say for example there is a method called search by name so that method will search for the customers based on the name so your

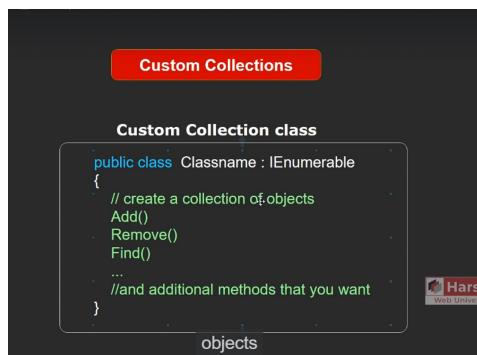
requirement is you would like to

add those type of methods in this class and that class is called as custom collection class  
so a custom collection class is a class that contains a collection of objects for example list of customer class

here customer is the name of the user defined class it can be any other class whatever you want in addition

it contains all the appropriate methods that will manipulate those collection of objects such as add remove find or search by name set by phone number or any other manipulations that you want to perform on that collection so those complete set of methods that manipulate the collection of objects can be created as a group and that group is the custom collection class in order to make that class as a custom collection technically it should inherit from a enumerable

generally IEnumerable represents that there are multiple values here in this case it is collection of objects.



If you want to check for example your customer id should begin with capital a or any other condition it cannot be checked by the predefined add method and that is what you can add validation logic here and that is what the purpose or reason why we create custom collections.

so if you ask me why we have to create custom collection class why can't we simply use the list of customer like

this because it gives you more grip on the methods you can add extra methods like for example search or search by 4

number etc which are not exist in the predefined collection classes and also in the same existing methods

like add also you can add additional validation logic that you want so this is called more grip in the

collection methods and that is what exactly offered by the custom collection classes

200. Custom Collections

## Custom Collections

**What**

- Custom collection class makes it easy to store collection of objects of specific class.
- It allows you to create additional properties and methods useful to manipulate the collection.

**Custom Collection**

```
public class Classname : IEnumerable
{
    private List<yourClassName> list = new List<yourClassName>();

    public IEnumerator GetEnumerator()
    {
        for (int index = 0; index < list.Count; index++)
        {
            yield return list[index];
        }
    }
}
```

Next: IComparable

Custom Collections

**Next: IEquatable**

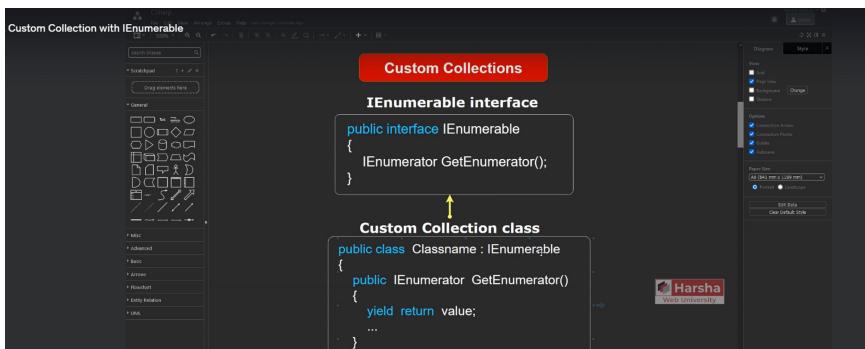


- It implements `IEnumerable` interface; and stores a collection as a private field.
- It can implement methods such as `Add`, `AddRange`, `Find` etc., and also indexer optionally.
- You can also create custom collection by implementing `IList<T>`; but you need to implement all methods of `IList<T>` interface in this case.
- You can also create custom collection by inheriting from `List`; it provides all methods of `List` to your collection class.

