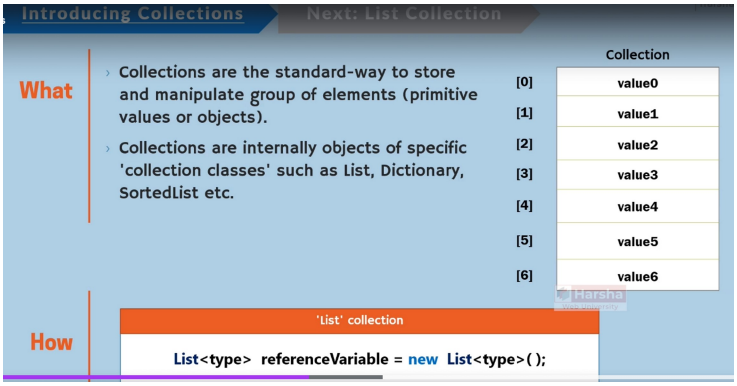
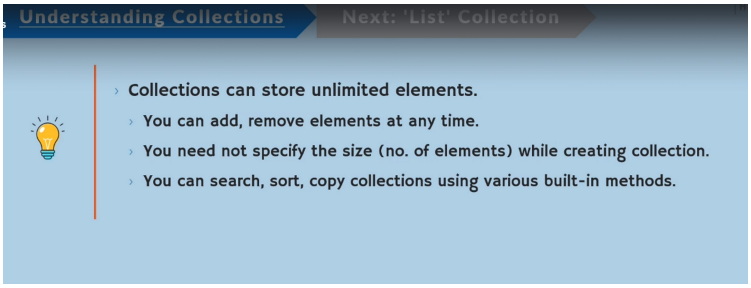
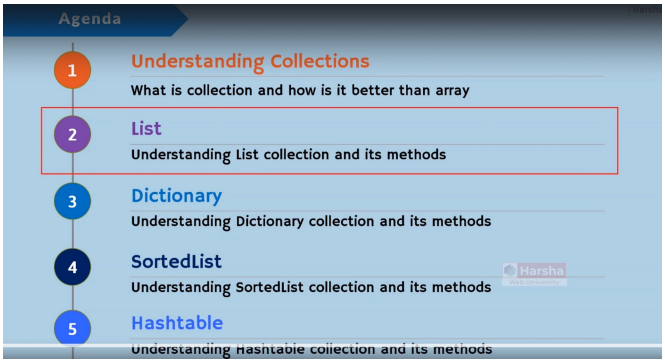
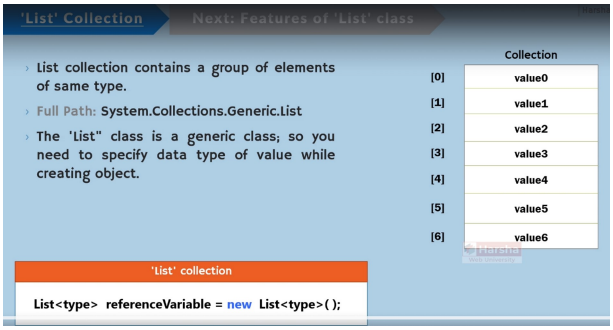
Section 22 : Collections

