

C# - Generic Collection

You have learned about the [collection](#) in the previous section, e.g. [ArrayList](#), [BitArray](#), [SortedList](#), [Queue](#), [Stack](#) and [Hashtable](#). These types of collections can store any type of items. For example, [ArrayList](#) can store items of different data types:

Example: C# ArrayList Collection

```
ArrayList arList = new ArrayList();  
  
arList.Add(1);  
arList.Add("Two");  
arList.Add(true);  
arList.Add(100.45);  
arList.Add(DateTime.Now);
```

The limitation of these collections is that while retrieving items, you need to cast into the appropriate data type, otherwise the program will throw a runtime exception. It also affects on performance, because of boxing and unboxing.

To overcome this problem, C# includes generic collection classes in the ***System.Collections.Generic*** namespace.

The following are widely used generic collections:

Generic Collections	Description
<code>List<T></code>	Generic <code>List<T></code> contains elements of specified type. It grows automatically as you add elements in it.
<code>Dictionary<TKey,TValue></code>	<code>Dictionary<TKey,TValue></code> contains key-value pairs.
<code>SortedList<TKey,TValue></code>	<code>SortedList</code> stores key and value pairs. It automatically adds the elements in ascending order of key by default.
<code>HashSet<T></code>	<code>HashSet<T></code> contains non-duplicate elements. It eliminates duplicate elements.
<code>Queue<T></code>	<code>Queue<T></code> stores the values in FIFO style (First In First Out). It keeps the order in which the values were added. It provides an <code>Enqueue()</code> method to add values and a <code>Dequeue()</code> method to retrieve values from the collection.
<code>Stack<T></code>	<code>Stack<T></code> stores the values as LIFO (Last In First Out). It provides a <code>Push()</code> method to add a value and <code>Pop()</code> & <code>Peek()</code> methods to retrieve values.

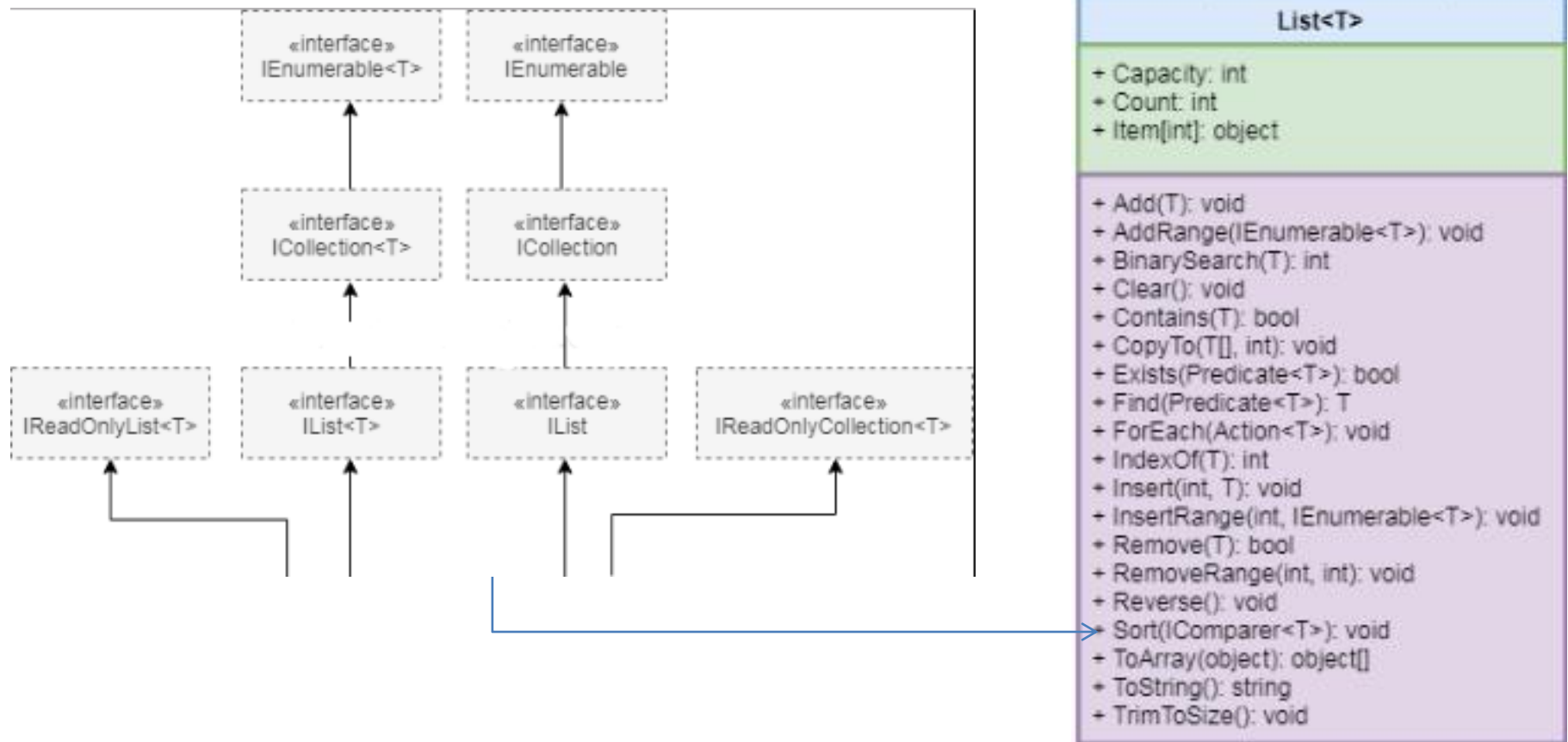
A generic collection gets all the benefit of generics. It doesn't need to do boxing and unboxing while storing or retrieving items and so performance is improved.