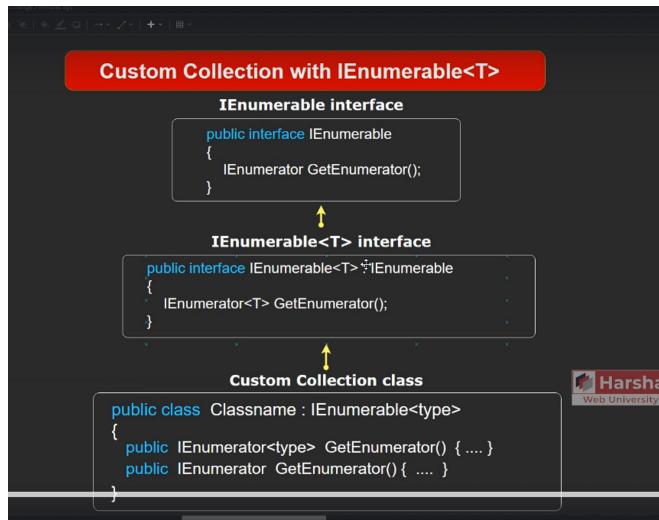
Harsha Web University

## Custom Collections with `IEnumerable`

Custom Collection with `IEnumerable<T>`



Rather than creating custom collection class based on non-generic version of `innumerable`; it is better to go with `innumerable of T`



```

//private collection as a list
private List<Customer> customers = new List<Customer>();

//implementing IEnumerable.GetEnumerator()
0 references
IEnumerator IEnumerable.GetEnumerator()
{
    return this.GetEnumerator();
}

//implementing IEnumerable<Customer>.GetEnumerator()
2 references
public IEnumerator<Customer> GetEnumerator()
{
    for (int i = 0; i < customers.Count; i++)
    {
        yield return customers[i]; //return and pause
    }
}
    
```

enumerator of customer here but the

but the problem with Innumerable of T is that it doesn't provide any specific concrete methods like for example Add, Add Range, Insert find etc so in order to implement those methods

also in your class it is better to implement ICollection or IList

and we will try that in the next lecture you.

## Custom Collection with ICollection<T>

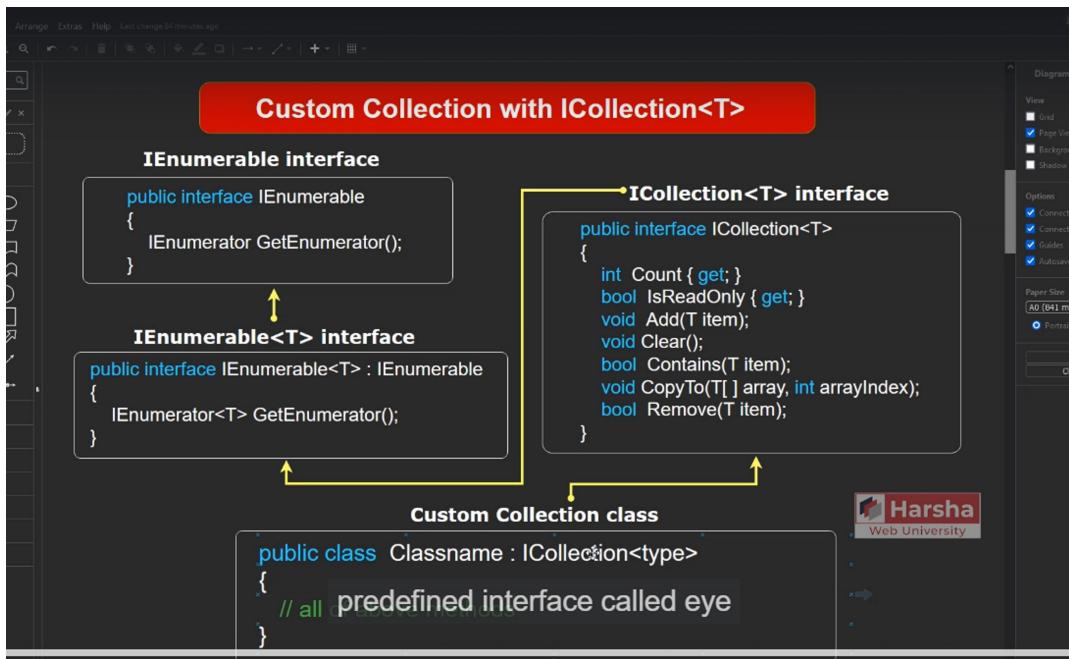
you can also try creating the custom collection class by implementing this predefined interface called ICollection

and this ICollection contains the methods that is add remove contents and other methods as you can see

so the benefit of implementing this interface is that in your class you need to implement all these methods

so that you can add custom logic in these methods in addition apart from these methods