17 July 2024 20:31

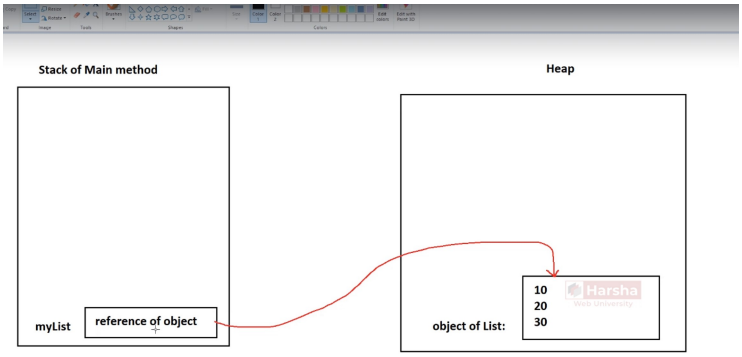








```
using System;
using System.Collections.Generic;

namespace ListExample
{
    0 references
    class Program
    {
        0 references
        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

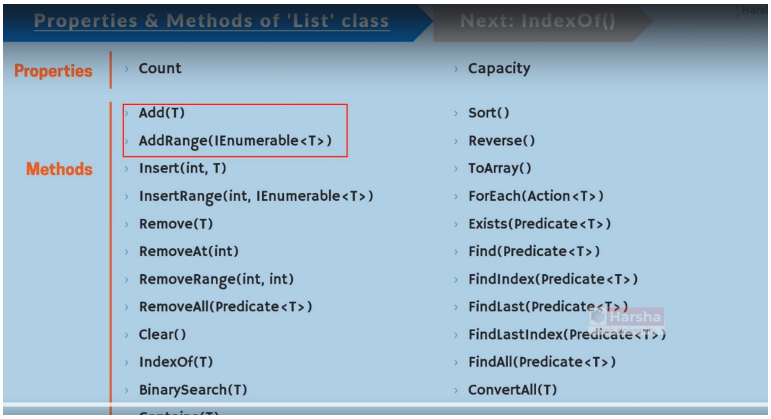
Next: Methods of List class

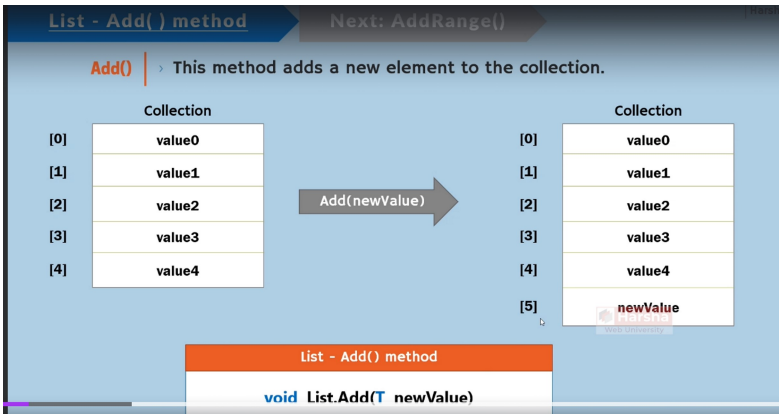
- It is dynamically sized. You can add, remove elements at any time.
- It allows duplicate values.
- It is index-based. You need to access elements by using zero-based index.
- It is not sorted by default. The elements are stored in the same order, how they are initialized.
- It uses arrays internally; 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.

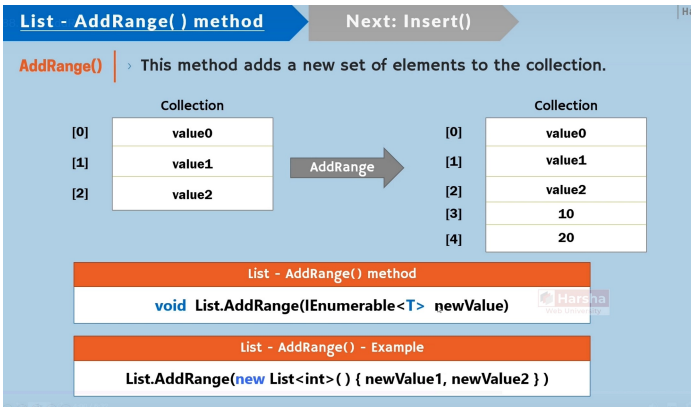
Properties & Methods of 'List' class

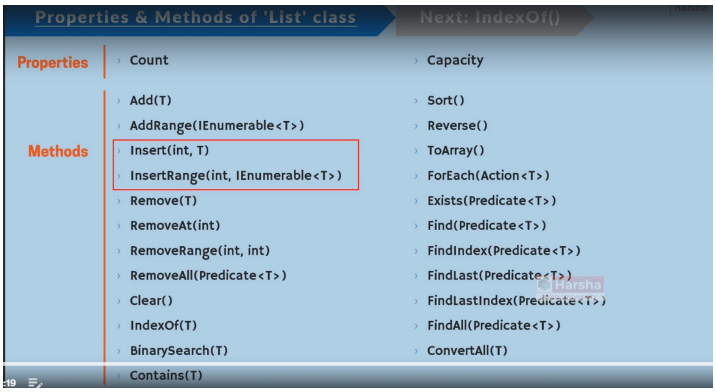
Next: IndexOf()

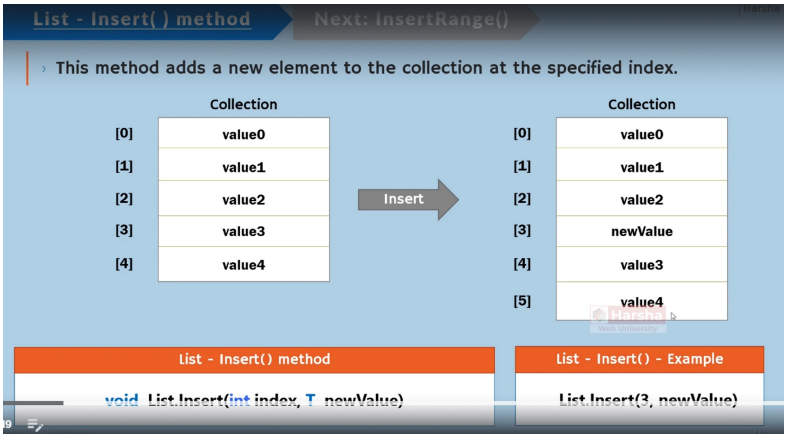
Properties	Count	Capacity
Methods	Add(T)	Sort()
	AddRange(IEnumerable<T>)	Reverse()
	Insert(int, T)	ToArray()
	InsertRange(int, IEnumerable<T>)	ForEach(Action<T>)
	Remove(T)	Exists(Predicate<T>)
	RemoveAt(int)	Find(Predicate<T>)
	RemoveRange(int, int)	FindIndex(Predicate<T>)
	RemoveAll(Predicate<T>)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

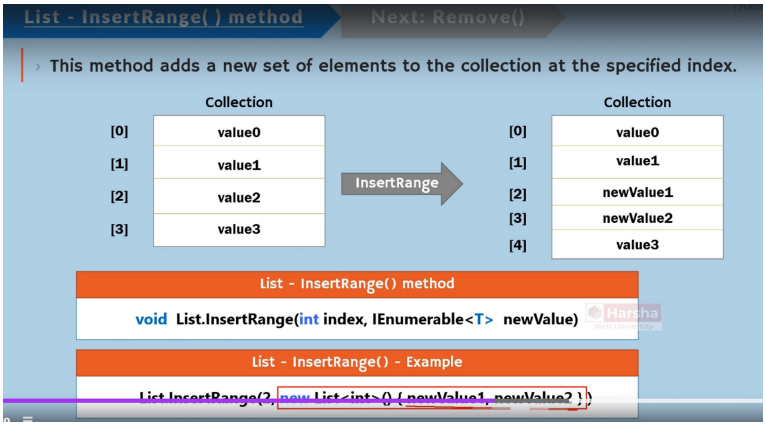

Properties & Methods of 'List' class
Next: IndexOf()
Properties
Count
Capacity
Methods
Add(T)
AddRange(IEnumerable<T>)
Insert(int, T)
InsertRange(int, IEnumerable<T>)
Remove(T)
RemoveAt(int)
RemoveRange(int, int)
RemoveAll(Predicate<T>)
Clear()
IndexOf(T)
BinarySearch(T)
Sort()
Reverse()
ToArray()
ForEach(Action<T>)
Exists(Predicate<T>)
Find(Predicate<T>)
FindIndex(Predicate<T>)
FindLast(Predicate<T>)
FindLastIndex(Predicate<T>)
FindAll(Predicate<T>)
ConvertAll(T)

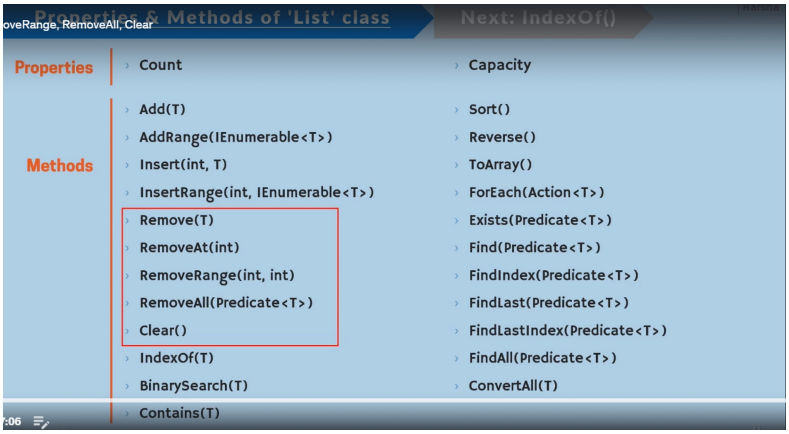

List - Add( ) method
Next: AddRange()
Add()
This method adds a new element to the collection.
Collection
[0] value0
[1] value1
[2] value2
[3] value3
[4] value4
Add(newValue)
Collection
[0] value0
[1] value1
[2] value2
[3] value3
[4] value4
[5] newValue
List - Add() method
void List.Add(T newValue)

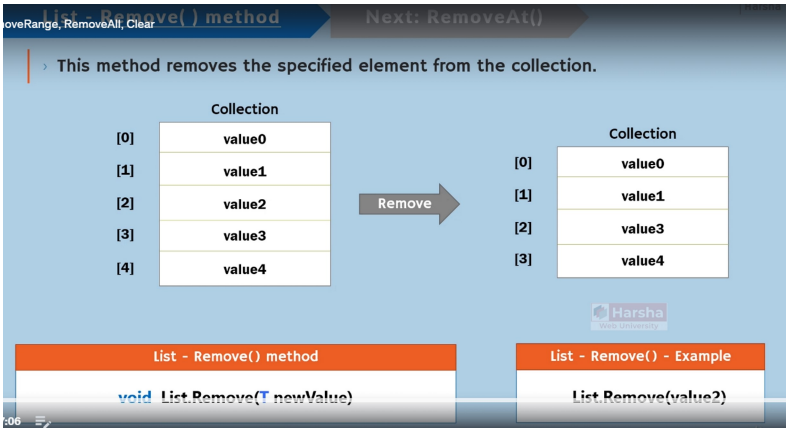

List - AddRange( ) method
Next: Insert()
AddRange()
This method adds a new set of elements to the collection.
Collection
[0] value0
[1] value1
[2] value2
AddRange
Collection
[0] value0
[1] value1
[2] value2
[3] 10
[4] 20
List - AddRange() method
void List.AddRange(IEnumerable<T> newValue)
List - AddRange() - Example
List.AddRange(new List<int>( ) { newValue1, newValue2 })

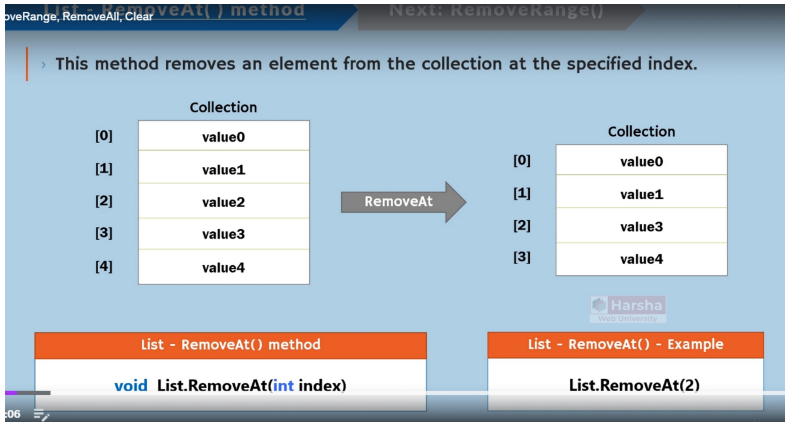

Properties & Methods of 'List' class
Next: IndexOf()
Properties
Count
Capacity
Methods
Add(T)
AddRange(IEnumerable<T>)
Insert(int, T)
InsertRange(int, IEnumerable<T>)
Remove(T)
RemoveAt(int)
RemoveRange(int, int)
RemoveAll(Predicate<T>)
Clear()
IndexOf(T)
BinarySearch(T)
Contains(T)
Sort()
Reverse()
ToArray()
ForEach(Action<T>)
Exists(Predicate<T>)
Find(Predicate<T>)
FindIndex(Predicate<T>)
FindLast(Predicate<T>)
FindLastIndex(Predicate<T>)
FindAll(Predicate<T>)
ConvertAll(T)

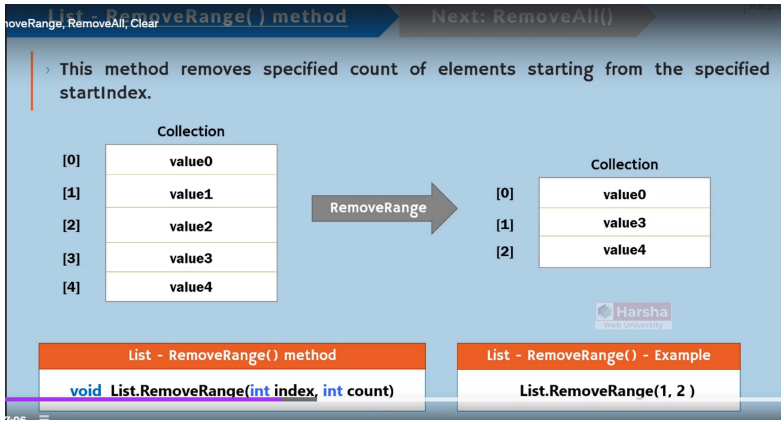

List - Insert( ) method
Next: InsertRange()
This method adds a new element to the collection at the specified index.
Collection
[0] value0
[1] value1
[2] value2
[3] value3
[4] value4
Insert
Collection
[0] value0
[1] value1
[2] value2
[3] newValue
[4] value3
[5] value4
List - Insert() method
void List.Insert(int index, T newValue)
List - Insert() - Example
List.Insert(3, newValue)

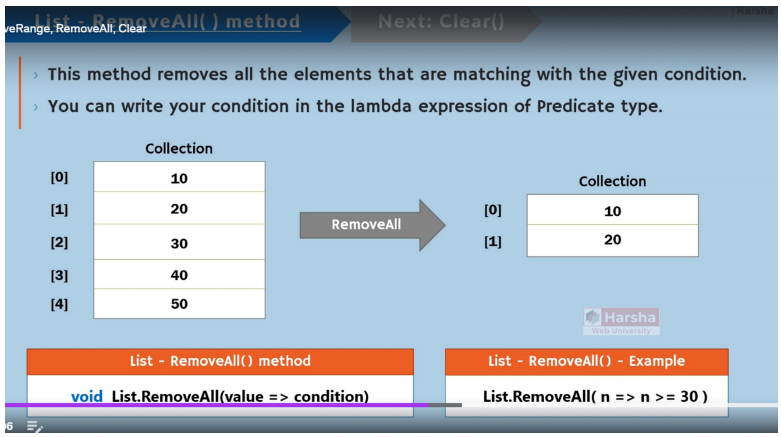

List - InsertRange( ) method
Next: Remove()
This method adds a new set of elements to the collection at the specified index.
Collection
[0] value0
[1] value1
[2] value2
[3] value3
InsertRange
Collection
[0] value0
[1] value1
[2] newValue1
[3] newValue2
[4] value3
List - InsertRange() method
void List.InsertRange(int index, IEnumerable<T> newValue)
List - InsertRange() - Example

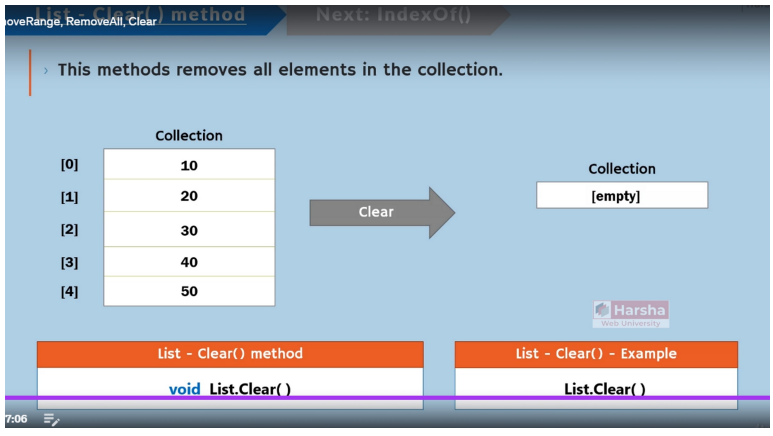

Properties & Methods of 'List' class
Next: IndexOf()
Properties
Count
Capacity
Methods
Add(T)
AddRange(IEnumerable<T>)
Insert(int, T)
InsertRange(int, IEnumerable<T>)
Remove(T)
RemoveAt(int)
RemoveRange(int, int)
RemoveAll(Predicate<T>)
Clear()
IndexOf(T)
BinarySearch(T)
Contains(T)
Sort()
Reverse()
ToArray()
ForEach(Action<T>)
Exists(Predicate<T>)
Find(Predicate<T>)
FindIndex(Predicate<T>)
FindLast(Predicate<T>)
FindLastIndex(Predicate<T>)
FindAll(Predicate<T>)
ConvertAll(T)

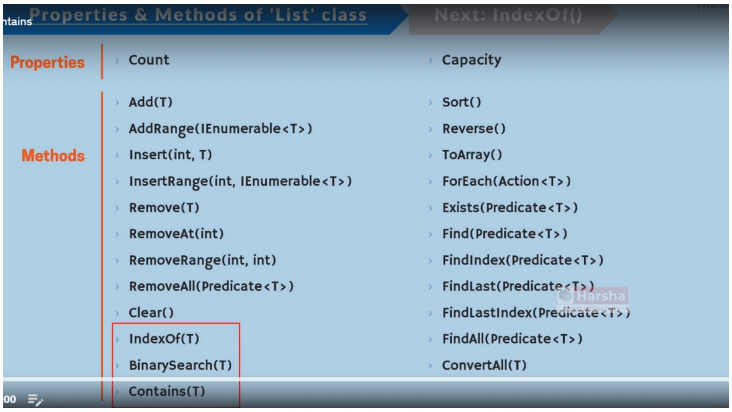

List - Remove( ) method
Next: RemoveAt()
This method removes the specified element from the collection.
Collection
[0] value0
[1] value1
[2] value2
[3] value3
[4] value4
Remove
Collection
[0] value0
[1] value1
[2] value3
[3] value4
List - Remove() method
void List.Remove(T newValue)
List - Remove() - Example
List.Remove(value2)

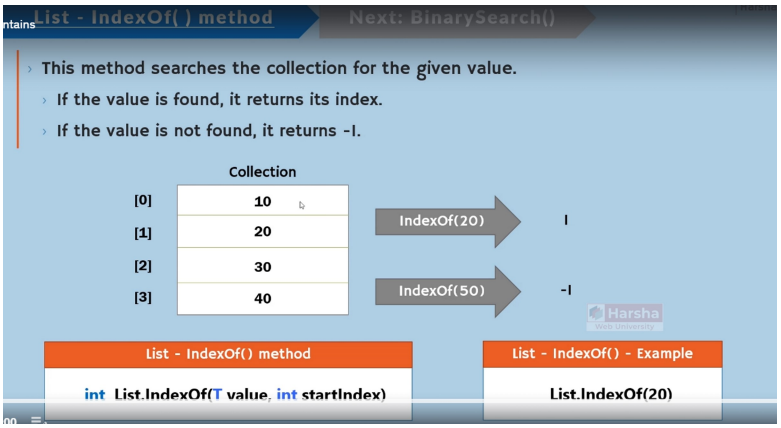

RemoveRange, RemoveAll, Clear
Next: RemoveRange()
This method removes an element from the collection at the specified index.
Collection
[0] value0
[1] value1
[2] value2
[3] value3
[4] value4
RemoveAt
Collection
[0] value0
[1] value1
[2] value3
[3] value4
List - RemoveAt() method
void List.RemoveAt(int index)
List - RemoveAt() - Example
List.RemoveAt(2)

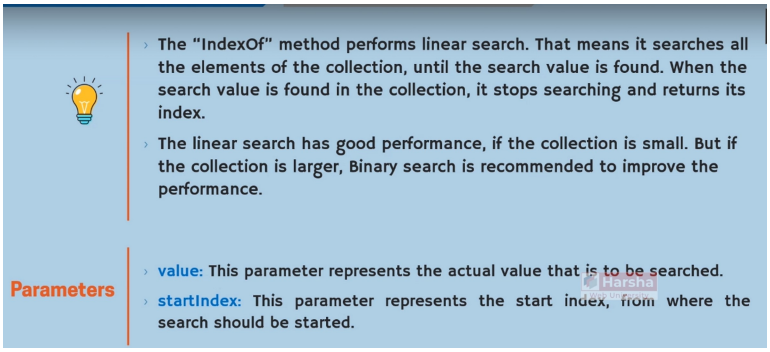

RemoveRange, RemoveAll, Clear
List - RemoveRange() method
Next: RemoveAll()
This method removes specified count of elements starting from the specified startIndex.
Collection
[0] value0
[1] value1
[2] value2
[3] value3
[4] value4
RemoveRange
Collection
[0] value0
[1] value3
[2] value4
List - RemoveRange() method
void List.RemoveRange(int index, int count)
List - RemoveRange() - Example
List.RemoveRange(1, 2)

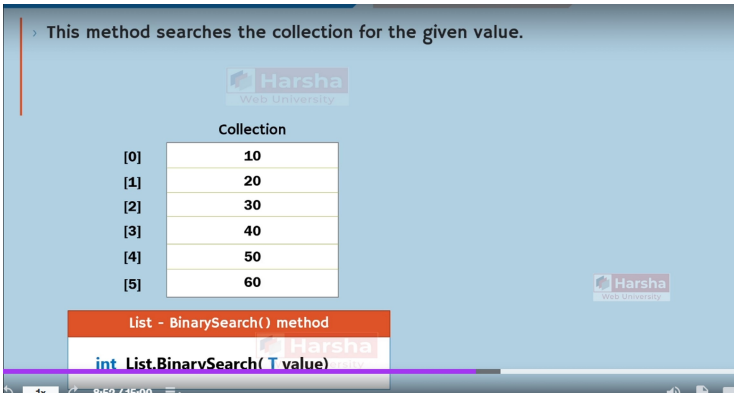

RemoveRange, RemoveAll, Clear
List - RemoveAll() method
Next: Clear()
This method removes all the elements that are matching with the given condition.
You can write your condition in the lambda expression of Predicate type.
