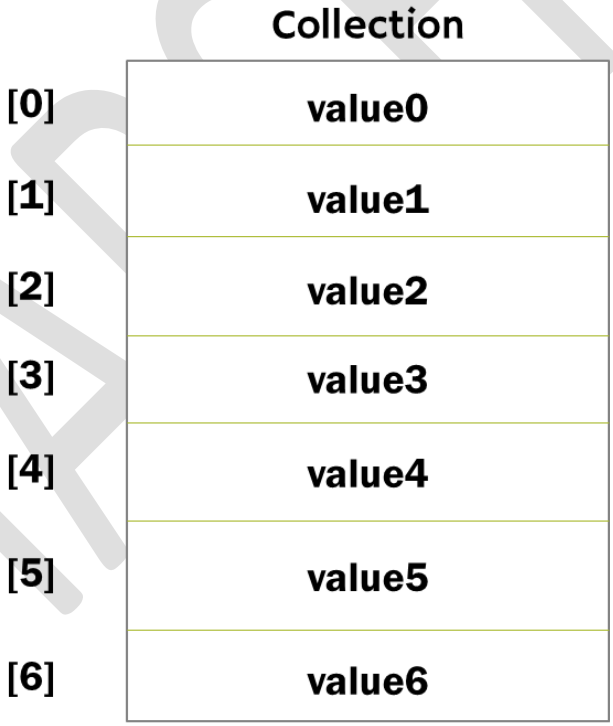
36

Collections

Introducing Collections

- 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.



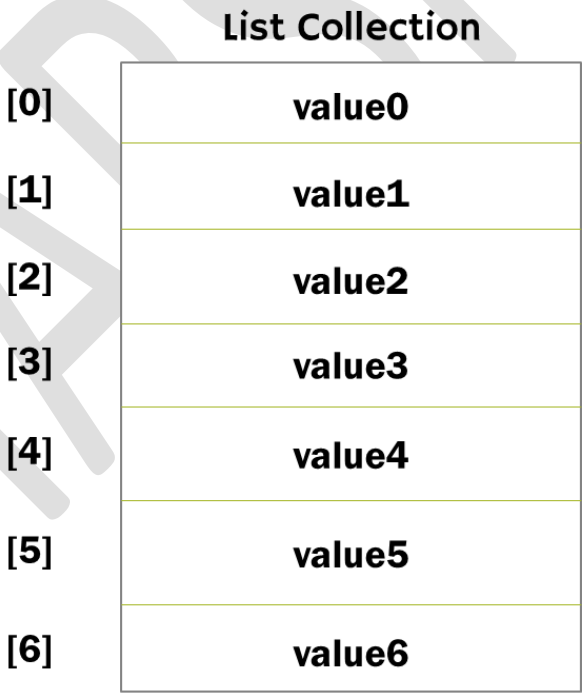
'List' collection

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

- 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.

The 'List' collection

- List collection contains a group of elements of same type.
- Full Path: System.Collections.Generic.List
- The 'List' class is a generic class; so you need to specify data type of value while creating object.



'List' collection

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

- 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 and 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)

List.Add( ) method

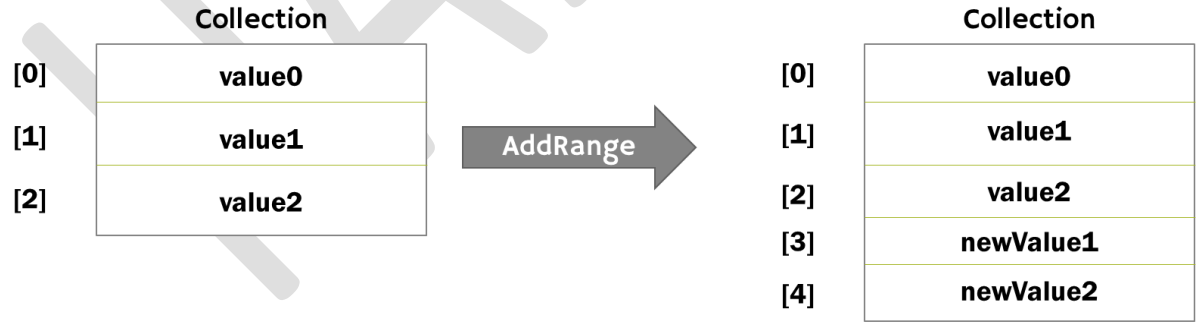
- This method adds a new element to the collection.

List - Add() method

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

List.AddRange( ) method

- This method adds a new set of elements to the collection.



List - AddRange() method

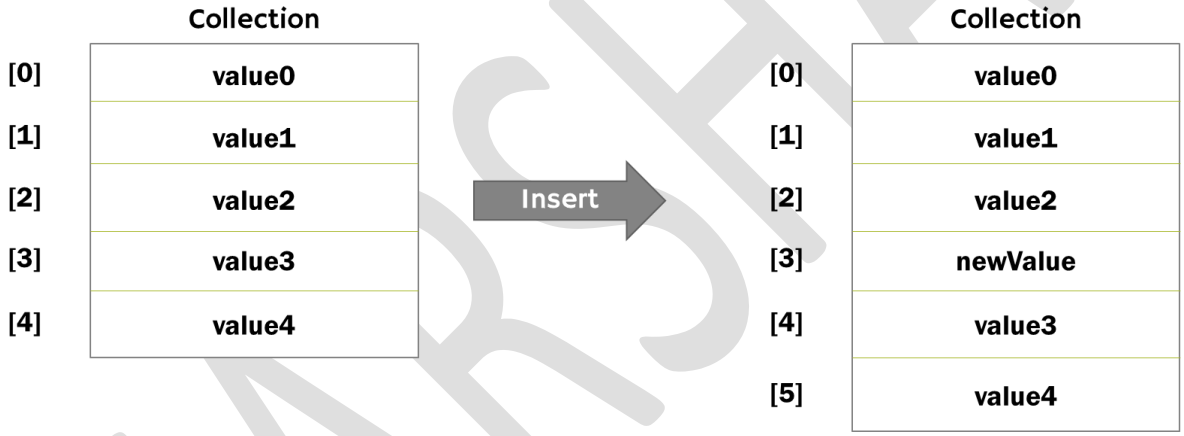
```
void List.AddRange(IEnumerable<T> newValue)
```

List - AddRange() - Example

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

List.Insert( ) method

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



List - Insert() method

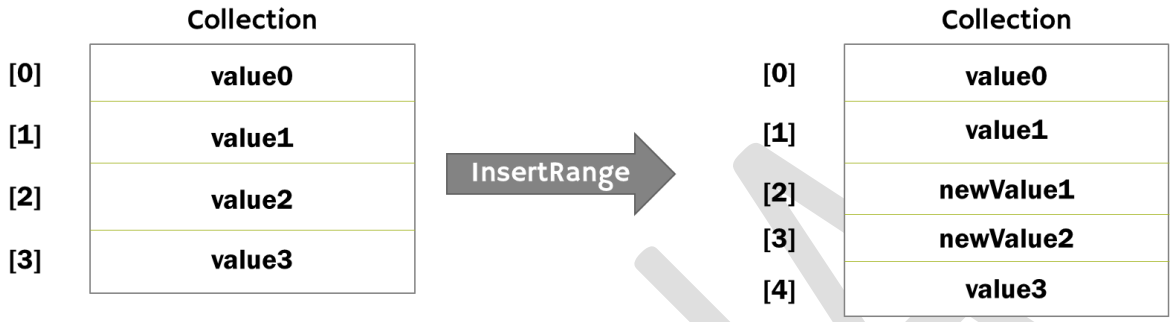
```
void  List.Insert(int index, T  newValue)
```

List - Insert() - Example

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

List.InsertRange( ) method

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



List - InsertRange( ) method

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

List - InsertRange( ) - Example

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

List.Remove( ) method

› This method removes the specified element from the collection.