you can add additional methods whatever needed in your own custom class



```
1 reference
public Customer Find(Predicate<Customer> match)
{
    return customers.Find(match);
}

0 references
public List<Customer> FindAll(Predicate<Customer> match)
{
    + customer.CustomerType;
}

//Find
Customer matchingCustomer = customersList.Find(cust => cust.CustomerID == 1001);
if (matchingCustomer != null)
{
    Console.WriteLine(matchingCustomer.CustomerID + ", " + matchingCustomer.CustomerName);
}
```

```
1 reference
public List<Customer> FindAll(Predicate<Customer> match)
{
    return customers.FindAll(match);
}

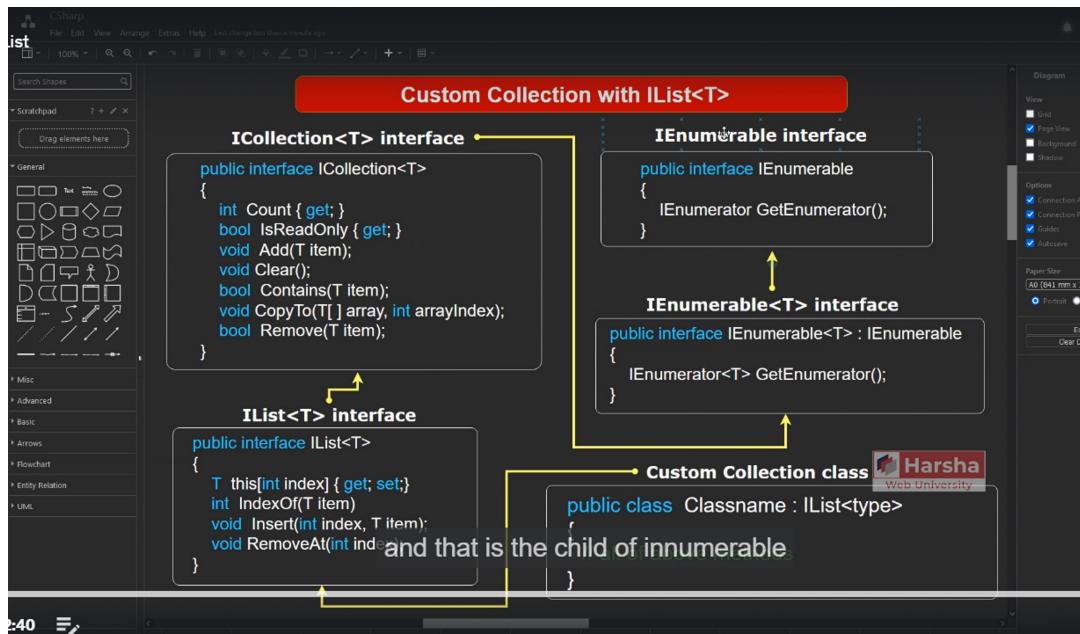
0 references
class Program
{
}

//FindAll
List<Customer> vipcust = customersList.FindAll(cust => cust.CustomerType ==
    TypeOfCustomer.VIPCustomer);

customersList.Clear();

Console.ReadKey();
```

# Custom Collection with `IList<T>`



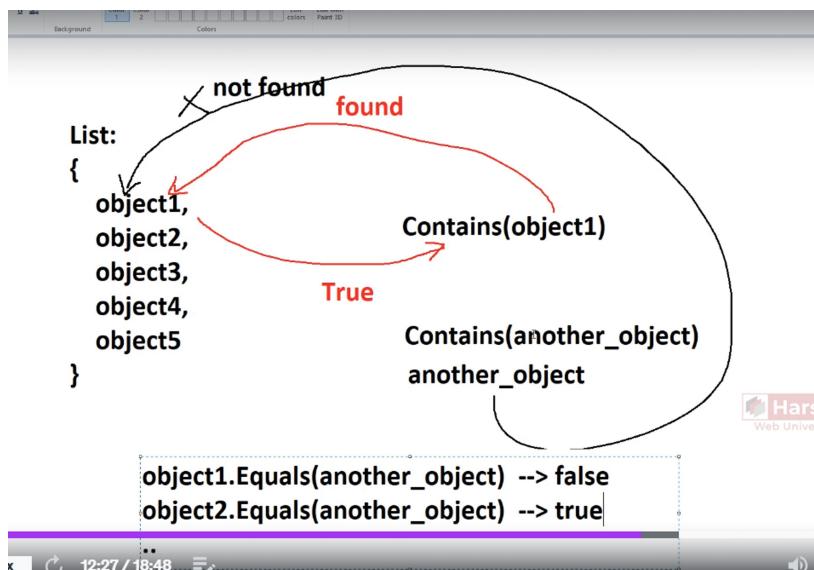
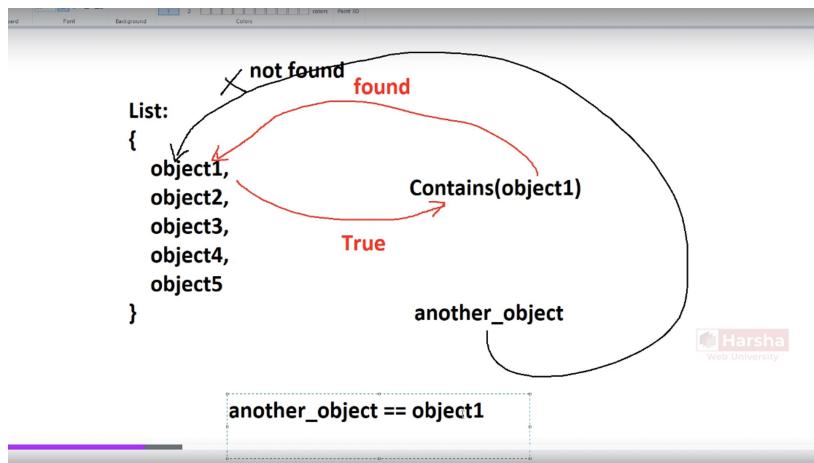
## IEquatable