Collection
[0] 10
[1] 20
[2] 30
[3] 40
[4] 50
RemoveAll
Collection
[0] 10
[1] 20
List - RemoveAll() method
void List.RemoveAll(value => condition)
List - RemoveAll() - Example
List.RemoveAll(n => n >= 30)


RemoveRange, RemoveAll, Clear
List - Clear() method
Next: IndexOf()
This methods removes all elements in the collection.
Collection
[0] 10
[1] 20
[2] 30
[3] 40
[4] 50
Clear
Collection
[empty]
List - Clear() method
void List.Clear( )
List - Clear() - Example
List.Clear( )


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 - IndexOf( ) method
Next: BinarySearch()
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
IndexOf(20)
1
IndexOf(50)
-1
List - IndexOf() method
int List.IndexOf(T value, int startIndex)
List - IndexOf() - Example
List.IndexOf(20)


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

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.

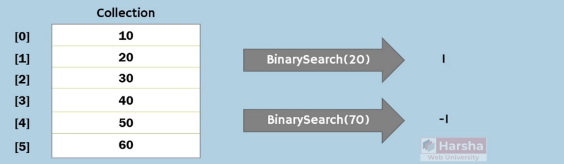
Parameters

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

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.

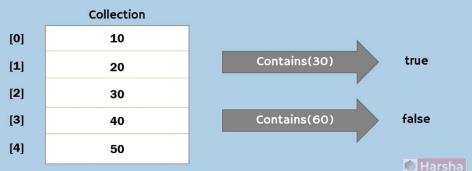


List - BinarySearch() method	List - BinarySearch() - Example
int List.BinarySearch(T value)	List.BinarySearch(20)

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.

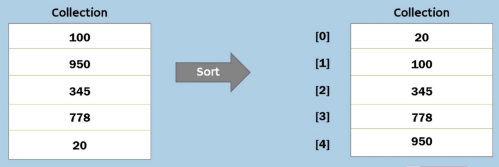


List - Contains() method	Contains() - Example
bool List.Contains(T value)	List.Contains(30)

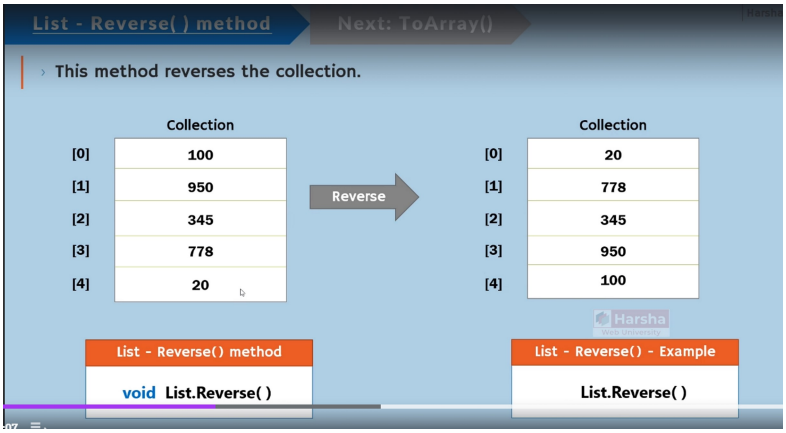
List - Sort( ) method

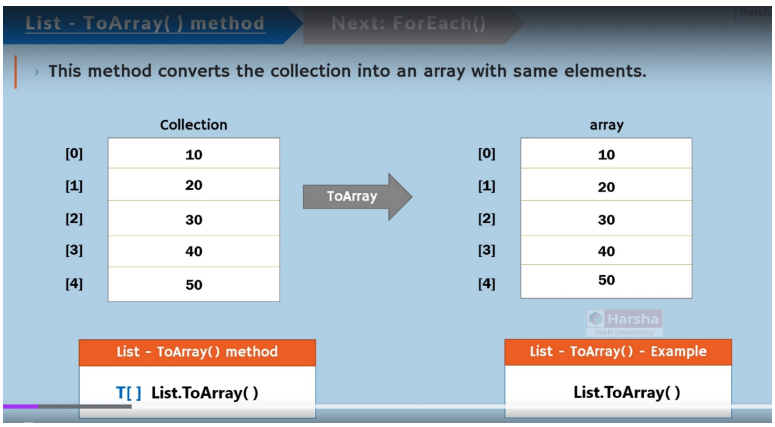
Next: Reverse()

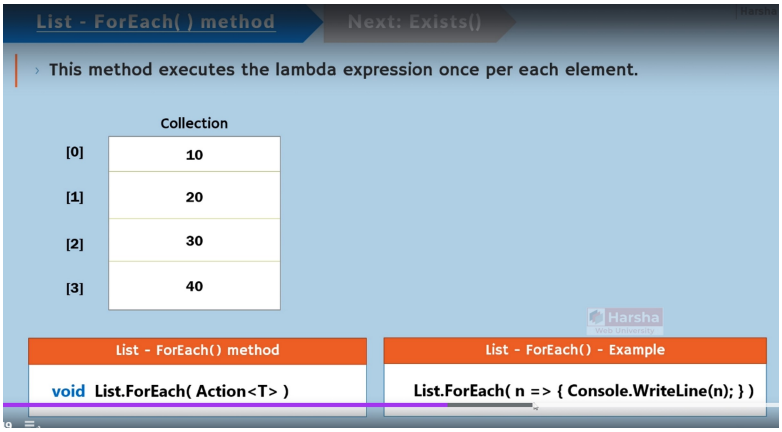
- This method sorts the collection in ascending order.



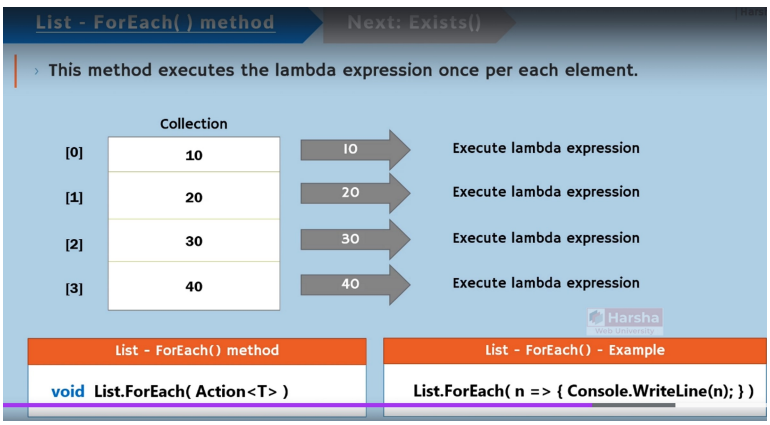
List - Sort() method	List - Sort() - Example
void List.Sort( )	List.Sort( )







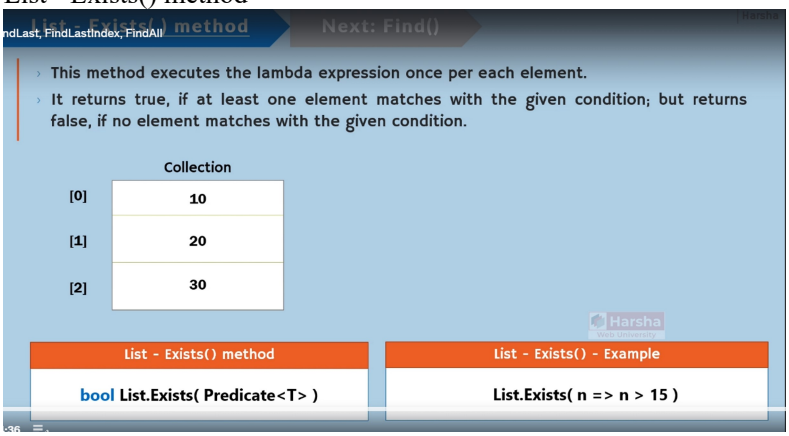
Action is a pre-defined delegate that returns void and takes an argument. It accept lamda expression or annynomous function.

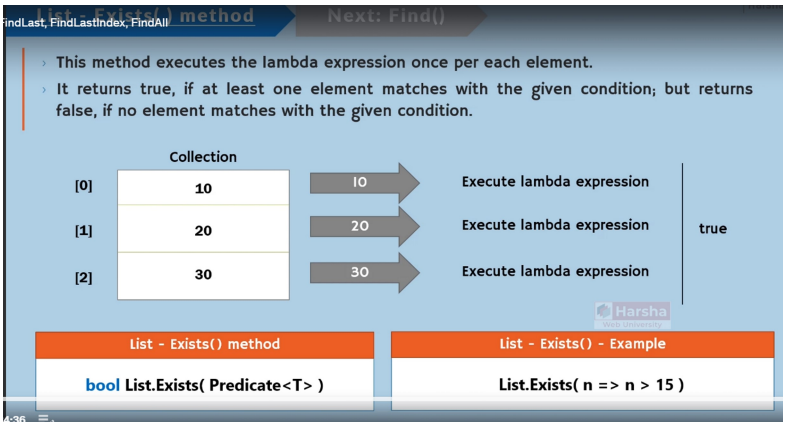


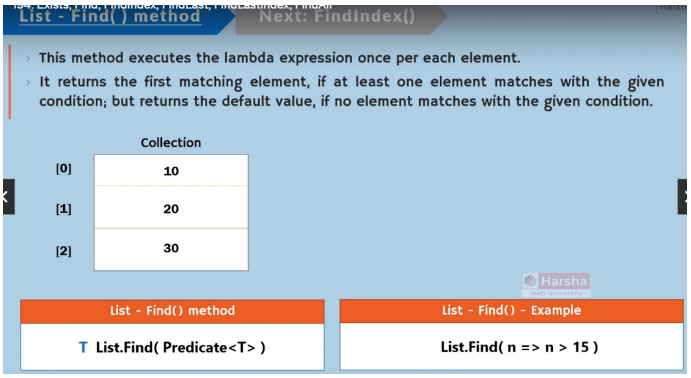
Exist, Find, FindIndex, FindLast, FindLastIndex, FindAll

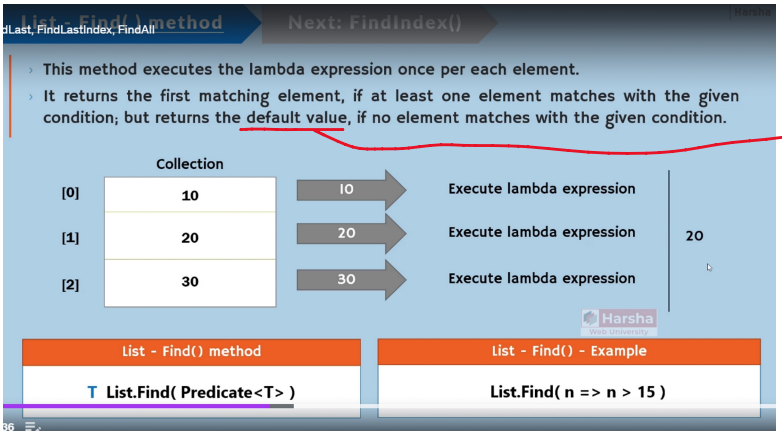
all these methods are provided by List Class.

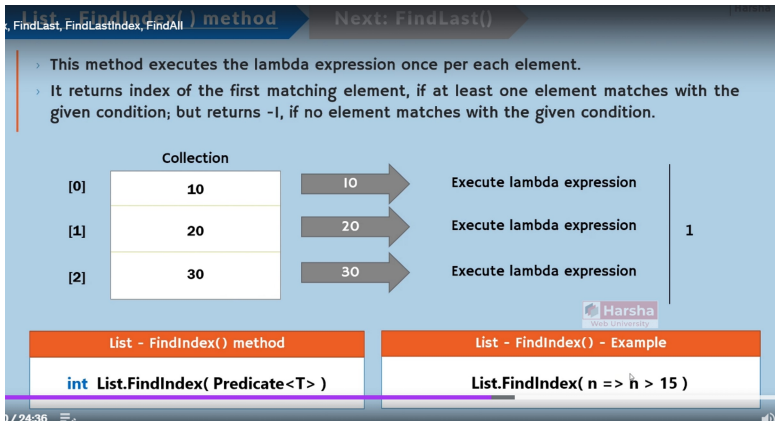
List - Exists() method

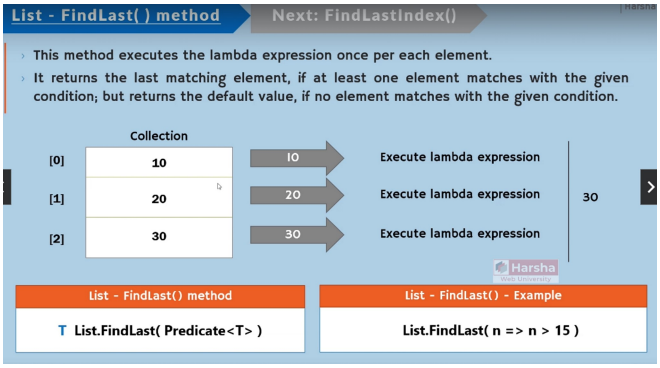


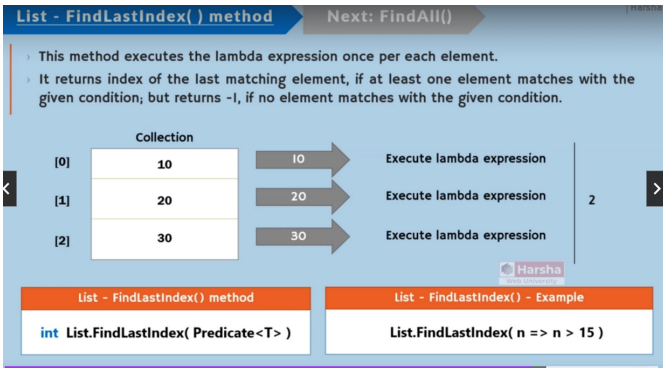


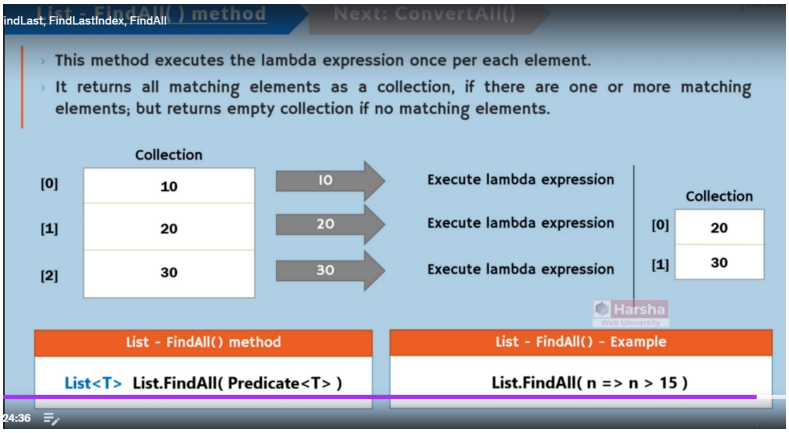


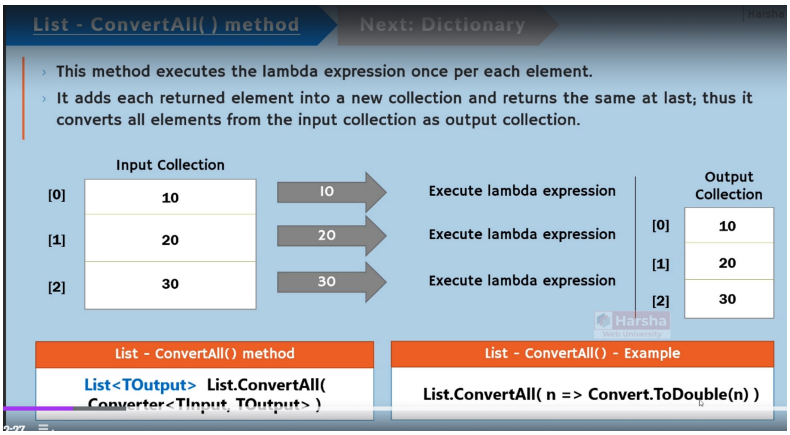

List - Find() method
Next: FindIndex()
This method executes the lambda expression once per each element.
It returns the first matching element, if at least one element matches with the given condition; but returns the default value, if no element matches with the given condition.
Collection
[0] 10
[1] 20
[2] 30
10
20
30
Execute lambda expression
Execute lambda expression
Execute lambda expression
20
Harsha
List - Find() method
T List.Find(Predicate<T>)
List - Find() - Example
List.Find(n => n > 15)

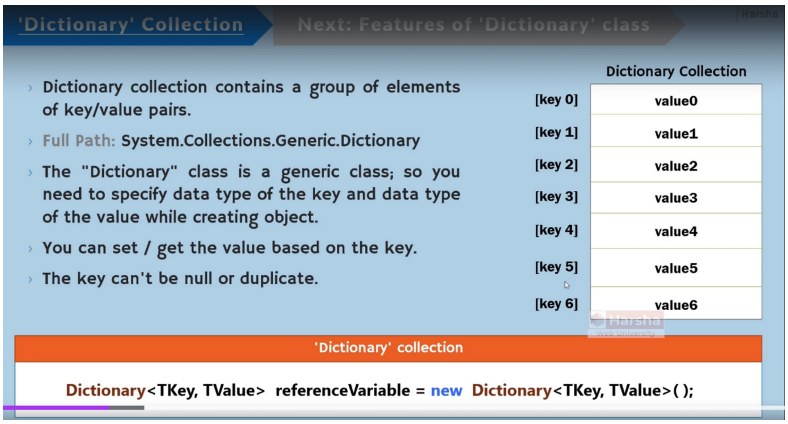

List - FindIndex() method
Next: FindLast()
This method executes the lambda expression once per each element.
It returns index of the first matching element, if at least one element matches with the given condition; but returns -1, if no element matches with the given condition.
Collection
[0] 10
[1] 20
[2] 30
10
20
30
Execute lambda expression
Execute lambda expression
Execute lambda expression
1
Harsha
List - FindIndex() method
int List.FindIndex(Predicate<T>)
List - FindIndex() - Example
List.FindIndex(n => n > 15)


List - FindLast() method
Next: FindLastIndex()
This method executes the lambda expression once per each element.
It returns the last 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
10
20
30
Execute lambda expression
Execute lambda expression
Execute lambda expression
30
Harsha
List - FindLast() method
T List.FindLast(Predicate<T>)
List - FindLast() - Example
List.FindLast(n => n > 15)

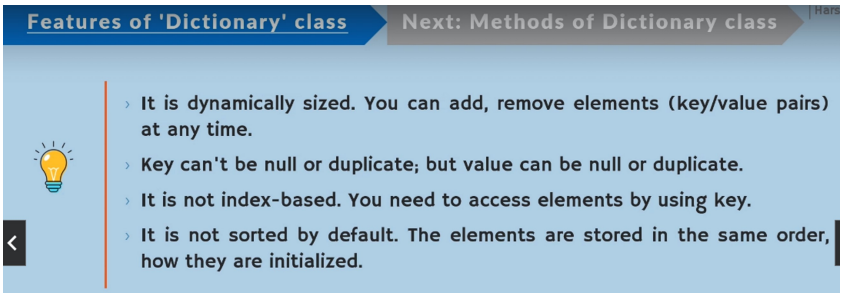

List - FindLastIndex() method
Next: FindAll()
This method executes the lambda expression once per each element.
It returns index of the last matching element, if at least one element matches with the given condition; but returns -1, if no element matches with the given condition.
Collection
[0] 10
[1] 20
[2] 30
10
20
30
Execute lambda expression
Execute lambda expression
Execute lambda expression
2
Harsha
List - FindLastIndex() method
int List.FindLastIndex(Predicate<T>)
List - FindLastIndex() - Example
List.FindLastIndex(n => n > 15)

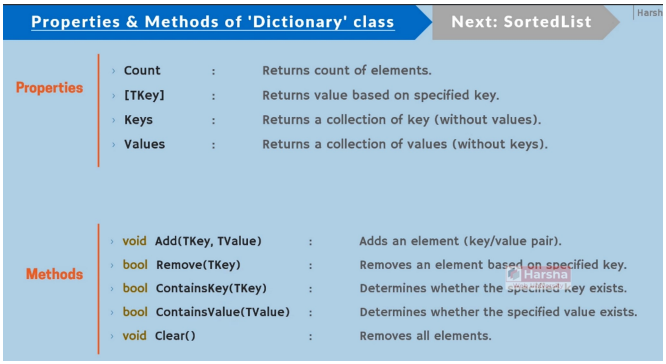


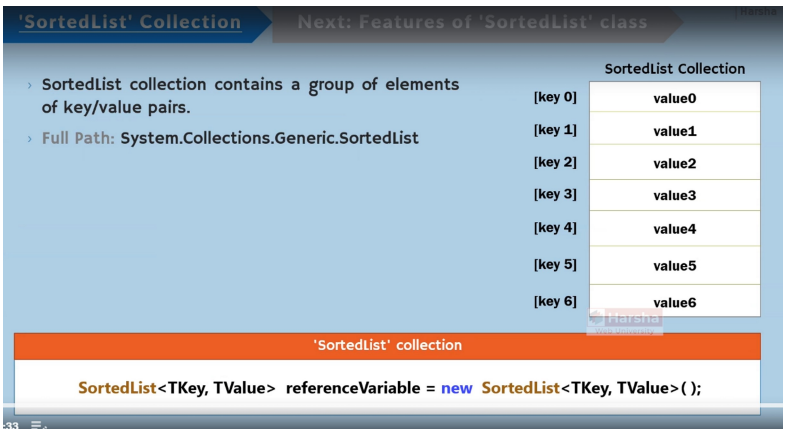




Group of key-value pair is called as Dictionary.







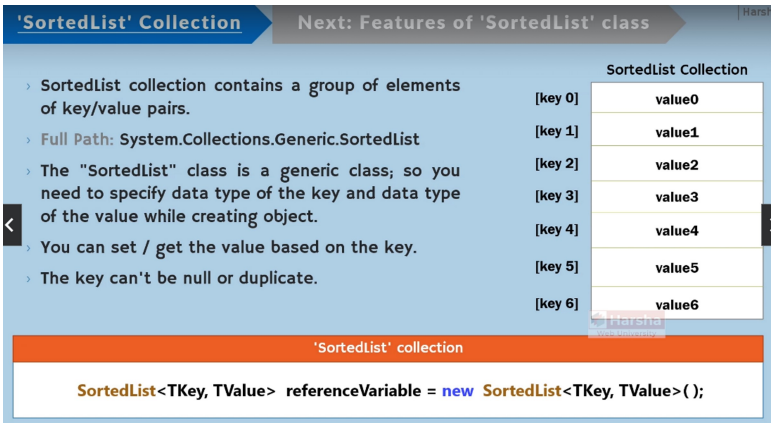
SortedList and Dictionary are similar.

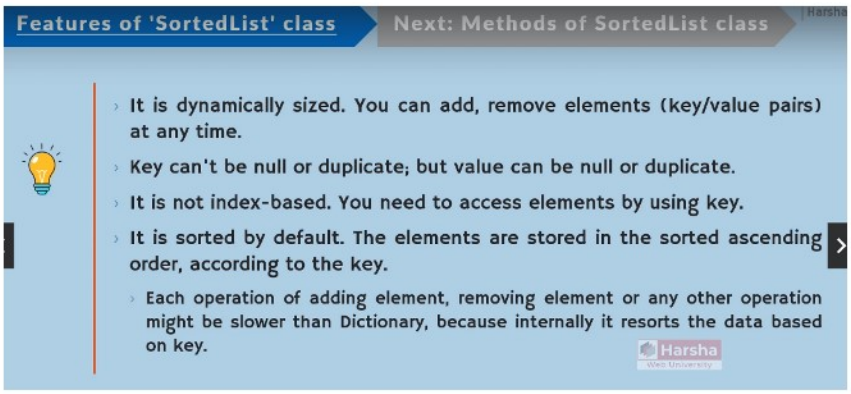
When you will use SortedList over Dictionary?

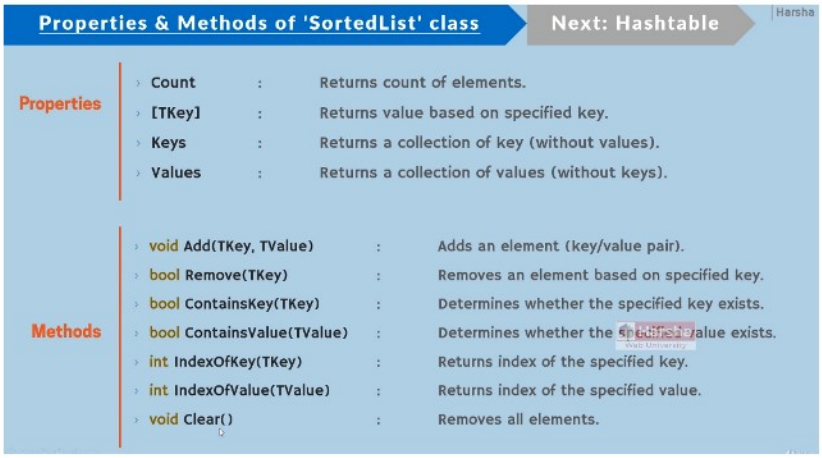
In case of large amount of data, ex: millions of records, use sortedList as it keeps all the record in sorted order whereas dictionary keeps data as they are stored.

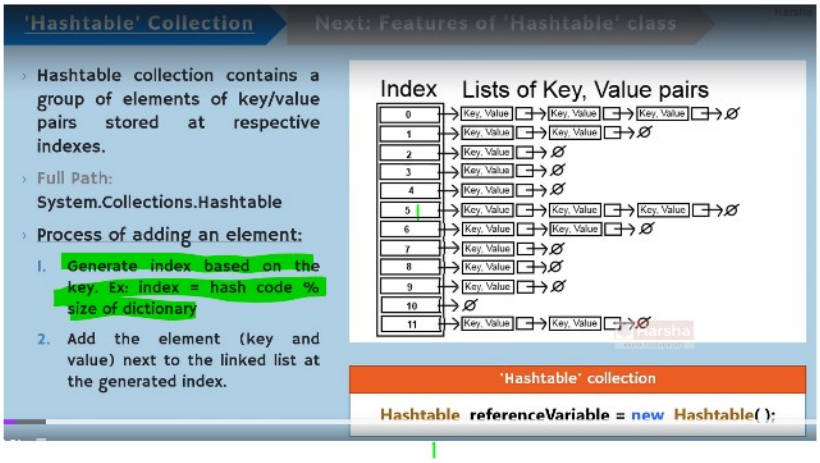
Dictionary performs Linear Search.

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



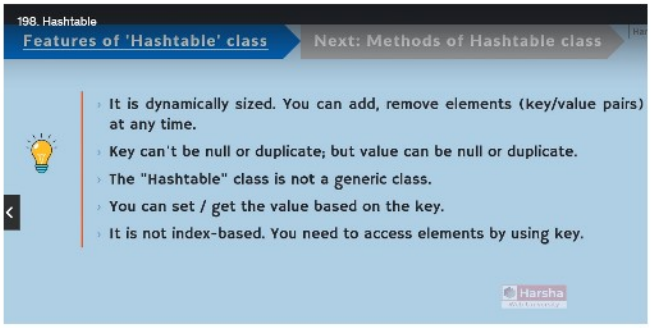


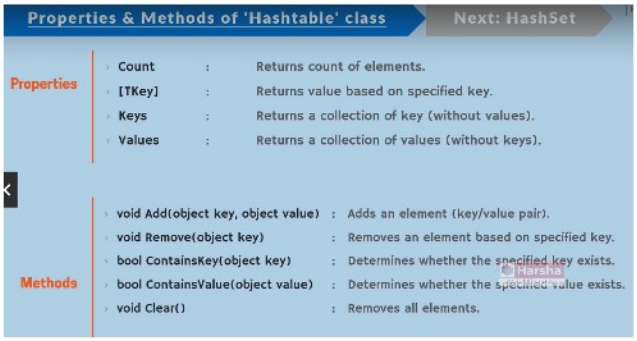




In case of string, system will generate the hash code, in case of number, same number will be used as a hash code.

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.





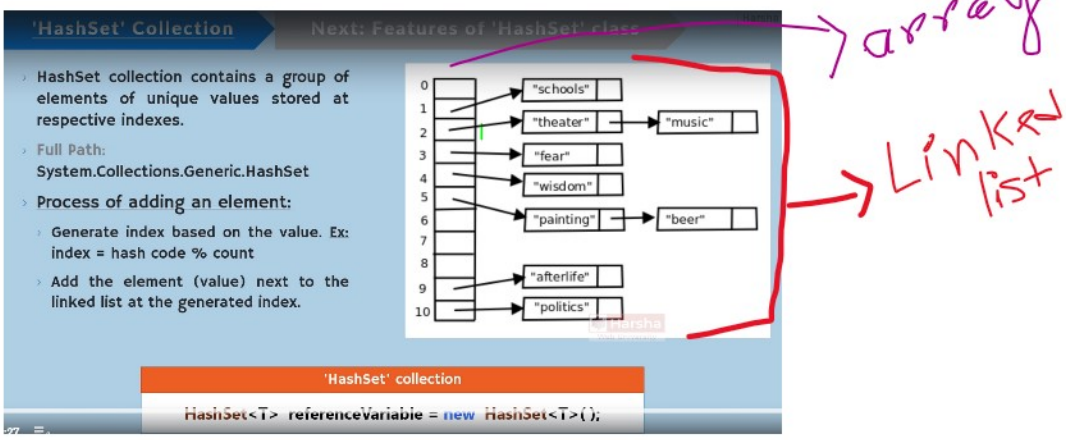
HashSet

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.

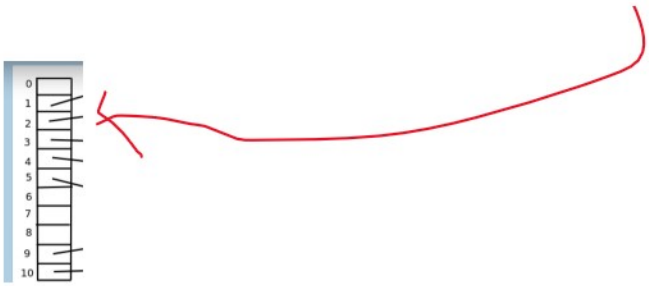
In hash-table, the index will be generated based on the hash code of the key right and you have unique key here.

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



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

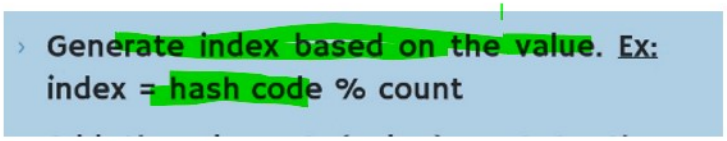
Here the capacity means the size of the internal array .



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

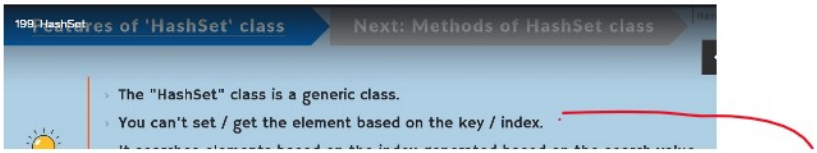
```
using System;