List - Remove() method

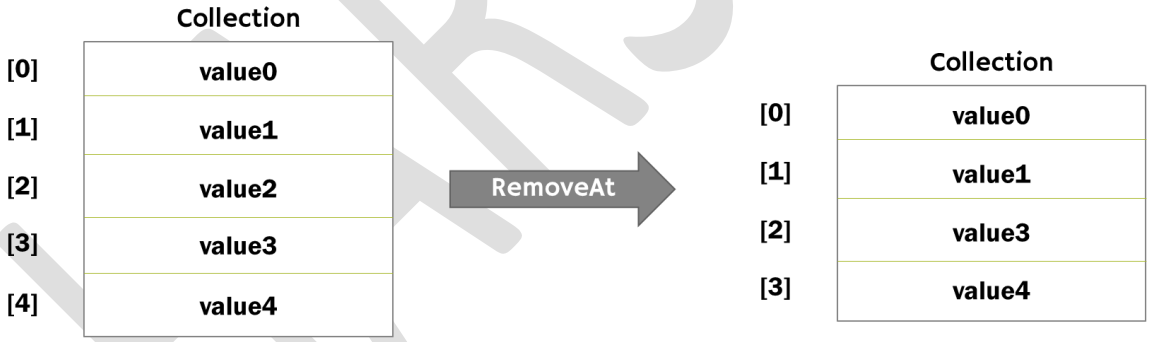
```
void List.Remove(T newValue)
```

List - Remove() - Example

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

List.RemoveAt( ) method

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



List - RemoveAt() method

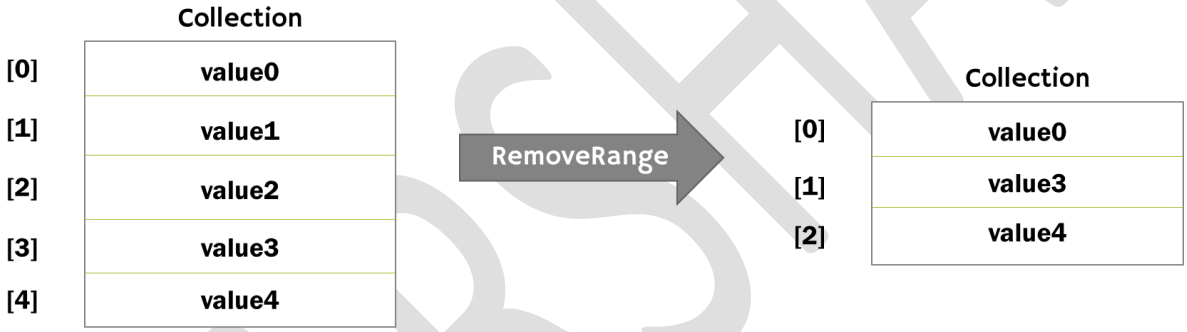
```
void List.RemoveAt(int index)
```

List - RemoveAt() - Example

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

List.RemoveRange( ) method

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



List - RemoveRange() method

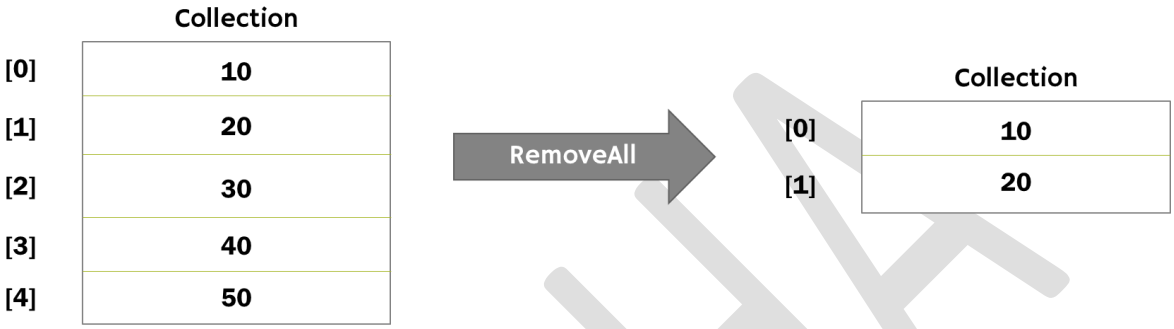
```
void List.RemoveRange(int index, int count)
```

List - RemoveRange() - Example

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

List.RemoveAll( ) method

- 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

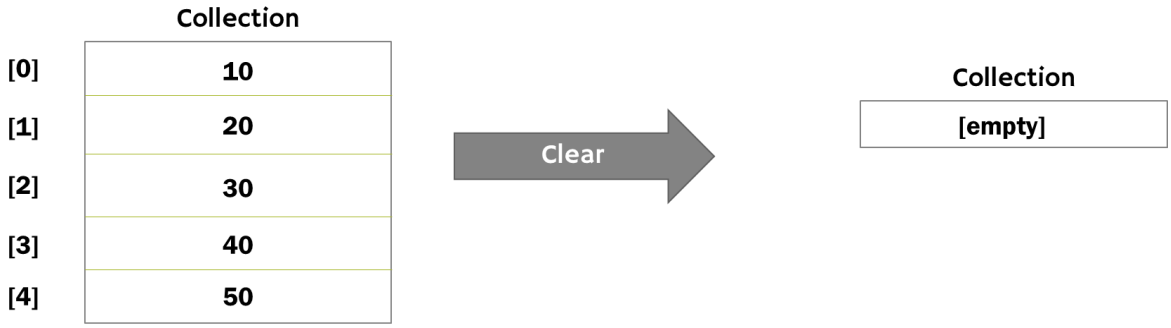
```
void List.RemoveAll(value => condition)
```

List - RemoveAll( ) - Example

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

List.Clear( ) method

- This methods removes all elements in the collection.



List - Clear( ) method

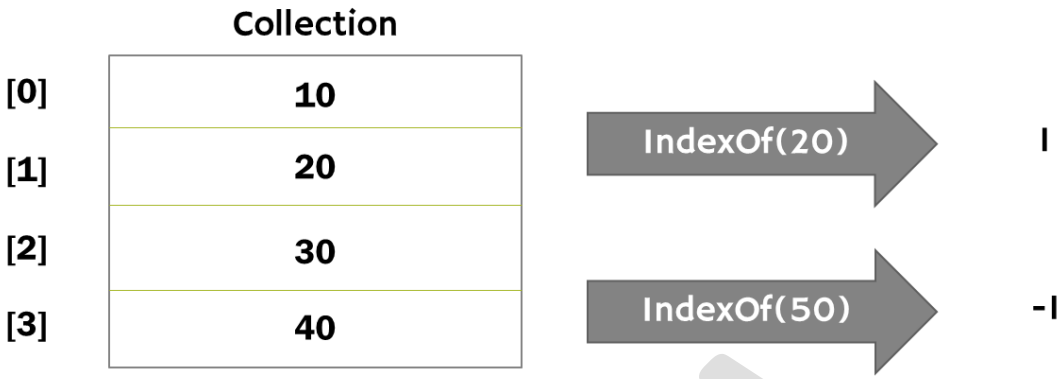
```
void List.Clear( )
```

List - Clear( ) - Example

```
List.Clear( )
```

List.IndexOf( ) method

- 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

```
int  List.IndexOf(T value, int startIndex)
```