In the `Ilist` there is a predefined method called `Contains` so if you supply an object for this method it will set for that object in the list and if that object is already found in the list it returns true otherwise false but the problem with `Contains` method is it can identify the same object but cannot identify the other objects with the same details.

so in order to solve this problem you are required to implement a equivalent interface in the model class.

suppose the requirement is through this contains method you want to search for the particular customer

if the customer details are matching you want to return true.



**IEquatable**      Next: **IComparable**

**What**

- › The `System.IEquatable<T>` interface has a method called "Equals", which determines whether the current object and parameter object are logically equal or not, by comparing data of fields.
- › It can be implemented in the class to make the objects comparable.
- › It is useful to invoke `List.Contains` method to check whether the object is present in the collection or not.

`interface System.IEquatable<T>`

```
public interface IEquatable<T>
{
    bool Equals(T other);
}
```

### Implementation of IEquatable interface

```
public class ClassName : IEquatable<ClassName>
{
    public bool Equals(ClassName other)
    {
        return this.field1 == other.field1 && this.field2 == other.field2;
    }
}
```



## IComparable

### IComparable<T>

#### IComparable interface

```
public interface IComparable
{
    int CompareTo(object obj)
}
```

↑  
Model class

```
public class Classname : IComparable
{
    public int CompareTo(object obj)
    {
        // return 0 or positive or negative value
    }
}
```



## IComparable

## Next: IComparer

| Harsha

### What

- The System.IComparable interface has a method called "CompareTo", which determines order of two objects i.e. current object and parameter object.

```
interface System.IComparable  
  
public interface IComparable  
{  
    int CompareTo(object obj);  
}
```



### Return value

- 0 : "this" object and parameter object occur in the same position (unchanged).
- <0 : "this" object comes first; parameter object comes next.
- >0 : parameter object comes first; "this" object comes next.

104, Mary, Designer  
102, Alexa, Manager  
101, Steven, Consultant  
103, Jade, Analyst

102.CompareTo(104)  
102 - 104 = -2

Expected final output:  
101, Steven, Consultant  
102, Alexa, Manager  
103, Jade, Analyst  
104, Mary, Designer

Harsha  
Web University

104, Mary, Designer  
102, Alexa, Manager  
101, Steven, Consultant  
103, Jade, Analyst

102.CompareTo(104)  
102 - 104 = -2

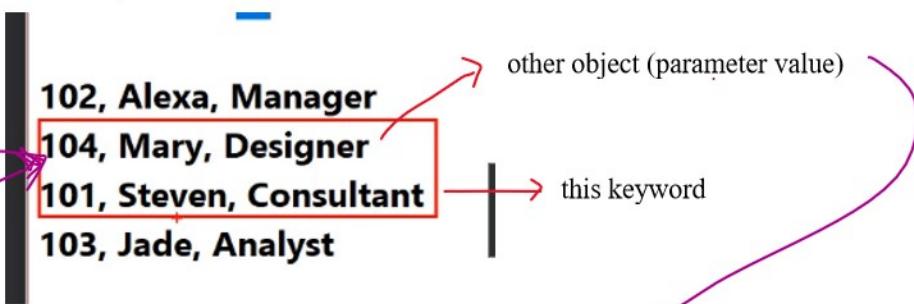
102, Alexa, Manager  
104, Mary, Designer  
101, Steven, Consultant  
103, Jade, Analyst