Class Object{
  getHashCode();
}

class String extends Object{

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

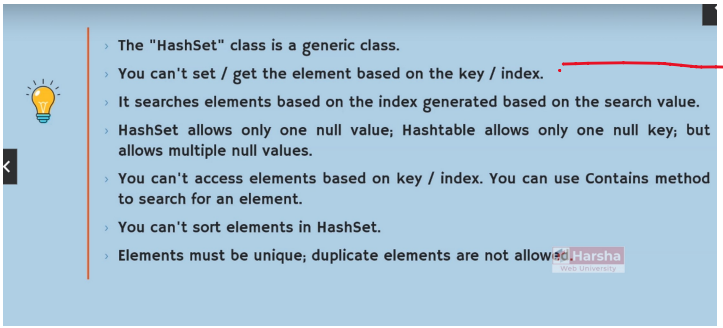
In case of numerical type, the same number will be treated as hash-code.



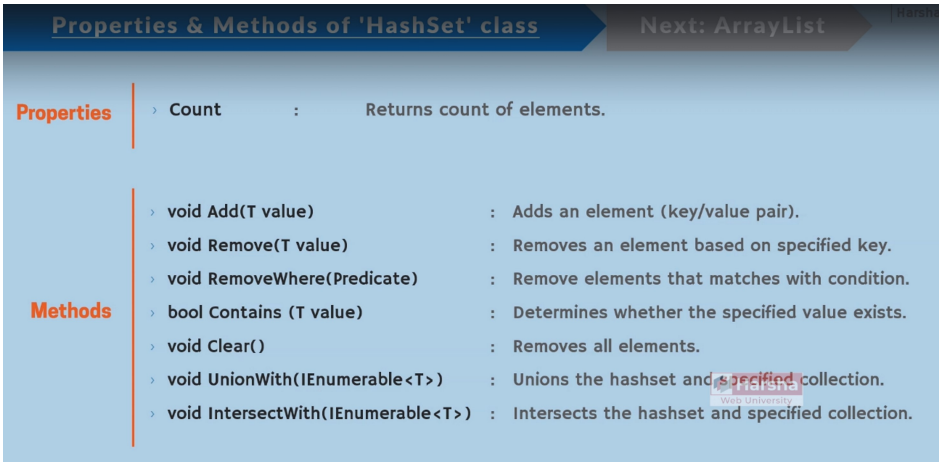
suppose you are looking for the particular value called painting so based on the value
called painting an index will be generated at that moment
and it goes to the particular index directly instead of searching all the previous elements
in the form of linear search so that is the reason it's extremely
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

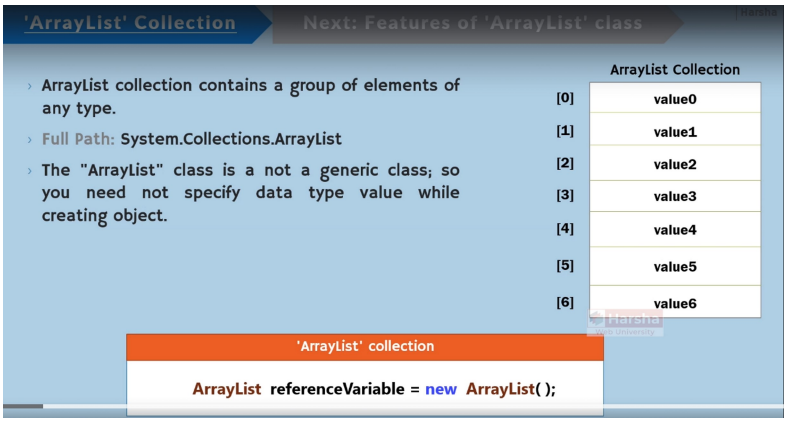


so in real time applications when you don't want to store the keys but mainly you want the collection to determine whether this particular value exists or not
for that cases we can store those set of elements in the hash set collection but alternatively you want to get the exact element
based on the 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

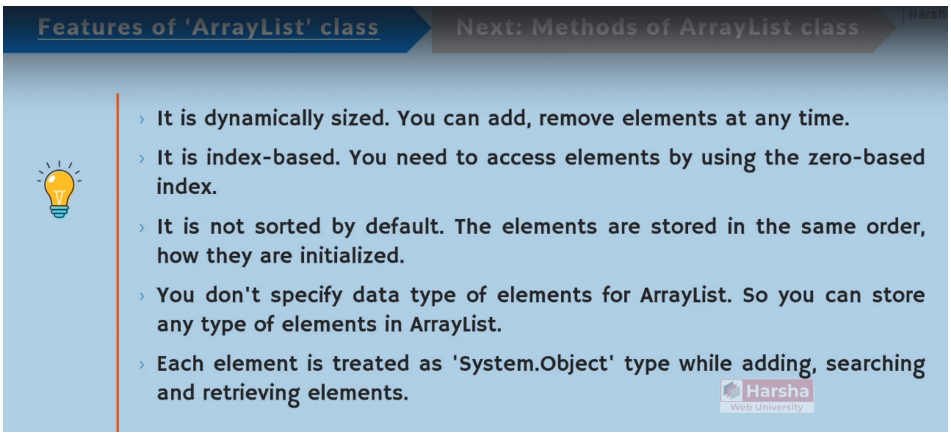
in case if you are performing adding and removing process less number of times but mainly you are trying to retrieve elements by using for each loop or based
on the key in that case prefer using sorted list

ArrayList



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***



Properties and methods are most similar to List class.

Properties & Methods of 'ArrayList' class

Next: Stack

Properties

- Count
- Capacity

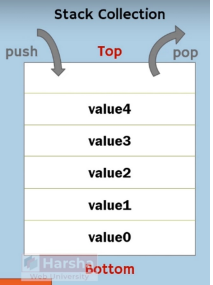
Methods

- Add(object)
- AddRange(ICollection)
- Insert(int, object)
- InsertRange(int, ICollection)
- Remove(object)
- RemoveAt(int)
- RemoveRange(int, int)
- Clear()
- IndexOf(object)
- BinarySearch(object)
- Contains(object)
- Sort()
- Reverse()
- ToArray()

'Stack' Collection

Next: Features of 'Stack' class

- Stack collection contains a group of elements based on LIFO (Last-In-First-Out) based collection.
- Full Path: System.Collections.Generic.Stack
- It stores the elements in bottom-to-up approach.
  - Firstly added item at bottom.
  - Lastly added item at top.



'Stack' collection

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

Features of 'Stack' class

Next: Methods of Stack class

- It is based on LIFO (Last-In-First-Out).
- The "Stack" class is a generic class; so you need to specify data type of elements while creating object.
- You can't access elements based on index.
- You can add elements using 'Push' method; remove elements using 'Pop' method; access last element using 'Peek' method.
- It is not index-based. You need to access elements by either using pop, peek or foreach loop.

Properties & Methods of 'Stack' class

Next: Queue

Properties

- Count : Returns count of elements.

Methods

- void Push(T) : Adds an element at the top of the stack.
- T Pop() : Removes and returns the element at top of stack.
- T Peek() : Returns the element at the top of the stack.
- bool Contains(T) : Determines whether the specified element exists.
- T[] ToArray() : Converts the stack as an array.
- void Clear() : Removes all elements.