List - IndexOf() - Example

```
List.IndexOf(20)
```

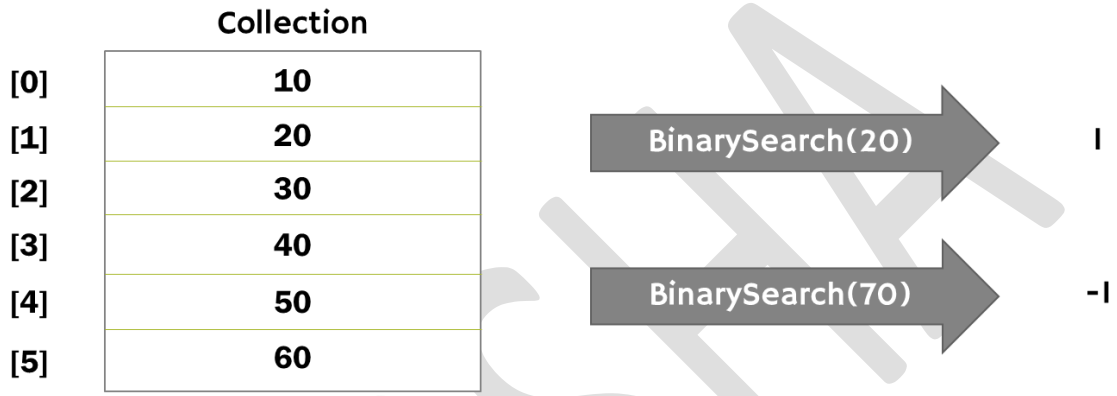
- The "IndexOf" method performs linear search. That means it searches all the elements of the collection, until the search value is found. When the search value is found in the collection, it stops searching and returns its index.
- The linear search has good performance, if the collection is small. But if the collection is larger, Binary search is recommended to improve the performance.

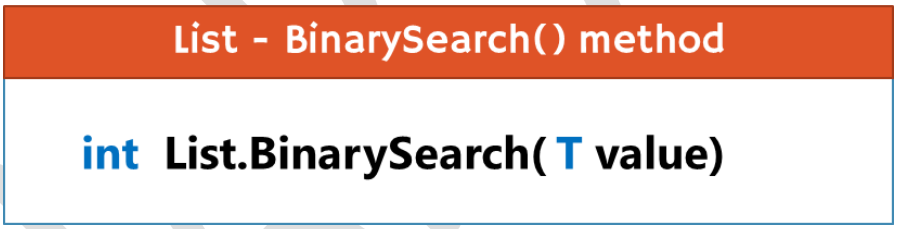
Parameters of IndexOf( ) method

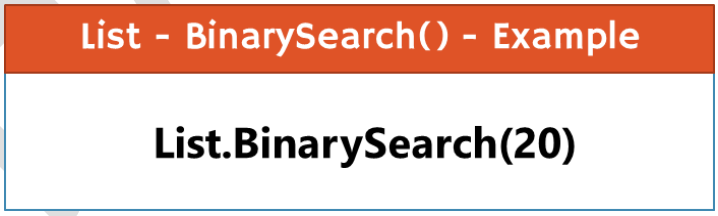
- **value:** This parameter represents the actual value that is to be searched.
- **startIndex:** This parameter represents the start index, from where the search should be started.

List.BinarySearch( ) method

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







- 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.

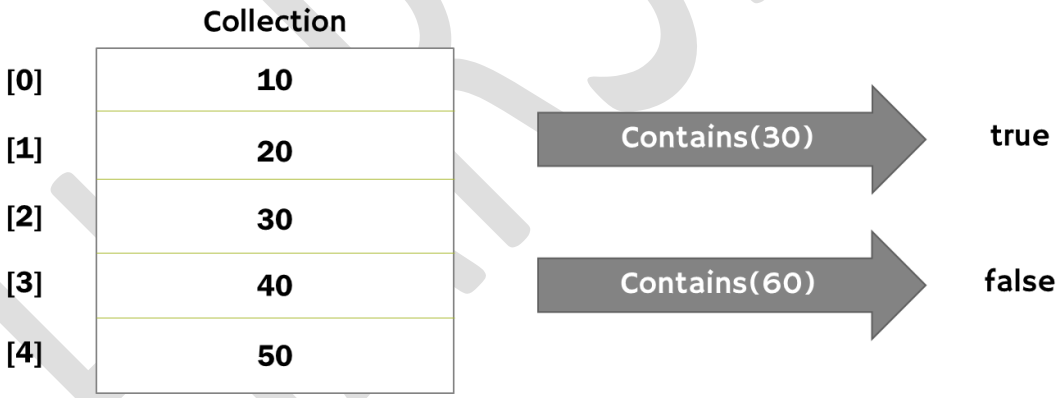
- If that item is greater than the search value, it searches only in the first half of the collection.
- If that item is less than the search value, it searches only in the second half of the array.
- Thus it searches only half of the array. So in this way, it improves performance

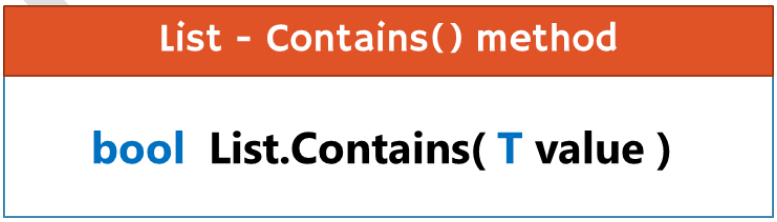
Parameters of BinarySearch( ) method

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

List.Contains( ) method

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



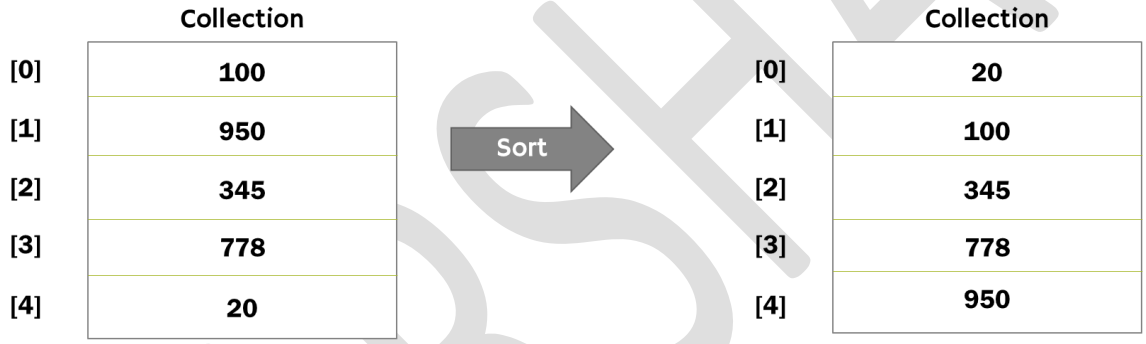


Contains() - Example

```
List.Contains(30)
```

List.Sort( ) method

› This method sorts the collection in ascending order.