Expected final output:  
101, Steven, Consultant  
102, Alexa, Manager  
103, Jade, Analyst  
104, Mary, Designer

Harsha  
Web University

```
public int CompareTo(object other)  
{  
    Employee otherEmp = (Employee)other;  
    Console.WriteLine(this.EmpID + ", " + otherEmp.EmpID);  
    return this.EmpID - otherEmp.EmpID;  
}
```

negative value will recall the  
CompareTo() method with these two  
objects.



```
0 references
public int CompareTo(object other)
{
    Employee otherEmp = (Employee)other;
    Console.WriteLine(this.EmpID + " " + otherEmp.EmpID);
    return this.EmpID - otherEmp.EmpID;
}
```

```
File Edit Format View Help
104, Mary, Designer
102, Alexa, Manager
101, Steven, Consultant
103, Jade, Analyst

102.CompareTo(104)
102 - 104 = -2

102, Alexa, Manager
104, Mary, Designer
101, Steven, Consultant
103, Jade, Analyst

101.CompareTo(104)
101 - 104 = -3

Expected final output:
101, Steven, Consultant
102, Alexa, Manager
103, Jade, Analyst
104, Mary, Designer
```

Harsha

Ln 14 Col 15 110% Windows (CRLF) UTF-8

File Edit Format View Help

104, Mary, Designer  
102, Alexa, Manager  
101, Steven, Consultant  
103, Jade, Analyst

**102.CompareTo(104)**  
**102 - 104 = -2**

102, Alexa, Manager  
104, Mary, Designer  
101, Steven, Consultant  
103, Jade, Analyst  
**101.CompareTo(104)**  
**101 - 104 = -3**

**102, Alexa, Manager**  
**101, Steven, Consultant**  
104, Mary, Designer  
103, Jade, Analyst  
**101.CompareTo(102)**  
**101 - 102 = -1**

101, Steven, Consultant  
102, Alexa, Manager  
104, Mary, Designer  
103, Jade, Analyst  
**103.CompareTo(104)**  
**103 - 104 = -1**

101, Steven, Consultant  
102, Alexa, Manager  
103, Jade, Analyst  
104, Mary, Designer

**Harsha**  
Expected final output:  
101, Steven, Consultant  
102, Alexa, Manager  
103, Jade, Analyst  
**104, Mary, Designer**

**101, Steven, Consultant**  
**102, Alexa, Manager**  
**103, Jade, Analyst**  
104, Mary, Designer  
**103.CompareTo(102)**  
**103 - 102 = 1**

**Harsha**  
Web University

**Expected final output:**  
**101, Steven, Consultant**  
**102, Alexa, Manager**  
**103, Jade, Analyst**  
**104, Mary, Designer**

Ant

Bat

Ant.CompareTo(Bat) "A" should come first. So it returns -1

Apple.CompareTo(Ant) 1

Apple.CompareTo(Apple) 0



IComparable → Next: IComparer

**What**

- The System.IComparable interface has a method called "CompareTo", which determines order of two objects i.e. current object and parameter object.
- It can be implemented in the class to make the objects sortable.
- It is useful to invoke `List.Sort` method to sort the collection of objects.

```
interface System.IComparable
{
    public interface IComparable
    {
        int CompareTo(object obj);
    }
}
```

**Return value**

- 0 : "this" object and parameter object occur in the same position (unchanged).
- <0 : "this" object comes first; parameter object comes next.
- >0 : parameter object comes first; this object comes next.