'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' collection

```
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

Properties

Count	:	Returns count of elements.

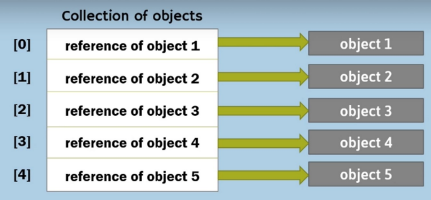
Methods

void Enqueue(T)	:	Adds an element at the top of the queue.
T Dequeue()	:	Removes and returns the element at top of queue.
T Peek()	:	Returns the element at the top of the queue.
bool Contains(T)	:	Determines whether the specified element exists.
T[] ToArray()	:	Converts the queue as an array.
void Clear()	:	Removes all elements.

Collection of Objects

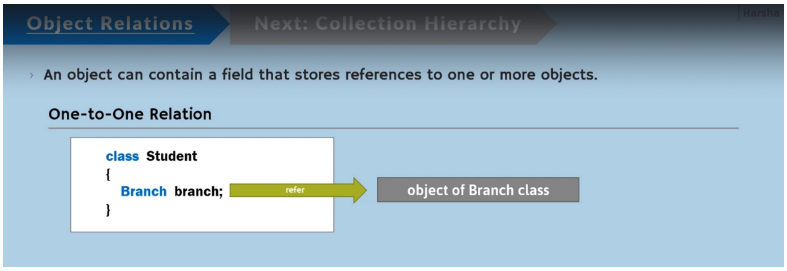
Next: Object Relations

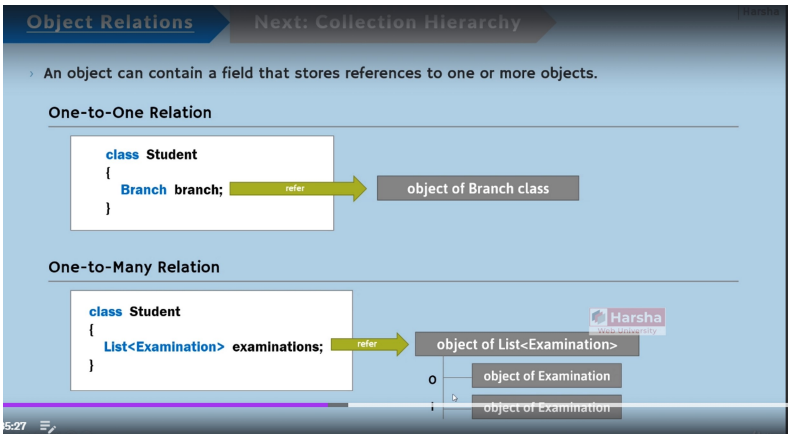
- 'Collection of objects' is an collection object, where each element stores a reference to some other object.
- Used to store details of groups of people or things.

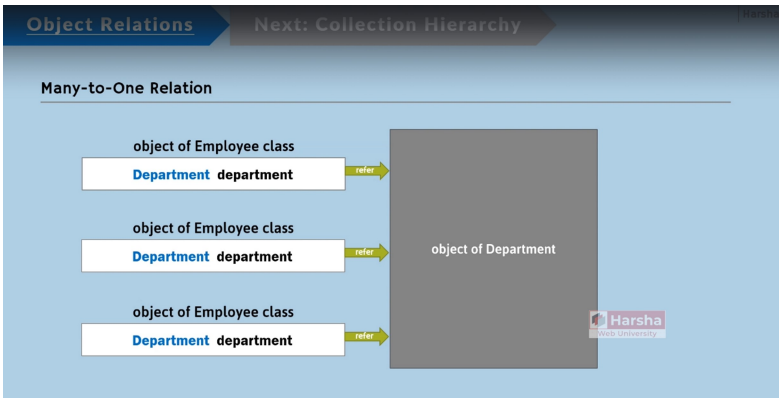


Collection of Objects

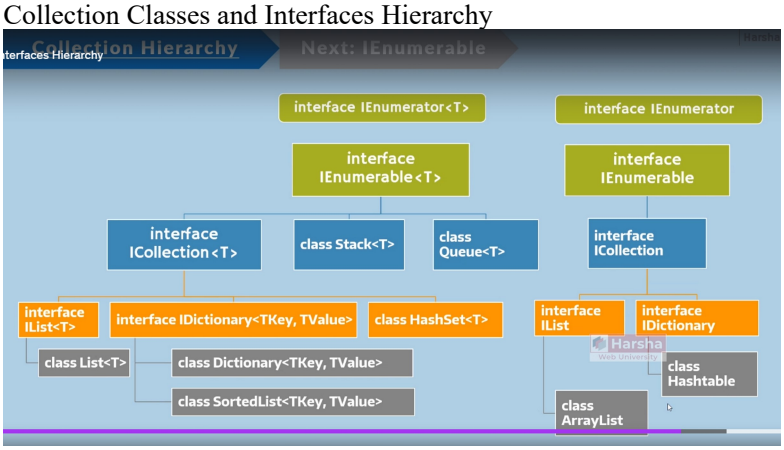
```
List<ClassName> referenceVariable = new List<ClassName>();
referenceVariable.Add(object1);
referenceVariable.Add(object2);
referenceVariable.Add(object3);
...
```







Collection Classes and Interfaces Hierarchy

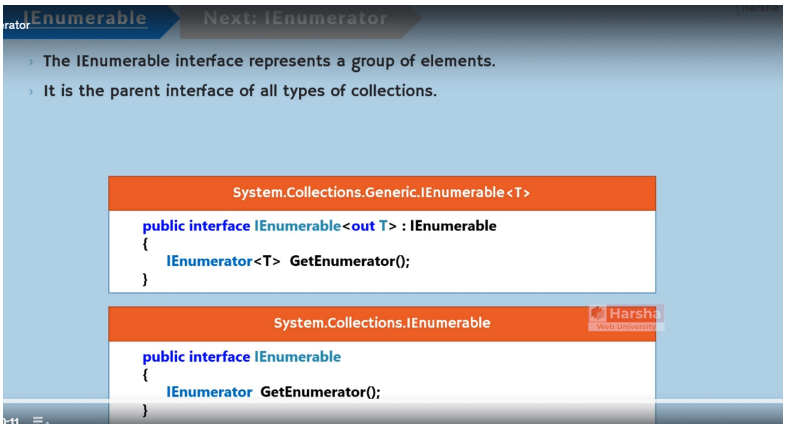


From developer point of view, remembering this hierarchical structure is very important not only for understanding purposes but also creating reference variable for the parent interface

instead of child class reference.

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

IEnumerable and Ienumerator

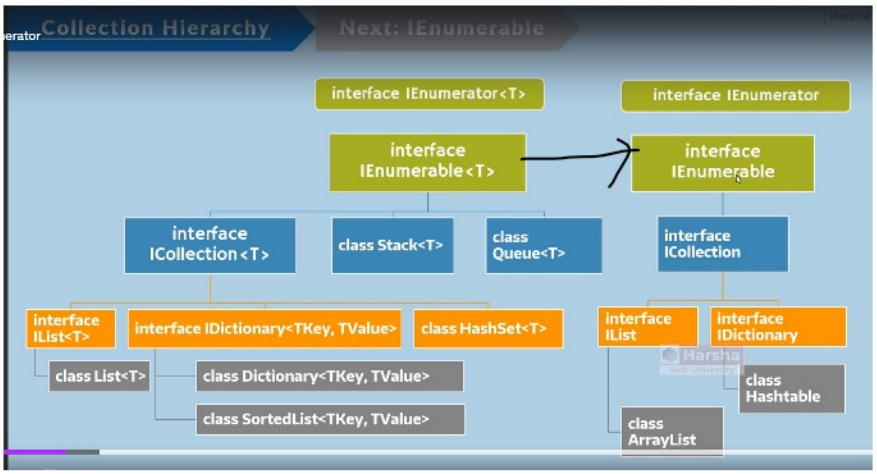


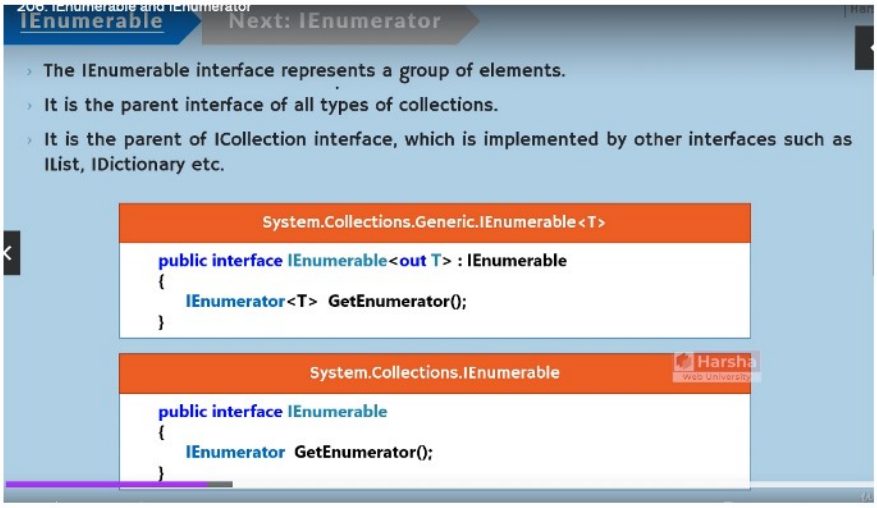
Actually IEnumerable and IEnumerator are not comparable both are two individual interfaces and one is not the alternative of other each has its own purpose and usage
but for a developer it is necessary to understand both of these interfaces and when and how to use each one
so this lecture is about how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

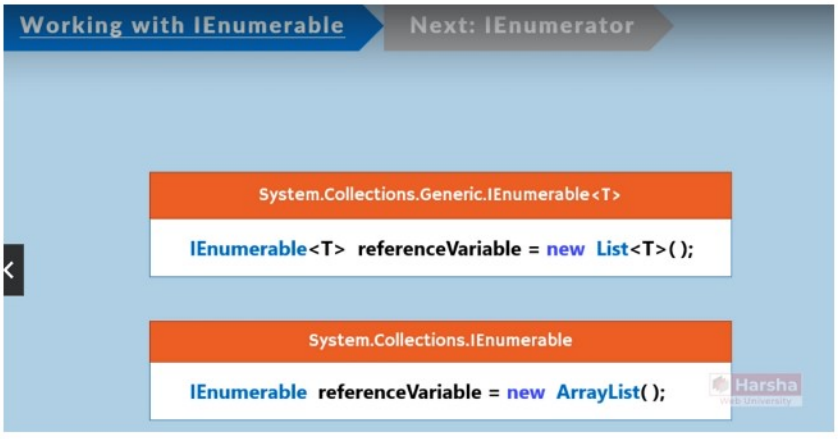
as you can see both of these interfaces are the predefined interfaces some what related to collections but not
only limited to collections both of these are for arrays also see first we will consider understanding
about IEnumerable then we can talk about IEnumerator first of all there is an interface called IEnumerable under the system.collections namespace it has only one method called numerator 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

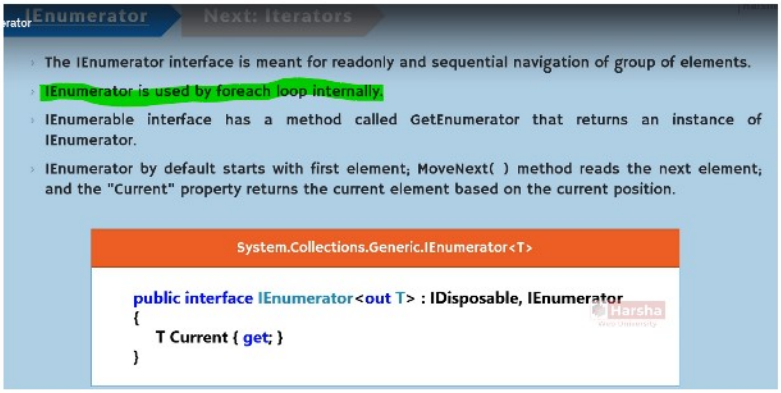
or anything else but the problem is that the default IEnumerable interface is not generic interface so it is not
possible to specify the data type that means it don't have any generic type here.
than so that is the reason in order to provide the ability to have generic collection classes there is an
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

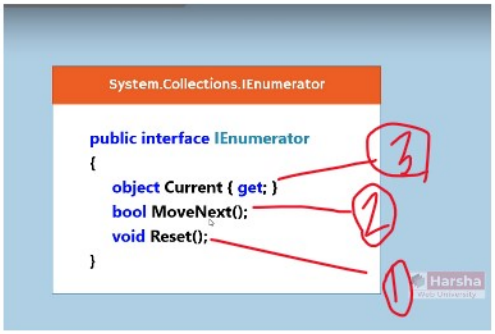
int string or any other custom
class while working with collections such as list, dictionary etc so all the other collection interfaces and classes
are directly or indirectly the children of IEnumerable as shown in the previous diagram so as you can see ICollection, IDictionary, IList and the classes such as hash set stack queue list dictionary etc are the children of IEnumberable which is ultimately the child of IEnumerable which is the non-genetic version.











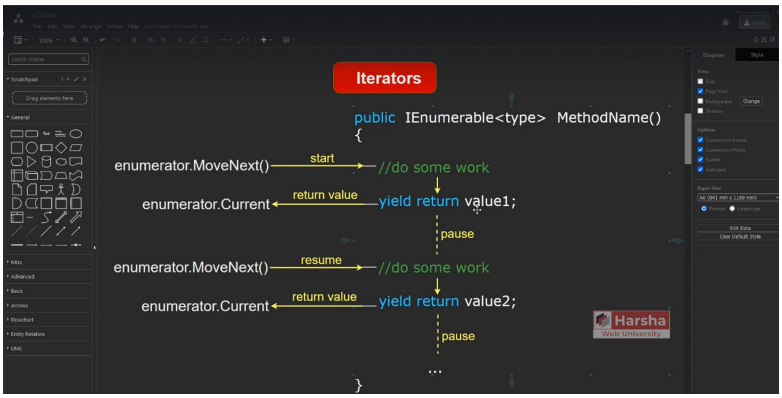
do you know the for each loop works based on IEnumerator the for each loop by default starts from
the beginning of the group that means beginning of the collection and goes through each and every element
in a sequential order right internally it works based on a numerator actually the IEnumerator has 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 foeach 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 each loop works.

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

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

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

IEnumerator is made for making foreach work.

Iterator and Yield Return

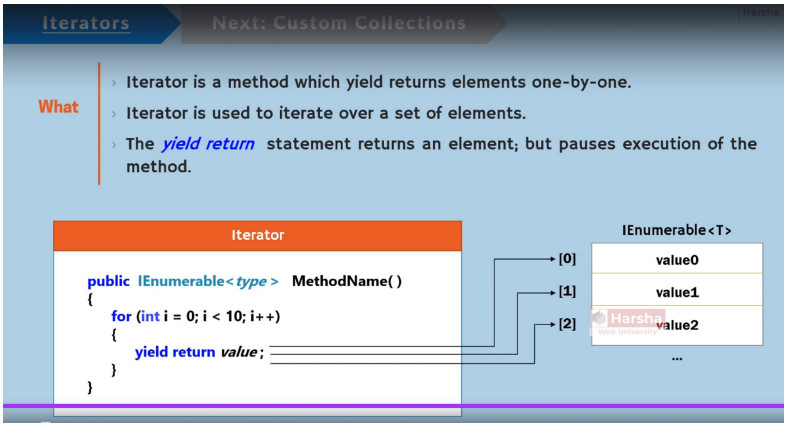


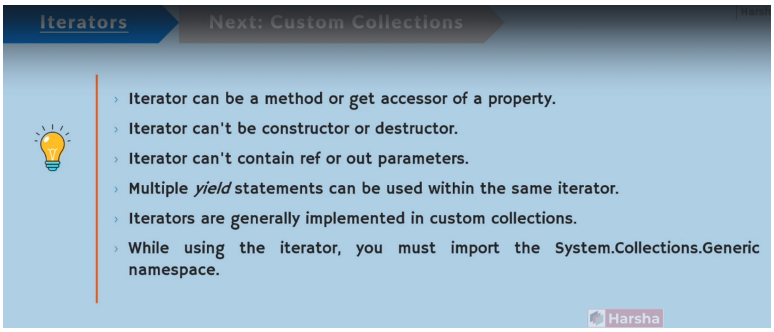
The Iterator is a special type of method in C#
Generally when you call the method the complete body of the method executes and at last it returns a value optionally but iterator's return type will be innumerable of particular type

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.

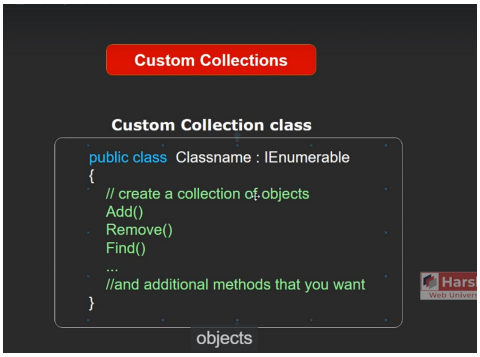
yield return statement returns a value but continues the execution of that method.





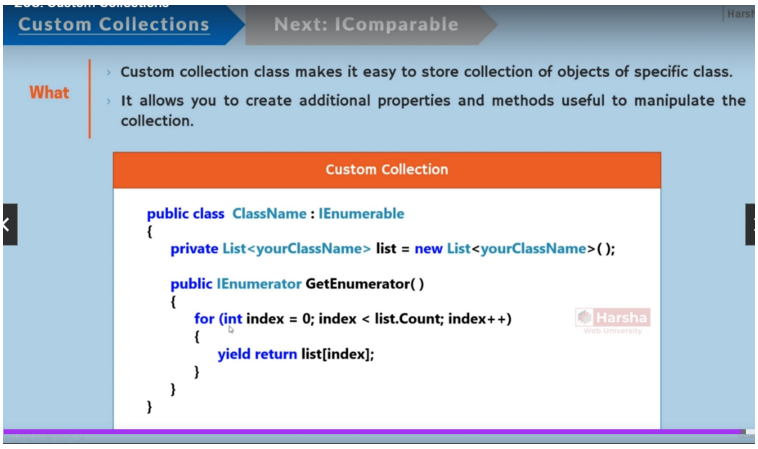
suppose you have created a collection of objects maybe collection of customers or employees whatever required
suppose you would like to add a specific set of methods that are applicable for that particular type of collection
so these additional methods might manipulate these collection of objects let's 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

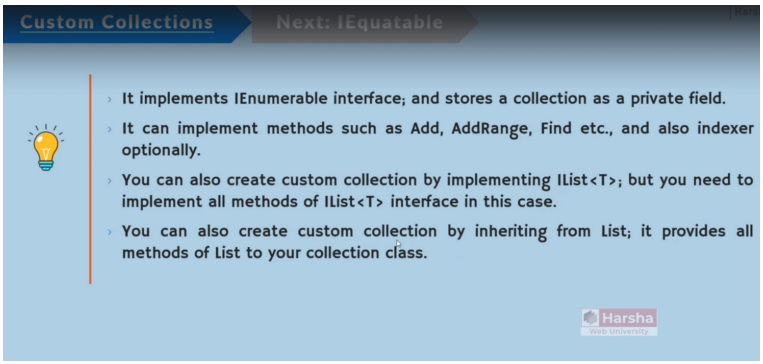
requirement is you would like to
add those type of methods in this class and that class is called as custom collection class
so a custom collection class is a class that contains a collection of objects for example list of customer class
here customer is the name of the user defined class it can be any 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.



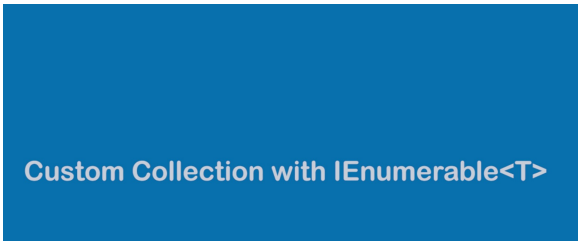
If you want to check for example your customer id should begin with capital a or any other condition it cannot be
checked by the predefined add method and that is what you can add validation logic here and that is what the purpose or reason why we create custom collections.

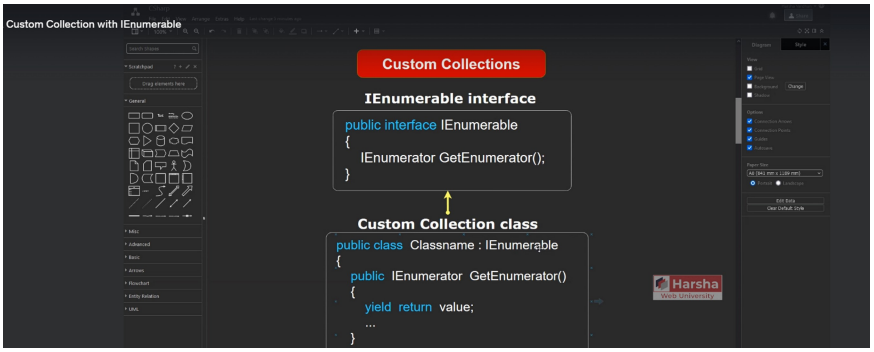
so if you ask me why we have to create custom collection class why can't we simply use the list of customer like
this because it gives you more grip on the methods you can add extra methods like for example search or search by 4
number etc which are not exist in the predefined collection classes and 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



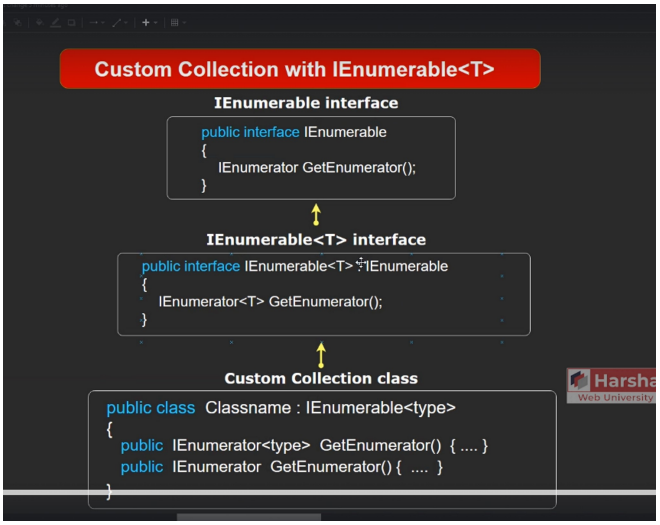


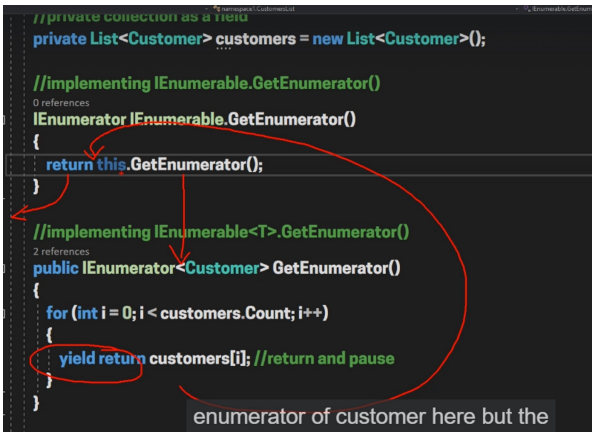
Custom Collections with IEnumerable





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



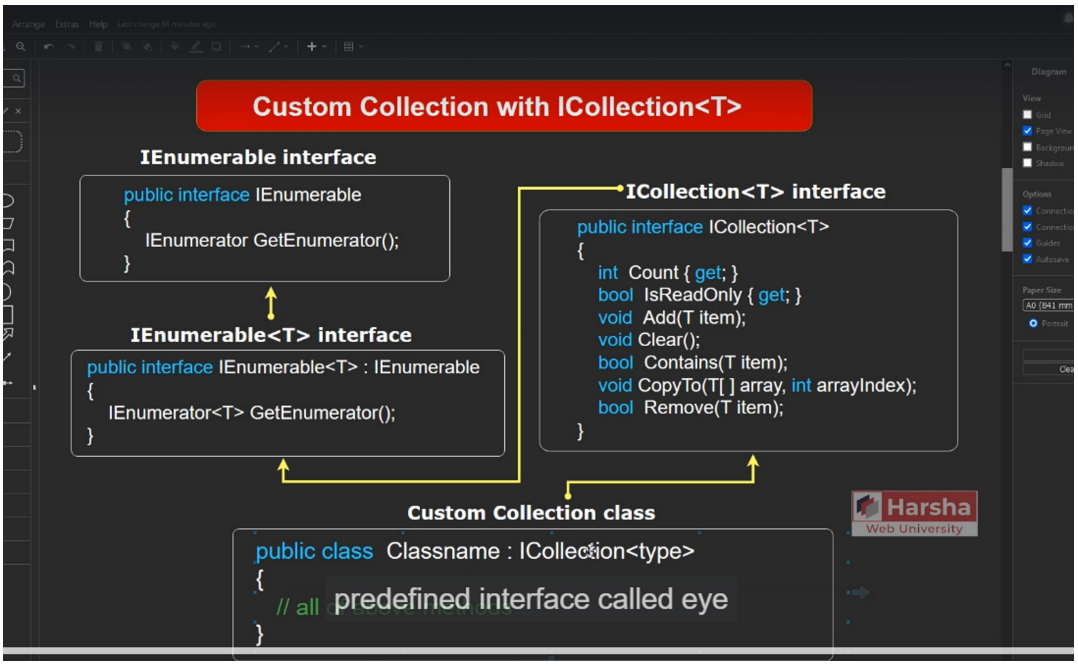


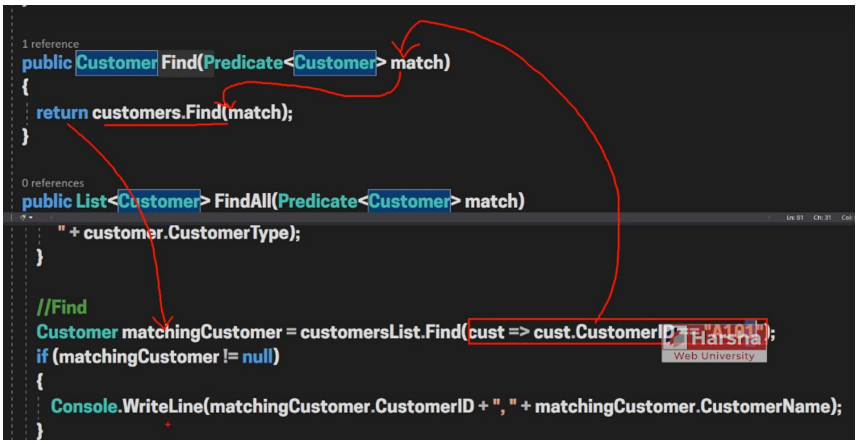
but the problem with Innumerable of T is that it doesn't provide any specific concrete methods like for example Add, Add Range, Insert find etc so in order to implement those methods
also in your class it is better to implement ICollection or IList

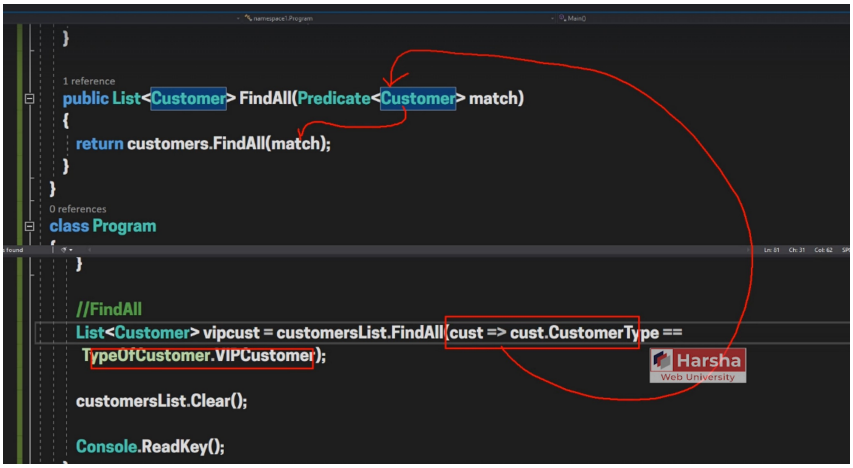
and we will try that in the next lecture you.

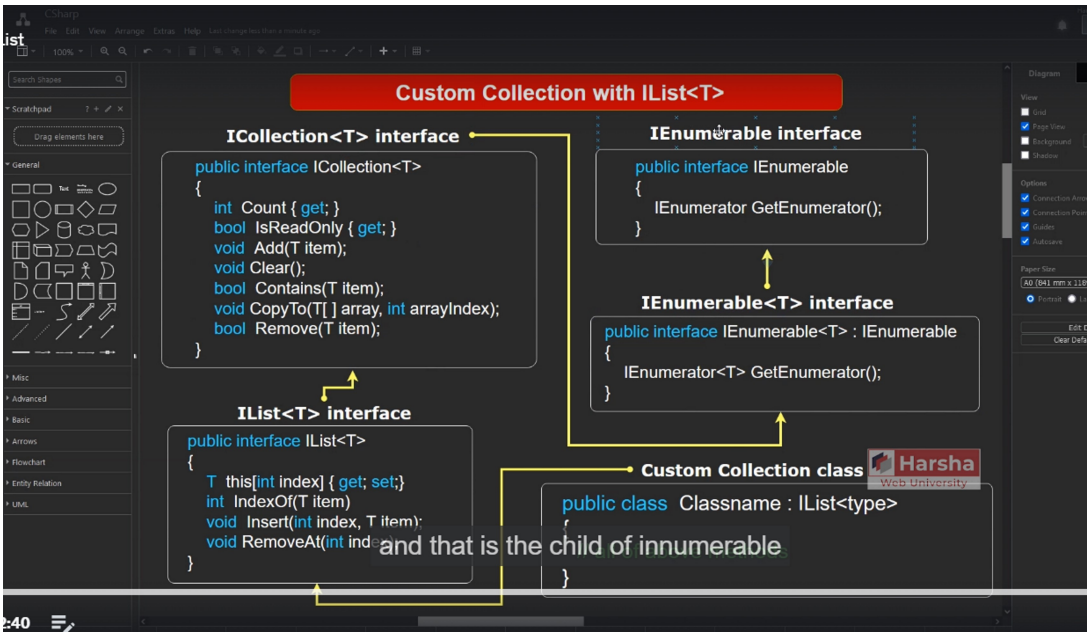
you can also try creating the custom collection class by implementing this predefined interface called ICollection
and this ICollection contains the methods that is add remove contents and other methods as you can see
so the benefit of implementing this interface is that in your class you need to implement all these methods
so that you can add custom logic in these methods in addition apart from these methods

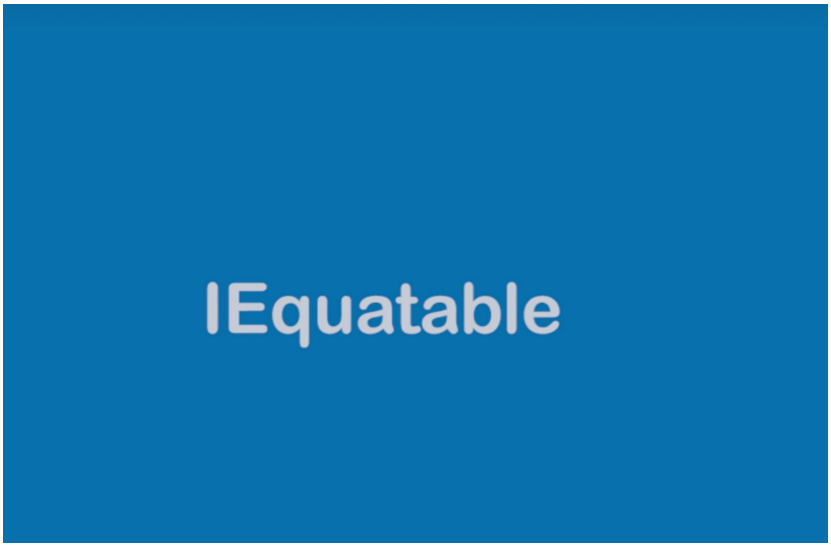
you can add additional methods whatever needed in your own custom class







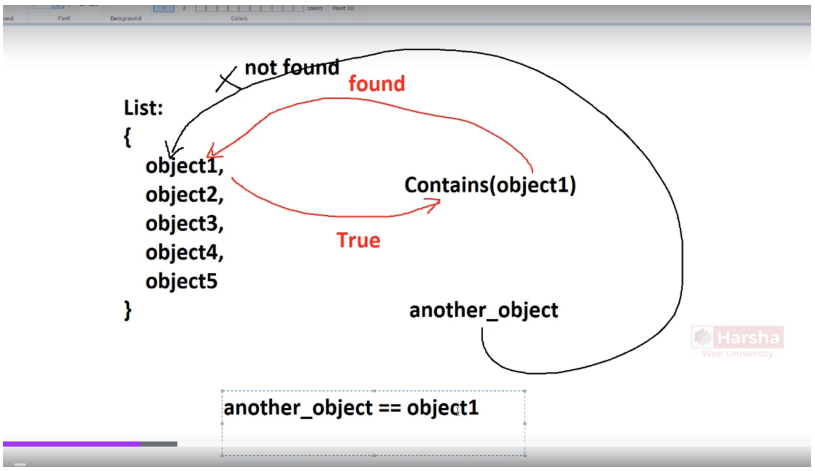


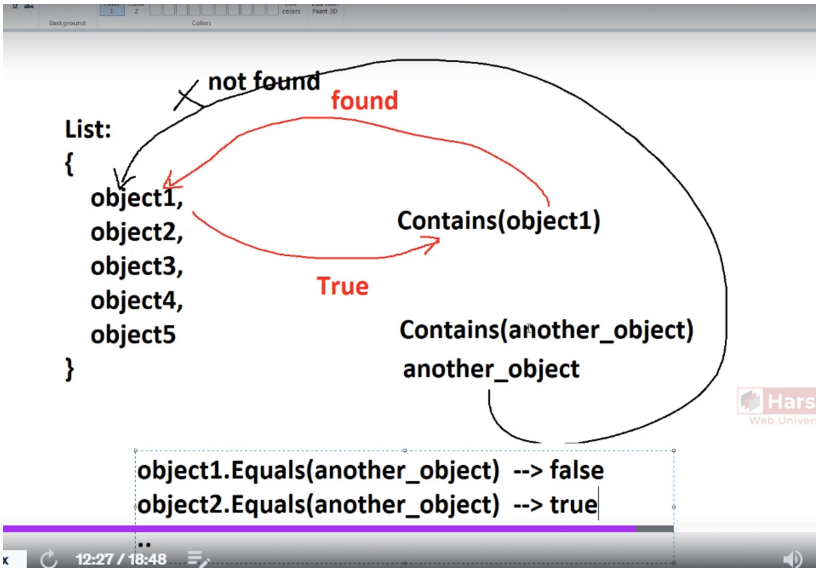


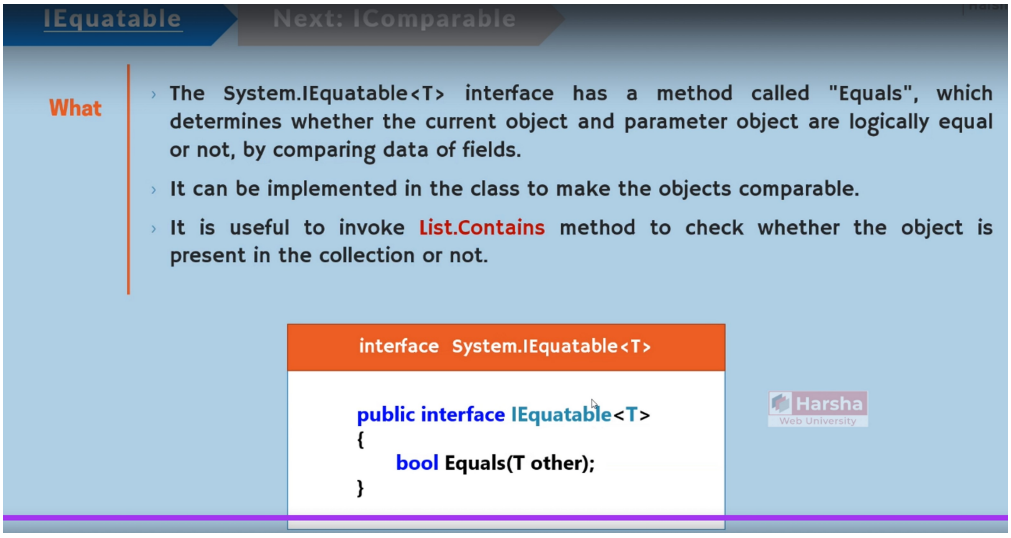
In the Ilist there is a predefined method called Contains so if you supply an object for this method it will set for that object in the list and if that object is already found in the list it returns true otherwise false but the problem with Contains method is it can identify the same object but cannot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



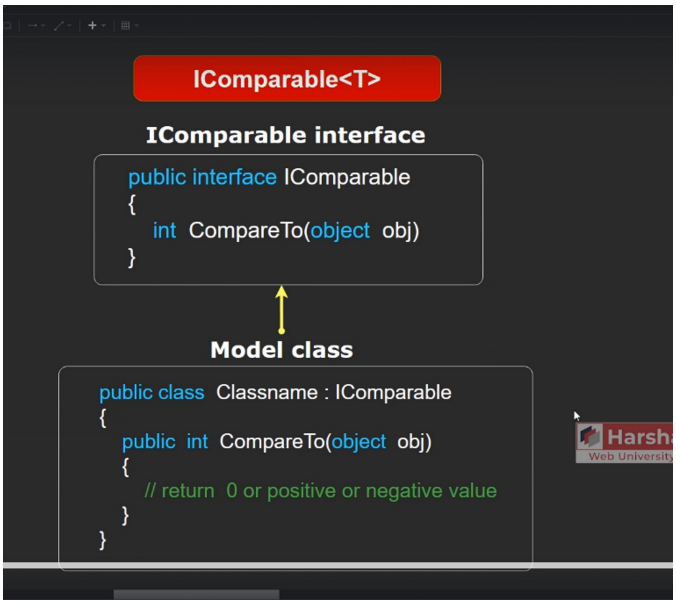


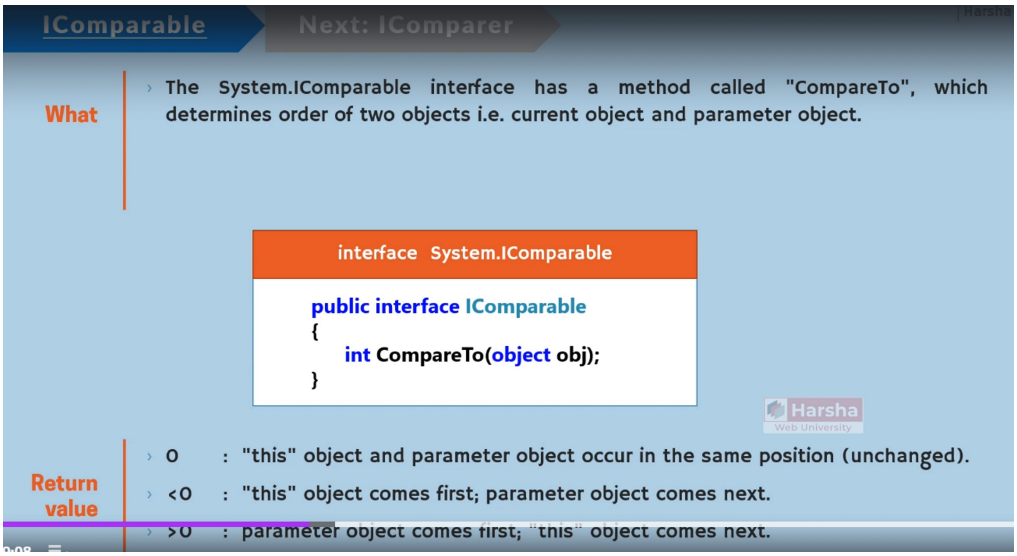


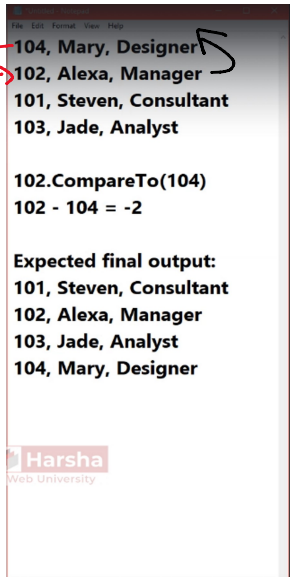
Implementation of IEquatable interface

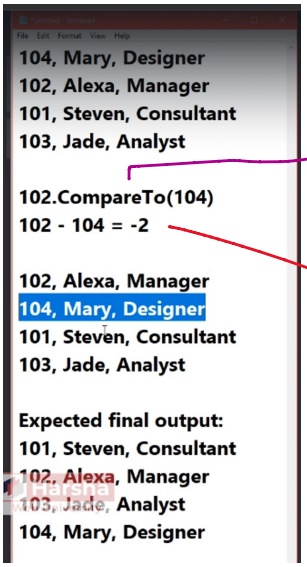
```
public class  ClassName : IEquatable<ClassName>
{
    public  bool  Equals(ClassName other)
    {
        return  this.field1 == other.field1  &&  this.field2 == other.field2;
    }
}
```


IComparable



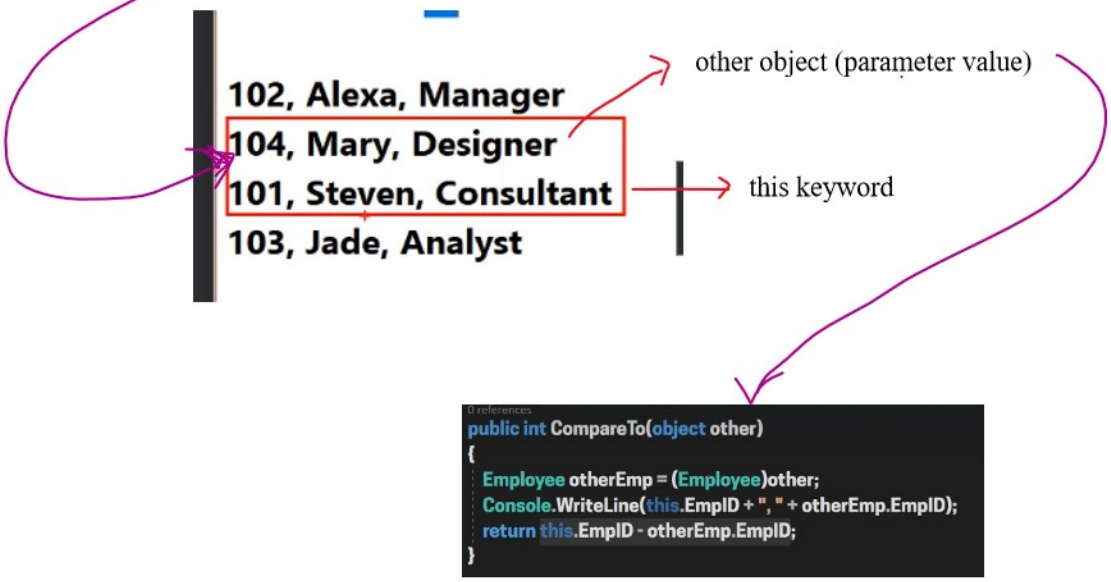




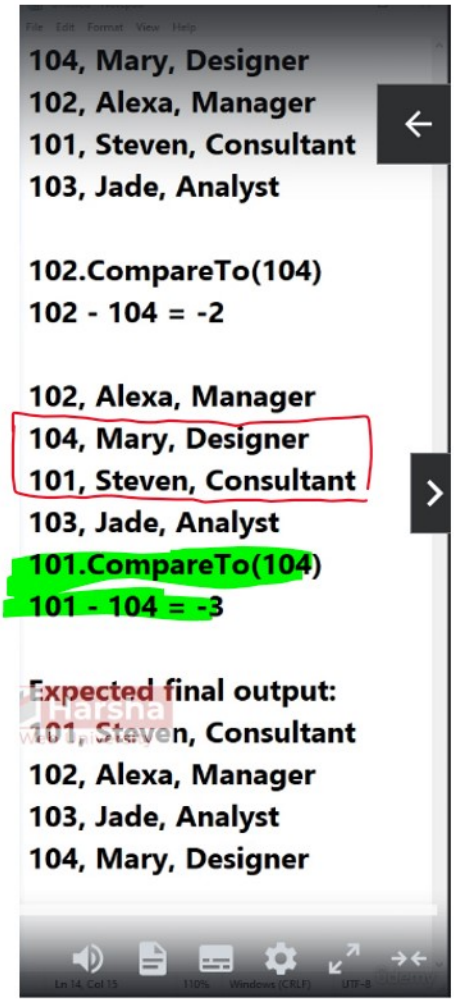