List - Sort() method

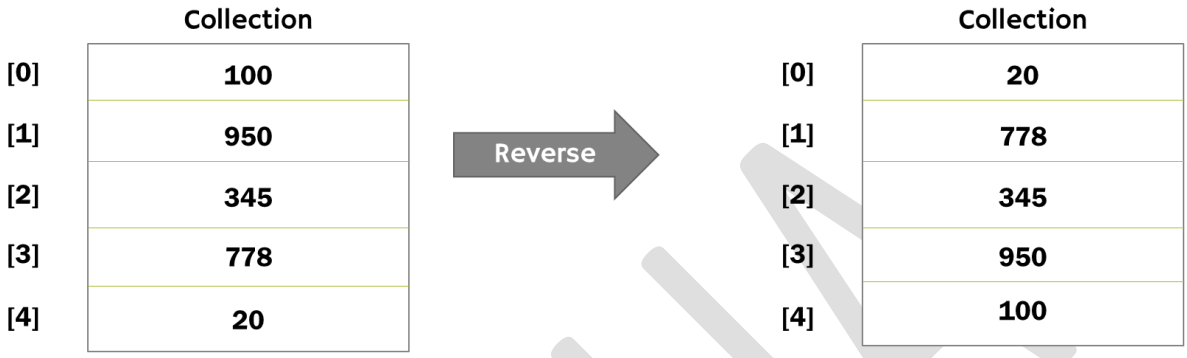
```
void List.Sort( )
```

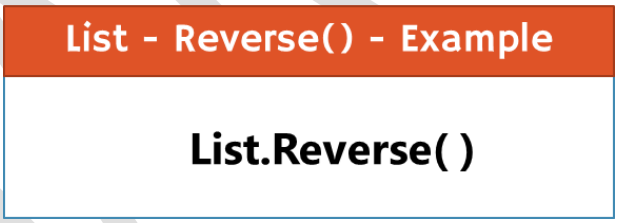
List - Sort() - Example

```
List.Sort( )
```

List.Reverse( ) method

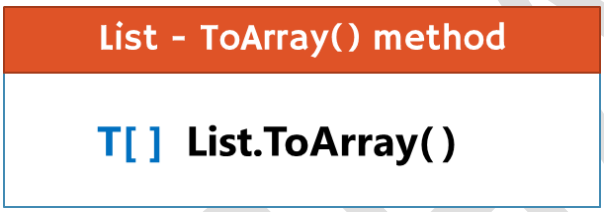
› This method reverses the collection.

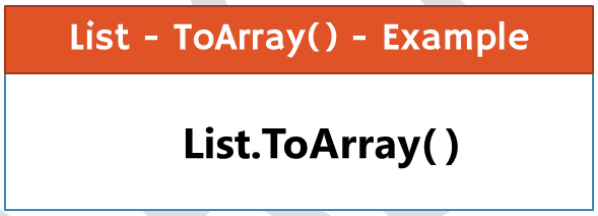




List.ToArray( ) method

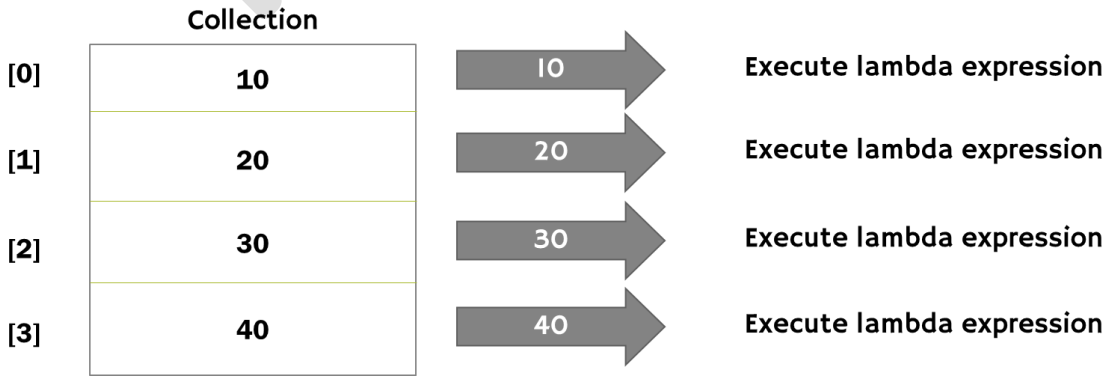
› This method converts the collection into an array with same elements.





List.ForEach( ) method

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



List - ForEach() method

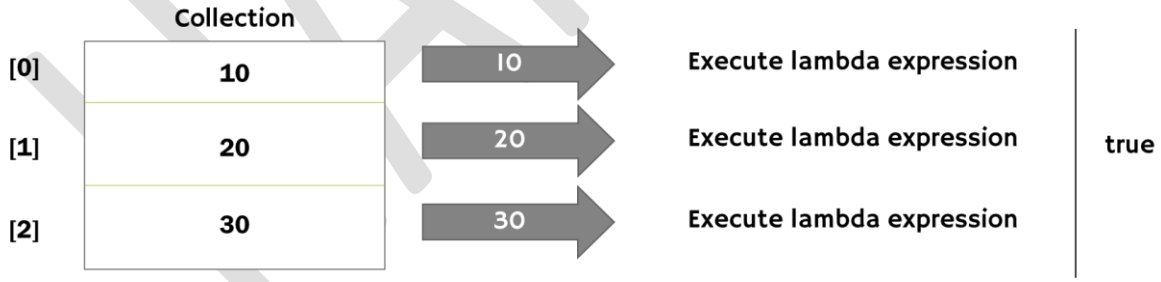
```
void List.ForEach( Action<T> )
```

List - ForEach() - Example

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

List.Exists( ) method

- 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.



List - Exists() method

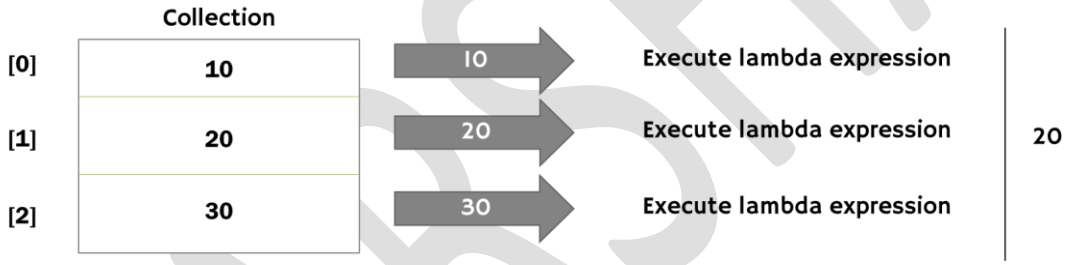
```
bool List.Exists( Predicate<T> )
```

List - Exists() - Example

```
List.Exists( n => n > 15 )
```

List.Find( ) method

- 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.



List - Find() method

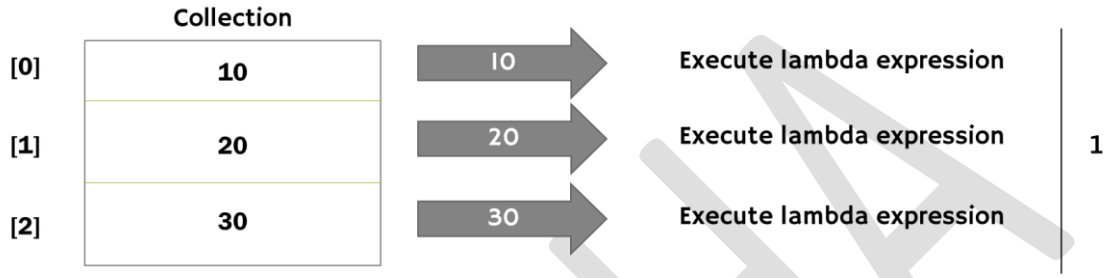
```
T List.Find( Predicate<T> )
```

List - Find() - Example

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

List.FindIndex( ) method

- 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.