213.IComparable 28:41 / 29:08



IComparable<T>

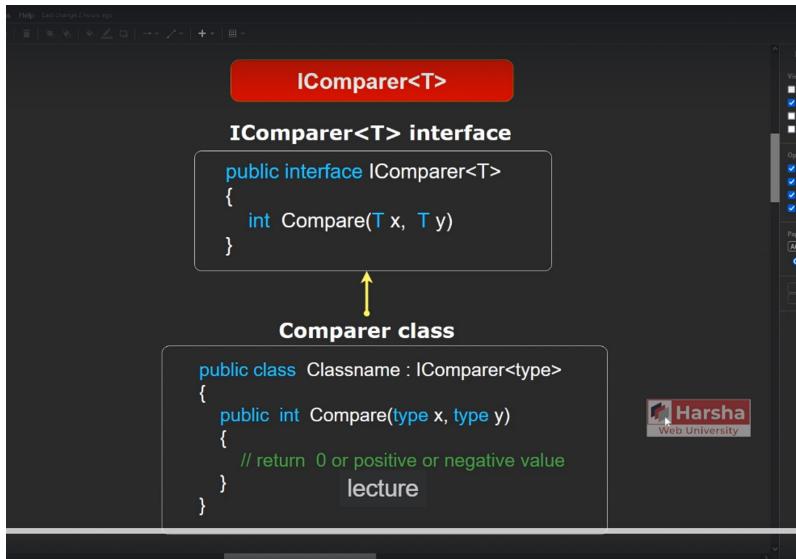
**IComparable interface**

```
public interface IComparable
{
    int CompareTo(object obj);
}
```

**Model class**

```
public class Classname : IComparable
{
    public int CompareTo(object obj)
    {
        // return 0 or positive or negative value
    }
}
```

Harsha Web University



so whenever your model class like employee class implements IComparable interface you can sort that list of employees we have already done the same earlier in the previous lecture but sometimes in the real world applications suppose a developer creates a model class but he does not implement a comparable interface in that model class but you are creating a list of that particular model class for example you have created list of employees but the developer of that employee class does not implemented that IComparable interface so then how do you sort that list of employees exactly to face this kind of situation IComparer is alternative for that Icomparable this IComparer interface can be implemented as a separate custom class so you need not make any changes in the actual model class.

**Return value**

- > 0 : both x and y are equal; so will be kept in the same position.
- > <0 : x comes first; y comes next.
- > >0 : y comes first; x comes next.

```

interface System.Collections.Generic.IComparer
public class ClassName : IComparer<T>
{
    public int Compare(T x, T y)
    {
        return value;
    }
}

```

# IComparer

## Part 2

IComparer

Next: Covariance and Contravariance

### What

- > The System.Collections.Generic.IComparer interface has a method called "Compare", which determines order of two objects i.e. current object and parameter object.
- > It can be implemented by a separate class to make the objects sortable.
- > It is useful to invoke `List.Sort` method to sort the collection of objects.
- > It is an alternative to IComparable; useful for the classes that doesn't implement IComparable.

```
interface System.Collections.Generic.IComparer  
public interface IComparer<T>  
{  
    int Compare(T x, T y);  
}
```



# Covariance



```

{
    0 references
    static void Main()
    {
        //create object
        LivingThing livingThing = new Parrot(); //not covariance
        Parrot parrot = new Parrot(); //normal

        IMover<LivingThing> mover = new Mover<Parrot>();
    }
}

```

by default, it is ok.

but this is not ok in generic parameter by default.

In order to solve this problem the **covariance** concept is introduced

solution:

```

2 references
interface IMover<out|T>
{
}

```

by adding 'out' keyword, it becomes 'covariance'. here out keyword says that, you can use this out keyword T only as **a return type of a method or property**, you can not use the same as argument type.

```
2 references
interface IMover<out T>
{
}
```

can use this out keyword **T** only as **a return type** of a method or property. you can not use the same as argument type.

```
Covariance
interface IMover<out T>
{
    0 references
    Method1(T x);
}
1 reference
class Mover<T> : IMover<T>
```

Not Allowed!

**Covariance** → **Next: Contravariance**

**What**

- › Covariance allows you to supply child type, where the parent type is expected.
- › Implemented with "out" keyword for the generic type parameter of an interface.
- › In this case, the generic type parameter can be used for return types of methods or properties created in the interface.