```
0 references
public int CompareTo(object other)
{
    Employee otherEmp = (Employee)other;
    Console.WriteLine(this.EmpID + ", " + otherEmp.EmpID);
    return this.EmpID - otherEmp.EmpID;
}
```

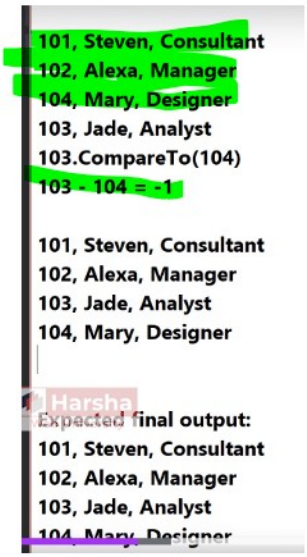
negative value will recall the ComapreTo() method with these two objects.

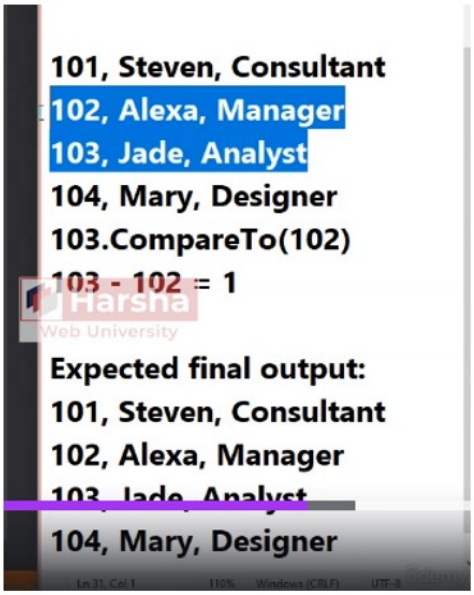