List - FindIndex() method

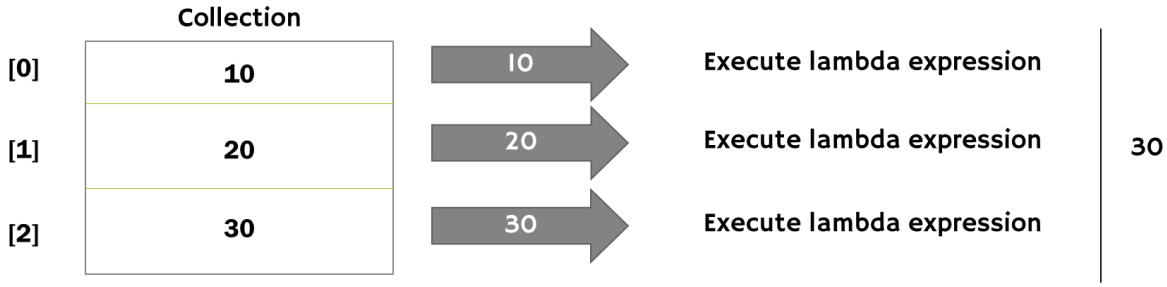
```
int List.FindIndex( Predicate<T> )
```

List - FindIndex() - Example

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

List.FindLast( ) method

- 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.



List - FindLast() method

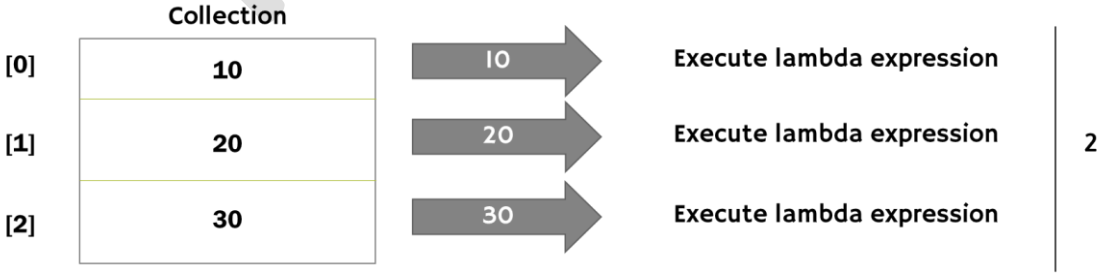
```
T List.FindLast( Predicate<T> )
```

List - FindLast() - Example

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

List.FindLastIndex( ) method

- 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.



List - FindLastIndex() method

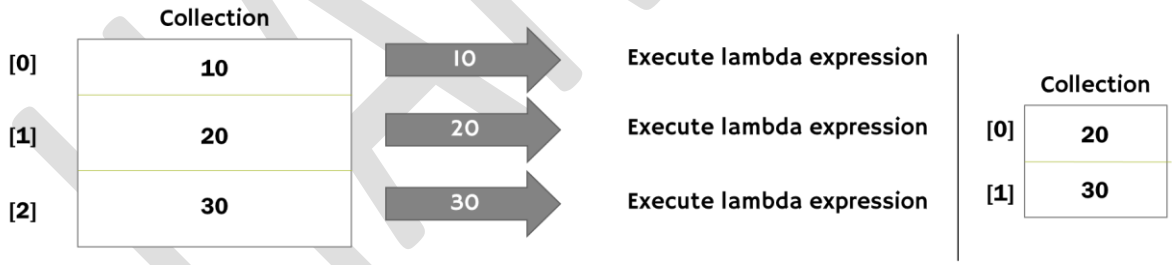
```
int List.FindLastIndex( Predicate<T> )
```

List - FindLastIndex() - Example

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

List.FindAll( ) method

- 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.



List - FindAll() method

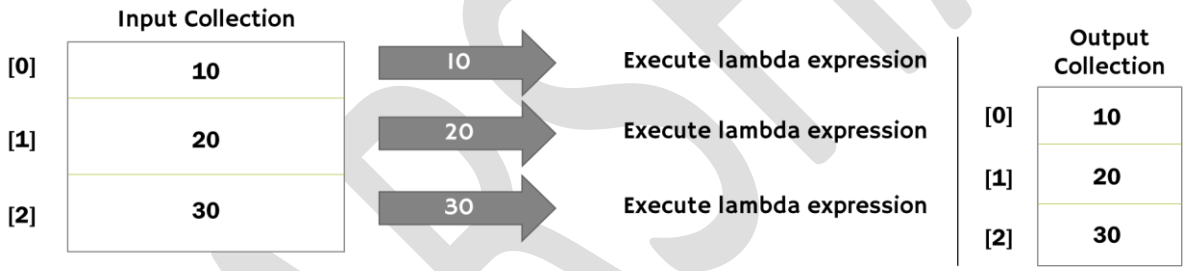
```
List<T> List.FindAll( Predicate<T> )
```

List - FindAll() - Example

```
List.FindAll( n => n > 15 )
```

List.ConvertAll( ) method

- 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.



List - ConvertAll() method

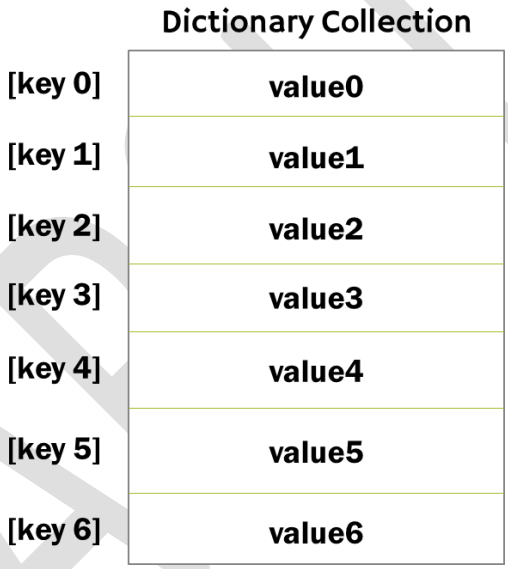
```
List<TOutput>  List.ConvertAll( Converter<TInput, TOutput> )
```

List - ConvertAll() - Example

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

The Dictionary Collection

- 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

```
Dictionary<TKey, TValue>  referenceVariable = new  Dictionary<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 not sorted by default. The elements are stored in the same order, how they are initialized.

Properties and Methods of 'Dictionary' class

Properties

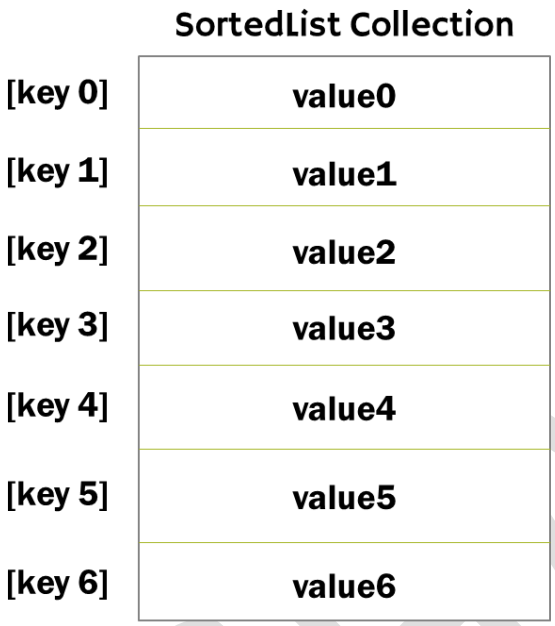
- **Count** : Returns count of elements.
- **[TKey]** : Returns value based on specified key.
- **Keys** : Returns a collection of key (without values).
- **Values** : Returns a collection of values (without keys).

Methods

- **void Add(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.

The SortedList Collection

- 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