**Covariance**      Harsha  
InterfaceName<ParentType> variable = new ClassName<ChildType>();

The child class name that is **living thing** is expected but in practical in the right hand side we are trying to supply **tSobhe** child class name so parent is expected but you are going to supply the **child that is parrot** that is where the covariance is implemented but this should be happened only with generic type parameters but not in simple type of code this is not covariance.

```
//IMover<Parrot> mover = new Mover<Parrot>();
IMover<LivingThing> mover = new Mover<Parrot>()
{
    thing = parrot
};
```

*Sob y bujhlaam, but etar real world benefit ta ki?*

```
class Sample
{
    //public void PrintValues(List<string> values)
    //{
    //    foreach (var item in values)
    //    {
    //        Console.WriteLine(item + ",");
    //    }
    //    Console.WriteLine();
    //}

    // but I want to receive all kind of list
    //public void PrintValues(List<Object> values)
    //{
    //    foreach (var item in values)
    //    {
    //        Console.WriteLine(item + ",");
    //    }
    //    Console.WriteLine();
    //}

    // but it does not work? so, this the case when we need Co-variance.

    //IEnumerable already implemented covariance concept.
    public void PrintValues(IEnumerable<Object> values)
    {
        foreach (var item in values)
        {
            Console.WriteLine(item + ",");
        }
        Console.WriteLine();
    }
}
```

```
Sample s = new Sample();
s.PrintValues(new List<string>
{
    "Hello",
    "World",
    "Bangladesh"
});
```

```
2 references
interface IMover<out T>
{
    2 references
    T Move();
}

1 reference
class Mover<T>: IMover<T>
{
    2 references
    public T thing { get; set; }
    2 references
    public T Move()
    {
        return thing;
    }
}

3
4
5
6 namespace System.Collections.Generic
7 {
8     public interface IEnumerable<out T> : IEnumerable
9     {
10         I IEnumerator<T> GetEnumerator();
11     }
12 }
```

and that is the same how we have

But List is not Covariance supported

But List is not Covariance supported.

```
8
9  namespace System.Collections.Generic
10 {
11     public class List<T> : IList<T>,
12         ICollection<T>, IEnumerable<T>,
13         IEnumerable, IList, ICollection,
14         IReadOnlyList<T>, IReadOnlyCollection<T>
15     {
16         public List();
17         public List(int capacity);
18         public List(IEnumerable<T> collection);
```

The screenshot shows two windows from Visual Studio. The left window displays a C# code editor with the following code:

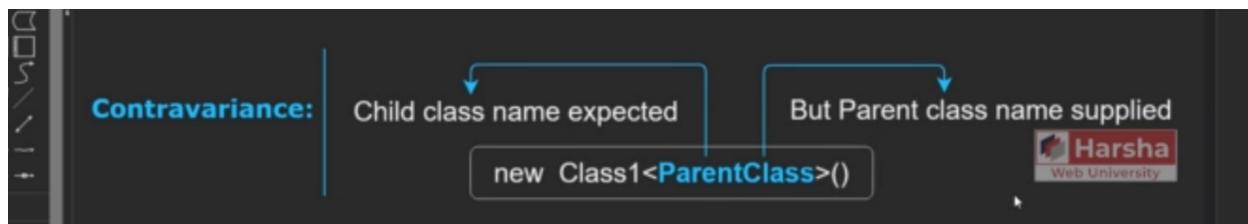
```
32  class Sample
33  {
34      public void PrintValues(List<object>
35          values)
36      {
37          foreach (var item in values)
38          {
39              Console.Write(item + " ");
40          }
41          Console.WriteLine();
42      }
43  }
```

The right window shows the definition of the `IEnumerable` interface from the `mscorlib` assembly:

```
1  Assembly mscorlib, Version=4.0.0.0, Culture=n
2
3
4
5
6  namespace System.Collections.Generic
7  {
8      public interface IEnumerable<out T> :
8          IEnumerable
9      {
10          I IEnumerator<T> GetEnumerator();
11      }
12  }
```

A tooltip for the `List<object>` call in the code editor points to the `IEnumerable` interface definition.

# Contravariance



Covariance Next: Contravariance

What

- > Covariance allows you to supply child type, where the parent type is expected.
- > Implemented with "out" keyword for the generate type parameter of an interface.
- > In this case, the generic type parameter can be used for return types of methods or properties created in the interface.

Covariance

Harsha Web University

```
InterfaceName<ParentType> variable = new ClassName<ChildType>();
```

**What**

- › Contravariance allows you to supply parent type, where the child type is expected.
- › Implemented with "in" keyword for the generic type parameter of an interface.
- › In this case, the generic type parameter can be used for parameter types of methods created in the interface.

## Contravariance



```
InterfaceName<ChildType> variable = new ClassName<ParentType>();
```