```
0 references
public int CompareTo(object other)
{
    Employee otherEmp = (Employee)other;
    Console.WriteLine(this.EmpID + ", " + otherEmp.EmpID);
    return this.EmpID - otherEmp.EmpID;
}
```




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


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

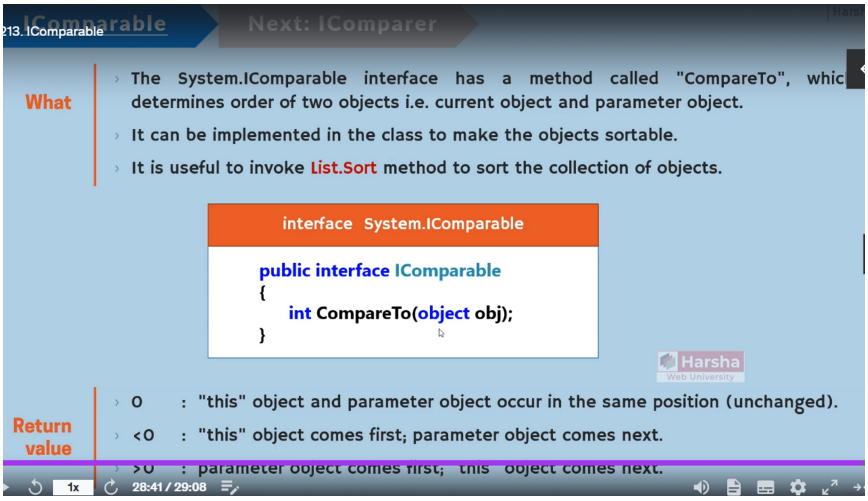
Ant

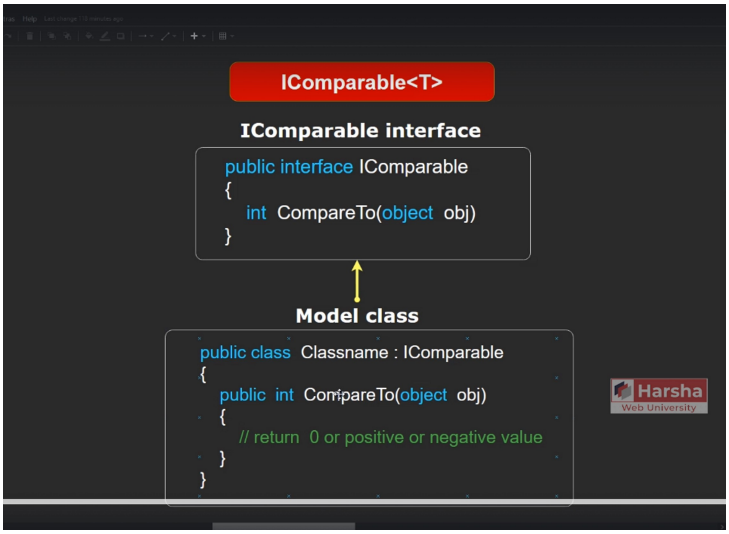
Bat

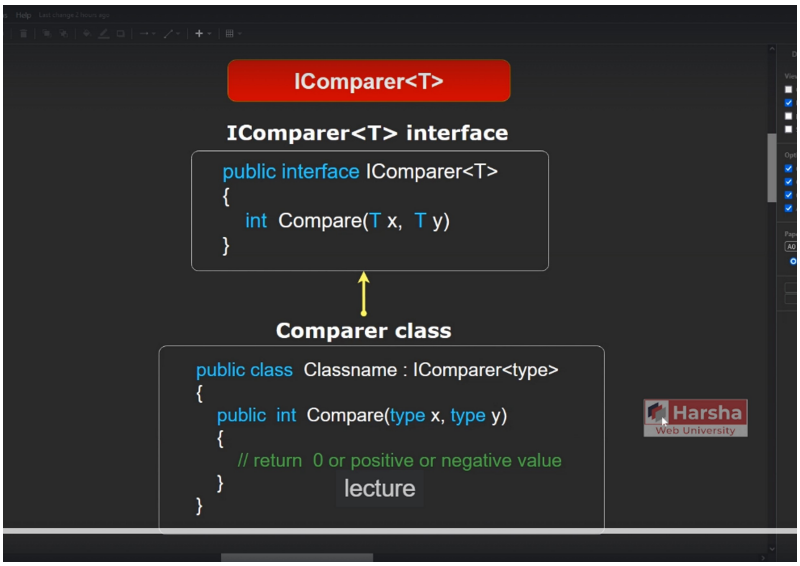
Ant.CompareTo(Bat) "A" should come first. So it returns -1

Apple.CompareTo(Ant) 1

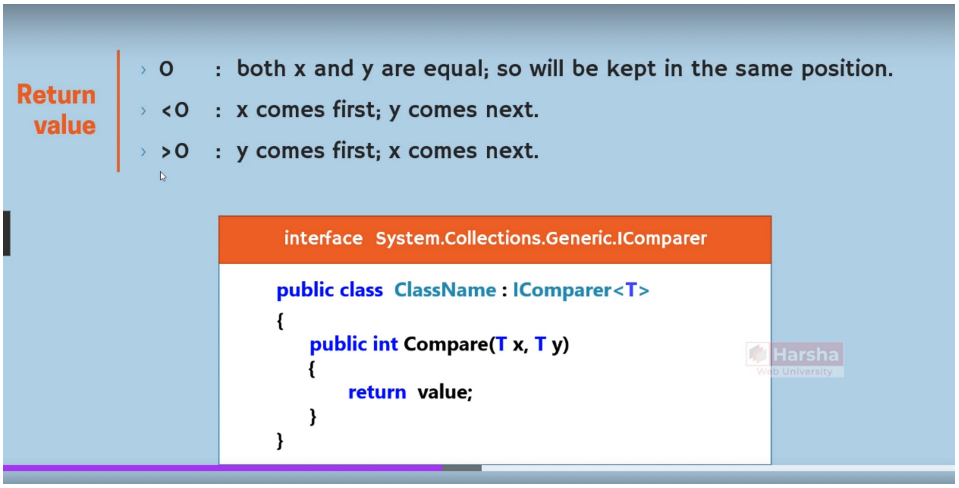
Apple.CompareTo(Apple) 0



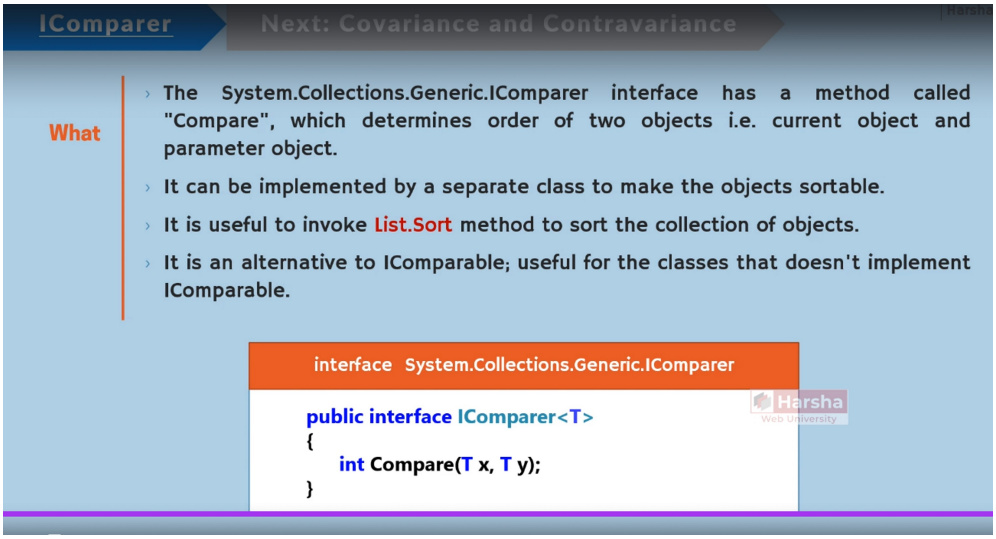




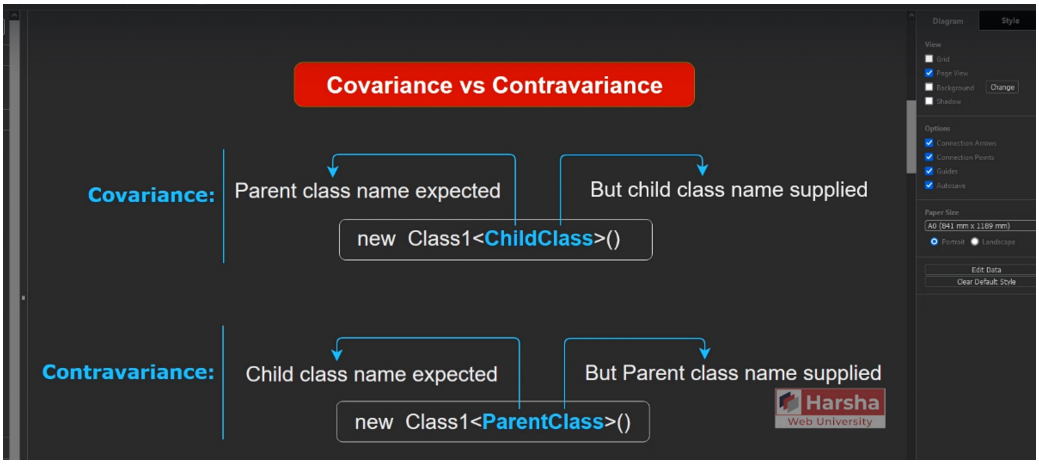
so whenever your model class like employee class implements IComparable interface you can sort that list of
employees we have already done the same earlier in the previous lecture but sometimes in the real world applications suppose a developer creates a model class but he does not implement a comparable 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.

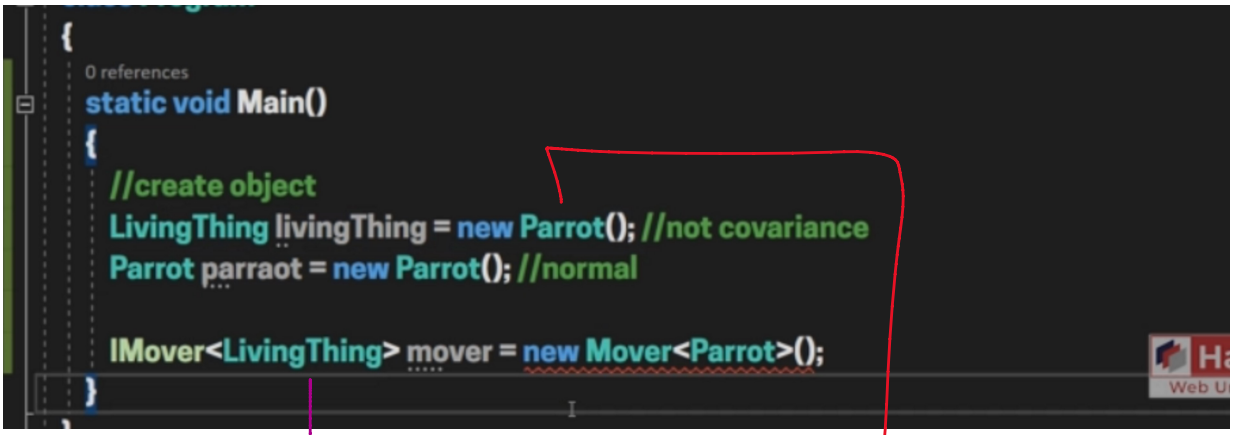


IComparer
Part 2



Covariance



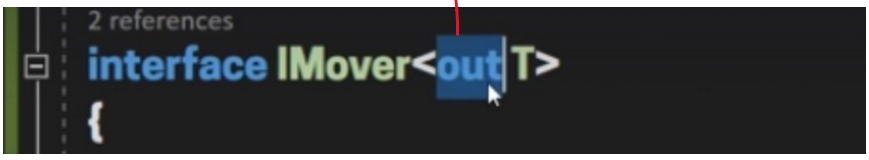


by defaut, it is ok.

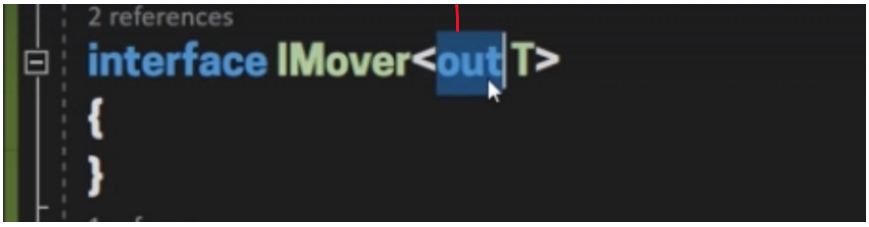
but this is not ok in generic parameter by default.

In order to solve this problem the covariance concept is introduced

solution:

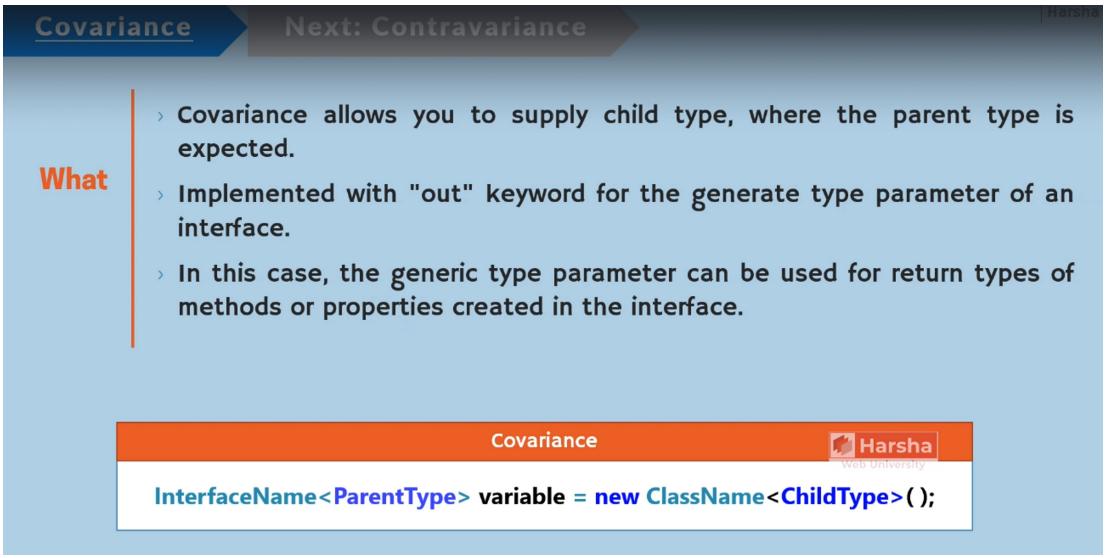


by adding 'out' keyword, it becomes 'covariance'. here out keyword says that, you can use this out keyword T only as a return type of a method or property. you can not use the same as argument type.

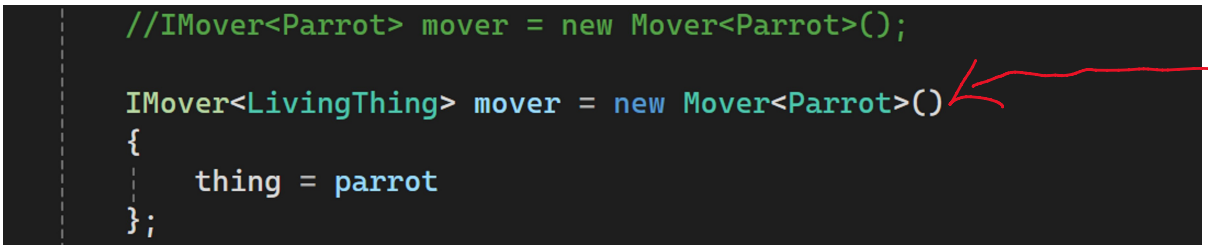


can use this out keyword T only as a return type of a method or property. you can not use the same as argument type.

Not Allowed!



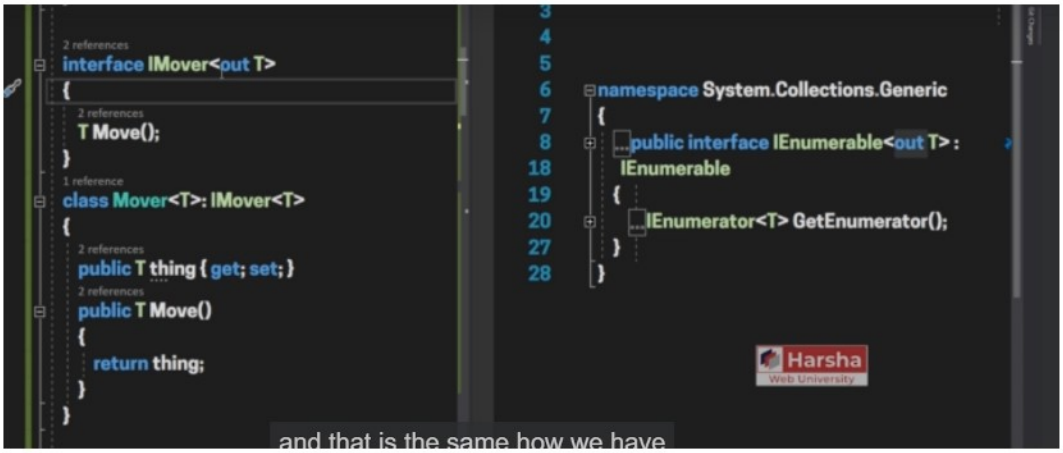
The child class name that is living thing is expected but in practical in the right hand side we are trying to supply tSobhe child class name so parent is expected but you are going to supply the child that is parrot that is where the covariance is
implemented but this should be happened only with generic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



Sob y bujhlam. but etar real world benefit ta ki?

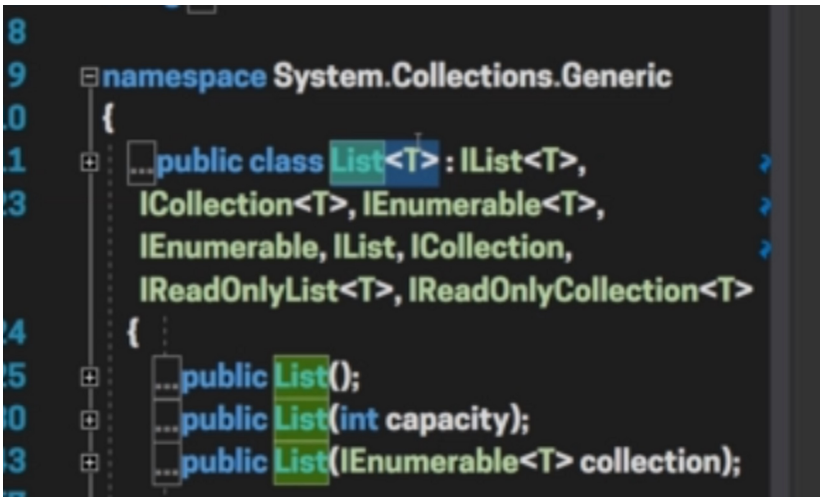
```
class Sample
{
    //public void PrintValues(List<string> values)
    //{
    //    foreach (var item in values)
    //    {
    //        Console.WriteLine(item + ",");
    //    }
    //    Console.WriteLine();
    //}