```
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 and Methods of 'SortedList' class

Properties

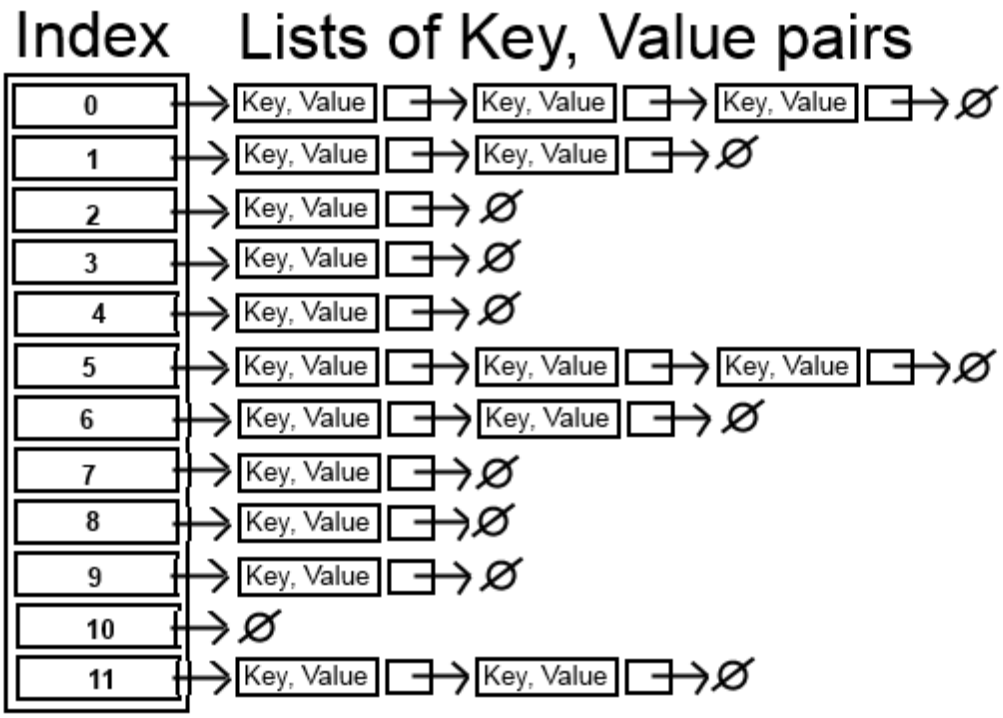
- **Count** : Returns count of elements.
- **[TKey]** : Returns value based on specified key.
- **Keys** : Returns a collection of key (without values).
- **Values** : Returns a collection of values (without keys).

Methods

- **void Add(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.

The Hashtable Collection

- Hashtable collection contains a group of elements of key/value pairs stored at respective indexes.
- Full Path: System.Collections.Hashtable
- The "Hashtable" class is not a generic class.
- You can set / get the value based on the key.
- The key can't be null or duplicate.



'Hashtable' collection

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

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

Properties and Methods of Hashtable class

Properties

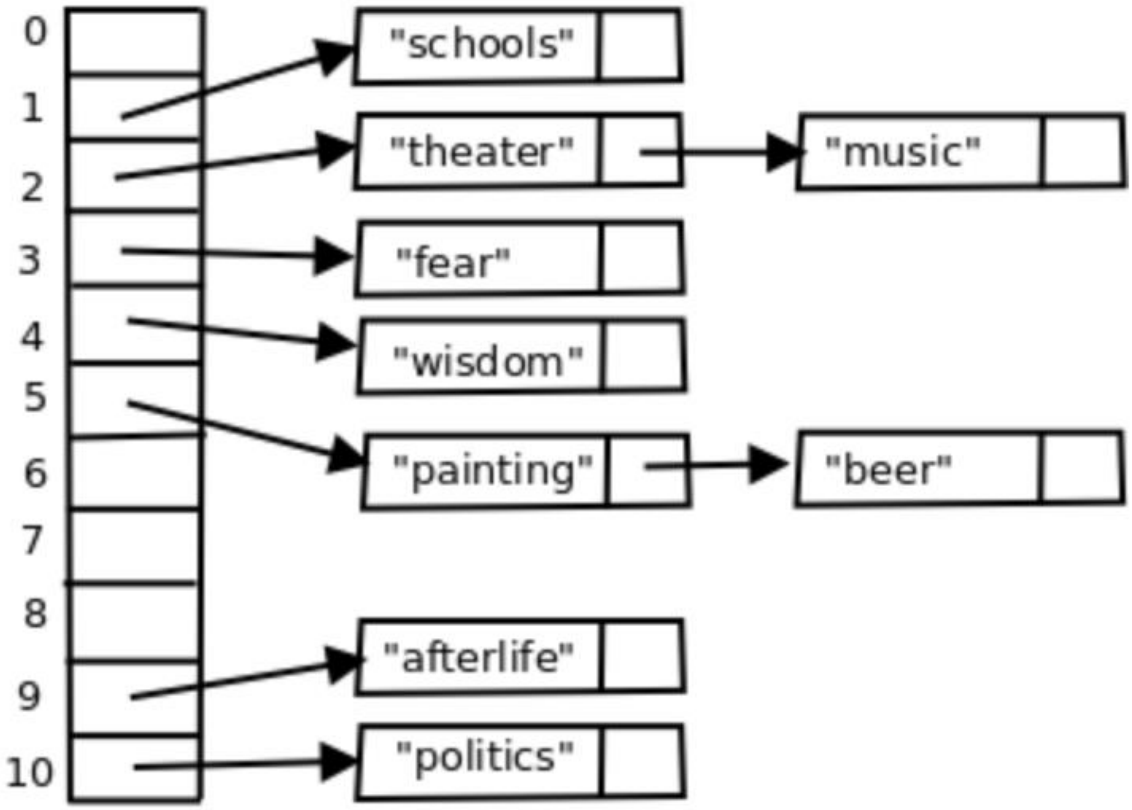
- **Count** : Returns count of elements.
- **[TKey]** : Returns value based on specified key.
- **Keys** : Returns a collection of key (without values).
- **Values** : Returns a collection of values (without keys).

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.

The HashSet Collection

- HashSet collection contains a group of elements of unique values stored at respective indexes.
- Full Path: System.Collections.Generic.HashSet
- The "HashSet" class is a generic class.
- You can't set / get the element based on the key / index.
- It searches elements based on the index generated based on the search value.



'HashSet' collection

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

- HashSet allows only one null value; Hashtable allows only one null key; but allows multiple null values.
- You can't access elements based on key / index. You can use Contains method to search for an element.
- You can't sort elements in HashSet.
- Elements must be unique; duplicate elements are not allowed.
- 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.

Properties and Methods of HashSet class

Properties

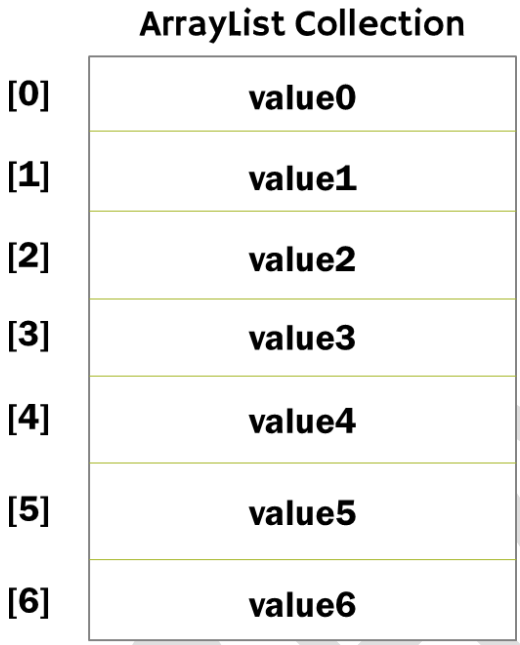
- **Count** : Returns count of elements.