Learn generic collection `List<T>` in the next section.

C# - List<T>

You have already learned about `ArrayList` in the previous section. An `ArrayList` resizes automatically as it grows. The `List<T>` collection is the same as an `ArrayList` except that `List<T>` is a generic collection whereas `ArrayList` is a non-generic collection.

The following diagram illustrates the `List<T>` hierarchy.



As shown in the above diagram, the `List<T>` class implements eight different interfaces that provide different functionalities. Hence, the `List<T>` object can be assigned to any of its interface type variables. However, it is recommended to create an object of `List<T>` and assign it to `IList<T>` or `List<T>` type variable, as shown below.

Example: List<T>

```
List<int> intList = new List<int>();  
  
//Or  
  
IList<int> intList = new List<int>();
```

In the above example, the first statement uses `List<T>` type variable, whereas the second statement uses `IList<T>` type variable. The `List<T>` is a concrete implementation of `IList<T>` interface. In the object-oriented programming, it is advisable to program to interface rather than concrete class. So use `IList<T>` type variable to create an object of `List<T>`.

Example: List<T>

```
List<int> intList = new List<int>();  
  
//Or  
  
IList<int> intList = new List<int>();
```

In the above example, the first statement uses `List<T>` type variable, whereas the second statement uses `IList<T>` type variable. The `List<T>` is a concrete implementation of `IList<T>` interface. In the object-oriented programming, it is advisable to program to interface rather than concrete class. So use `IList<T>` type variable to create an object of `List<T>`.

However, `List<T>` includes more helper methods than `IList<T>` interface. The following table lists the important properties and methods of `List<T>` class:

In the above example, the first statement uses `List<T>` type variable, whereas the second statement uses `IList<T>` type variable. The `List<T>` is a concrete implementation of `IList<T>` interface. In the object-oriented programming, it is advisable to program to interface rather than concrete class. So use `IList<T>` type variable to create an object of `List<T>`.

However, `List<T>` includes more helper methods than `IList<T>` interface. The following table lists the important properties and methods of `List<T>` class:

Property	Usage
Items	Gets or sets the element at the specified index
Count	Returns the total number of elements exists in the <code>List<T></code>

Method	Usage
Add	Adds an element at the end of a List<T>.
AddRange	Adds elements of the specified collection at the end of a List<T>.
BinarySearch	Search the element and returns an index of the element.
Clear	Removes all the elements from a List<T>.
Contains	Checks whether the specified element exists or not in a List<T>.
Find	Finds the first element based on the specified predicate function.
Foreach	Iterates through a List<T>.
Insert	Inserts an element at the specified index in a List<T>.
InsertRange	Inserts elements of another collection at the specified index.
Remove	Removes the first occurrence of the specified element.
RemoveAt	Removes the element at the specified index.
RemoveRange	Removes all the elements that match with the supplied predicate function.
Sort	Sorts all the elements.
TrimExcess	Sets the capacity to the actual number of elements.
TrueForAll	Determines whether every element in the List<T> matches the conditions defined by the specified predicate.

Add Elements into List

Use the `IList.Add()` method to add an element into a List collection. The following example adds int value into a List<T> of *int* type.