    // 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();
    }
}
```

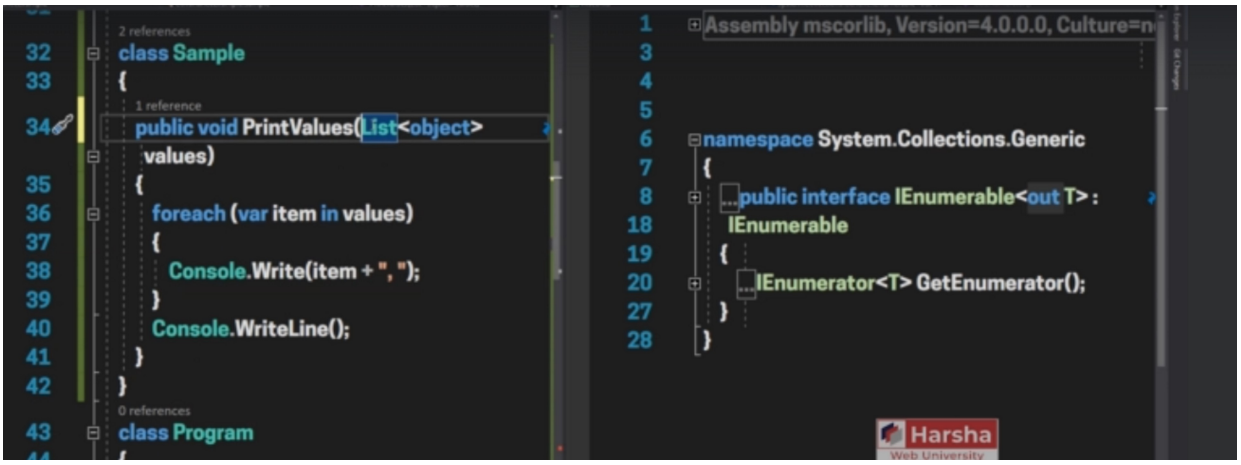
```
Sample s = new Sample();
s.PrintValues(new List<string>
{
    "Hello",
    "World",
    "Bangladesh"
});
```



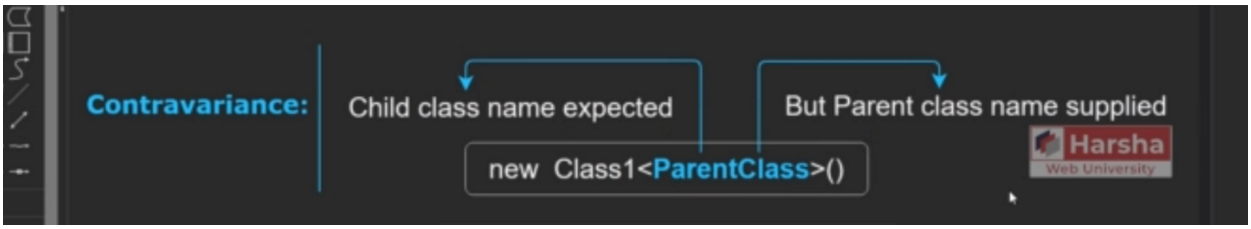
But List is not Covariance supported

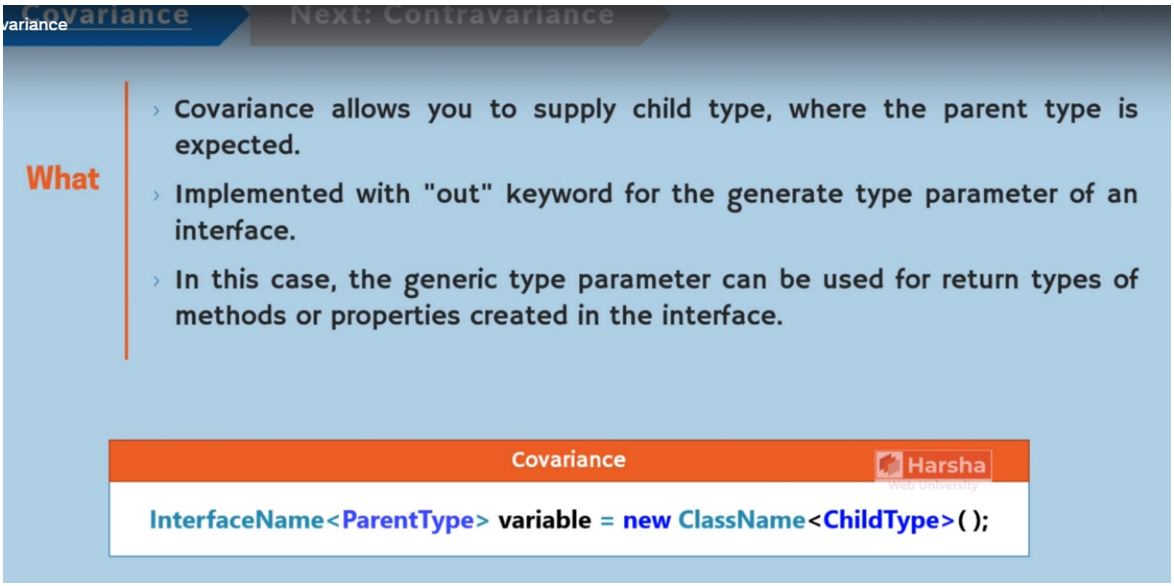
But List is not Covariance supported.





Contravariance





Contravariance

What

- Contravariance allows you to supply parent type, where the child type is expected.
- Implemented with "in" keyword for the generate 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>();
```