Methods

- **void Add(T value)** : Adds an element (key/value pair).
- **void Remove(T value)** : Removes an element based on specified key.
- **void RemoveWhere(Predicate)** : Remove elements that matches with condition.
- **bool Contains (T value)** : Determines whether the specified value exists.
- **void Clear()** : Removes all elements.
- **void UnionWith(IEnumerable<T>)** : Unions the hashset and specified collection.
- **void IntersectWith(IEnumerable<T>)** : Intersects the hashset and specified collection.

The ArrayList Collection

- 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

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

- 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.

Properties and Methods of ArrayList class

Properties

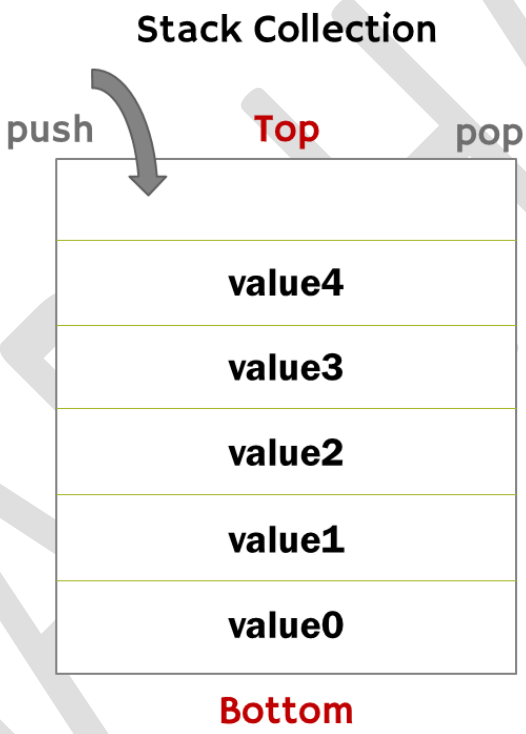
- Count
- Capacity

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()

The Stack Collection

- Stack collection contains a group of elements based on LIFO (Last-In-First-Out) based collection.
- Full Path: System.Collections.Generic.Stack
- 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.



'Stack' collection

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

- It is based on LIFO (Last-In-First-Out).

- 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.
- It stores the elements in bottom-to-up approach.
    - Firstly added item at bottom.
    - Lastly added item at top.

Properties and Methods of Stack class

Properties

- **Count** : Returns count of elements.

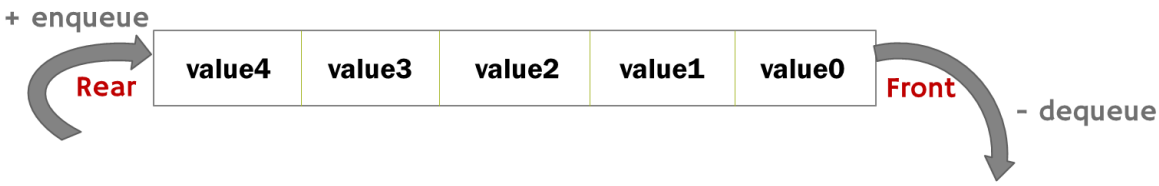
Methods

- **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.

The Queue Collection

- Queue collection contains a group of elements based on FIFO (First-In-First-Out) based collection.
- Full Path: System.Collections.Generic.Queue
- The "Queue" class is a generic class.

- You can't access elements based on index.



'Queue' collection

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

- It is based on FIFO (First-In-First-Out).
- 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.
- It stores the elements in front-to-rear approach.
  - Firstly added item at front.
  - Lastly added item at rear.

Properties and Methods of Queue class

Properties

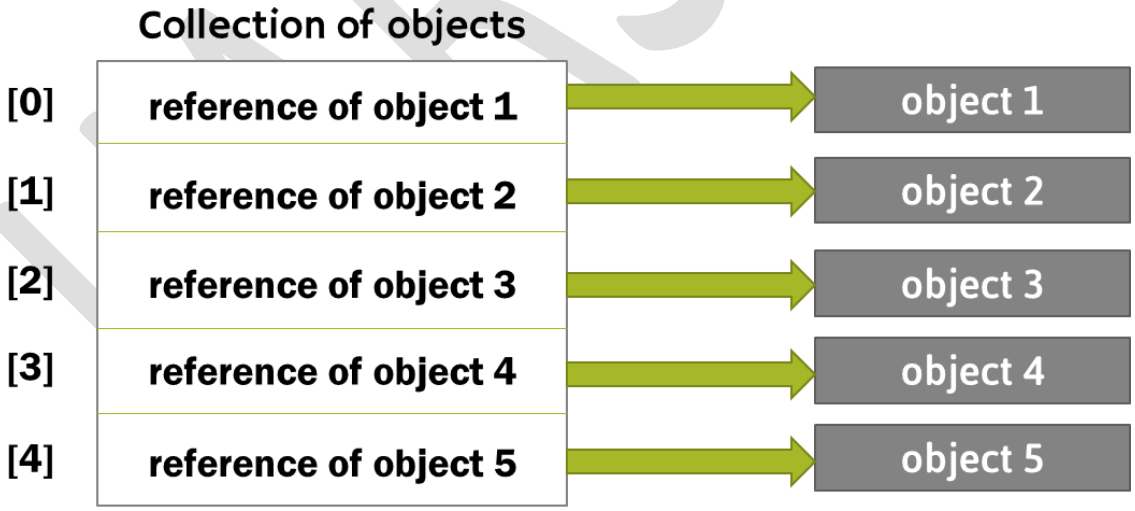
- **Count** : Returns count of elements.

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

- 'Collection of objects' is an object that stores a set of references to other objects.
- Used to store details of groups of people or things.



Collection of Objects

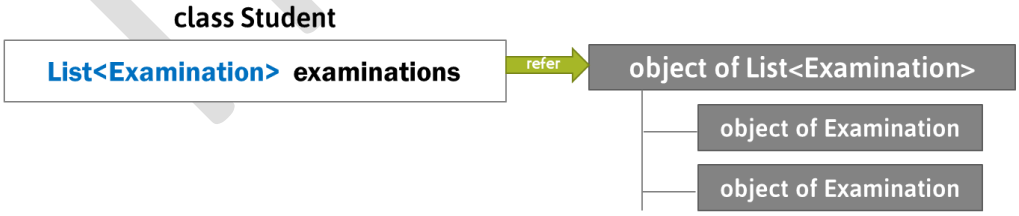
```
List<ClassName>  referenceVariable = new  List<ClassName>( );
referenceVariable.Add(object1);
referenceVariable.Add(object2);
referenceVariable.Add(object3);
...
```

Object Relations

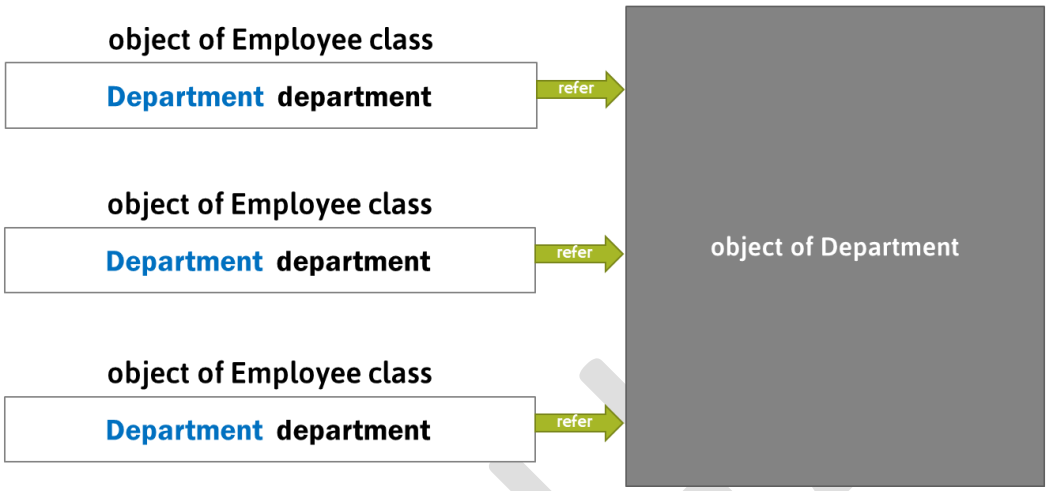
- An object can contain a field that stores references to one or more objects.

One-to-One Relation

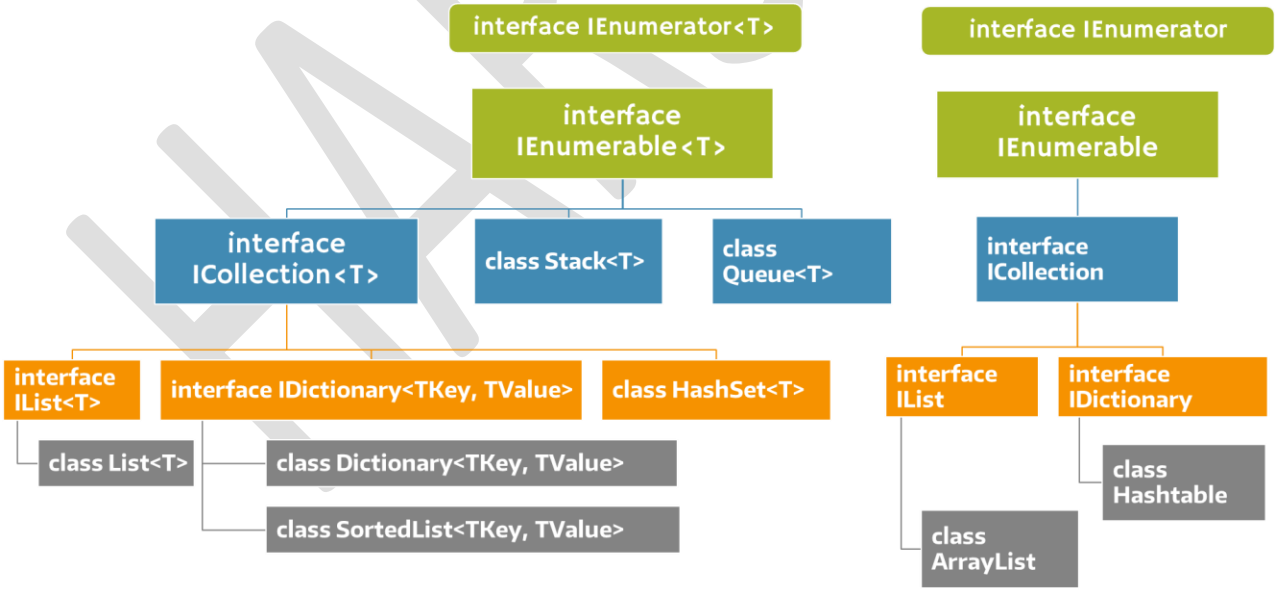
One-to-Many Relation



Many-to-One Relation



Collection Hierarchy



IEnumerable

- 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> GetEnumerator();
}
```

System.Collections.IEnumerable

```
public interface IEnumerable : IEnumerable
{
    IEnumerator GetEnumerator();
}
```

Working with IEnumerable

System.Collections.Generic.IEnumerable<T>

```
IEnumerable<T>  referenceVariable = new  List<T>( );
```

System.Collections.IEnumerable

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

IEnumerator

- 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();
}
```

Working with IEnumerator

System.Collections.Generic.IEnumerator<T>

```
IEnumerator<T>  referenceVariable = new  List<T>( ).GetEnumerator( );
```

System.Collections.IEnumerator

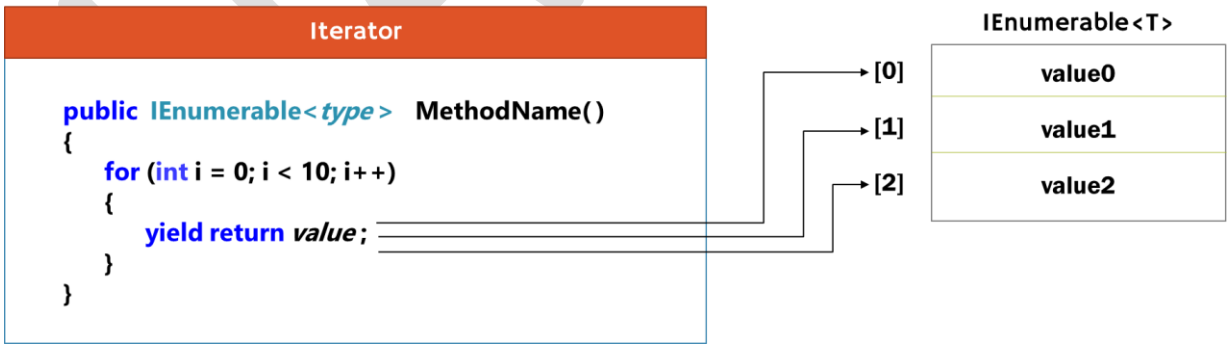
```
IEnumerator  referenceVariable = new  ArrayList( ).GetEnumerator( );
```

Iterators

- Iterator is a method which yield returns elements one-by-one.
- Iterator is used to iterator over a set of elements.
- The *yield return* statement returns an element; but be ready to return the next element.

Iterator

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



- 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.

Custom Collections

- 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];
        }
    }
}
```

- 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; but you need to implement all methods of IList interface in this case.
- You can also create custom collection by inheriting from List; it provides all methods of List to your collection class.

IEquatable

- 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

- 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.

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.

interface System.IComparable

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

Implementation of IComparable interface

```
public class ClassName : IComparable
{
    public int CompareTo(object obj)
    {
        if (this.field1 == other.field1)
            return 0; //equal
        else if (this.field1 < other.field1)
            return -1; //"this" object comes first.
        else
            return 1; //parameter object comes first.
    }
}
```

IComprer

- 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.

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 interface IComparer<T>
{
    int Compare(T x, T y);
}
```

Implementation of IComparer interface

```
public class ClassName : IComparer<T>
{
    public int Compare(ClassName x, ClassName y)
    {
        if (x.field1 == y.field1)
            return 0;  //equal
        else if (x.field1 < y.field1)
            return -1;  //x object comes next.
        else
            return 1;  //y object comes next.
    }
}
```

Covariance

- You can supply an object of child type where the parent type is expected.
- Applicable for delegates, generics.

Covariance

```
IEnumerable<object> objects = new List<string>();
```

Contravariance

- You can supply an object of parent type where the child type is expected.
- Applicable for delegates only.

Contravariance

```
Action<string> act = Method1;
//assume static void Method1(object o)
```