Add() signature: `void Add(T item)`

Example: Adding elements into List

```
IList<int> intList = new List<int>();
intList.Add(10);
intList.Add(20);
intList.Add(30);
intList.Add(40);

IList<string> strList = new List<string>();
strList.Add("one");
strList.Add("two");
strList.Add("three");
strList.Add("four");
strList.Add("four");
strList.Add(null);
strList.Add(null);

IList<Student> studentList = new List<Student>();
studentList.Add(new Student());
studentList.Add(new Student());
studentList.Add(new Student());
```


You can also add elements at the time of initialization using object initializer syntax as below:

Example: Add elements using object initializer syntax

```
ICollection<int> intList = new List<int>(){ 10, 20, 30, 40 };
```

//Or

```
ICollection<Student> studentList = new List<Student>() {  
    new Student(){ StudentID=1, StudentName="Bill"},  
    new Student(){ StudentID=2, StudentName="Steve"},  
    new Student(){ StudentID=3, StudentName="Ram"},  
    new Student(){ StudentID=1, StudentName="Moin"}  
};
```

AddRange()

Use `List.AddRange()` method adds all the elements from another collection.

AddRange() signature: *void AddRange(IEnumerable<T> collection)*

Example: AddRange

```
ICollection<int> intList1 = new List<int>();  
intList1.Add(10);  
intList1.Add(20);  
intList1.Add(30);  
intList1.Add(40);  
  
List<int> intList2 = new List<int>();  
  
intList2.AddRange(intList1);
```

Accessing List

Use a foreach or for loop to iterate a List<T> collection.

Example: Accessing List

```
List<int> intList = new List<int>() { 10, 20, 30 };  
  
intList.ForEach(el => Console.WriteLine(el));
```

Try it

If you have initialized the List<T> with an IList<T> interface then use separate foreach statement with implicitly typed variable:

Example: Accessing List

```
IList<int> intList = new List<int>() { 10, 20, 30, 40 };  
  
foreach (var el in intList)  
    Console.WriteLine(el);
```

Access individual items by using an indexer (i.e., passing an index in square brackets):

Example: Accessing List

```
IList<int> intList = new List<int>() { 10, 20, 30, 40 };  
  
int elem = intList[1]; // returns 20
```

Use the *Count* property to get the total number of elements in the List.

Example: Access List elements

```
IList<int> intList = new List<int>() { 10, 20, 30, 40 };  
  
Console.WriteLine("Total elements: {0}", intList.Count);
```

Output:

```
Total elements: 4
```

Use for loop to access list as shown below:

Example: Accessing List using for loop example:

```
IList<int> intList = new List<int>() { 10, 20, 30, 40 };  
  
for (int i = 0; i < intList.Count; i++)  
    Console.WriteLine(intList[i]);
```

List<T> implements IList<T>, so List<T> implicitly type cast to IList<T>.

Example: Access List

```
static void Print(IList<string> list)
{
    Console.WriteLine("Count: {0}", list.Count);
    foreach (string value in list)
    {
        Console.WriteLine(value);
    }
}

static void Main(string[] args)
{
    string[] strArray = new string[2];
    strArray[0] = "Hello";
    strArray[1] = "World";
    Print(strArray);

    List<string> strList = new List<string>();
    strList.Add("Hello");
    strList.Add("World");
    Print(strList);
}
```

Insert Elements into List

Use the `IList.Insert()` method inserts an element into a `List<T>` collection at the specified index.

Insert() signature: `void Insert(int index, T item);`

Example: Insert elements into List

```
IList<int> intList = new List<int>(){ 10, 20, 30, 40 };  
  
intList.Insert(1, 11); // inserts 11 at 1st index: after 10.  
  
foreach (var el in intList)  
    Console.Write(el);
```

Remove Elements from List

The `Remove()` and `RemoveAt()` methods to remove items from a `List<T>` collection.

`Remove()` signature: `bool Remove(T item)`

`RemoveAt()` signature: `void RemoveAt(int index)`

Example: Remove elements from List

```
IList<int> intList = new List<int>(){ 10, 20, 30, 40 };

intList.Remove(10); // removes the 10 from a list

intList.RemoveAt(2); //removes the 3rd element (index starts from 0)

foreach (var el in intList)
    Console.Write(el);
```


TrueForAll()

The `TrueForAll()` is a method of the `List<T>` class that returns true if the specified condition turns out to be true, otherwise false. Here, the condition can be specified as the [predicate](#) type delegate or the [lambda expression](#).

TrueForAll() signature: `bool TrueForAll(Predicate<T> match)`

Example: TrueForAll()

```
List<int> intList = new List<int>(){ 10, 20, 30, 40 };  
  
bool res = intList.TrueForAll(e1 => e1%2 == 0);// returns true
```

The following example uses `isPositiveInt()` as a `Predicate<int>` type delegate as a parameter to `TrueForAll`.

Example: TrueForAll()

```
static bool isPositiveInt(int i)
{
    return i > 0;
}

static void Main(string[] args)
{
    List<int> intList = new List<int>(){10, 20, 30, 40};

    bool res = intList.TrueForAll(isPositiveInt);
}
```



Points to Remember :

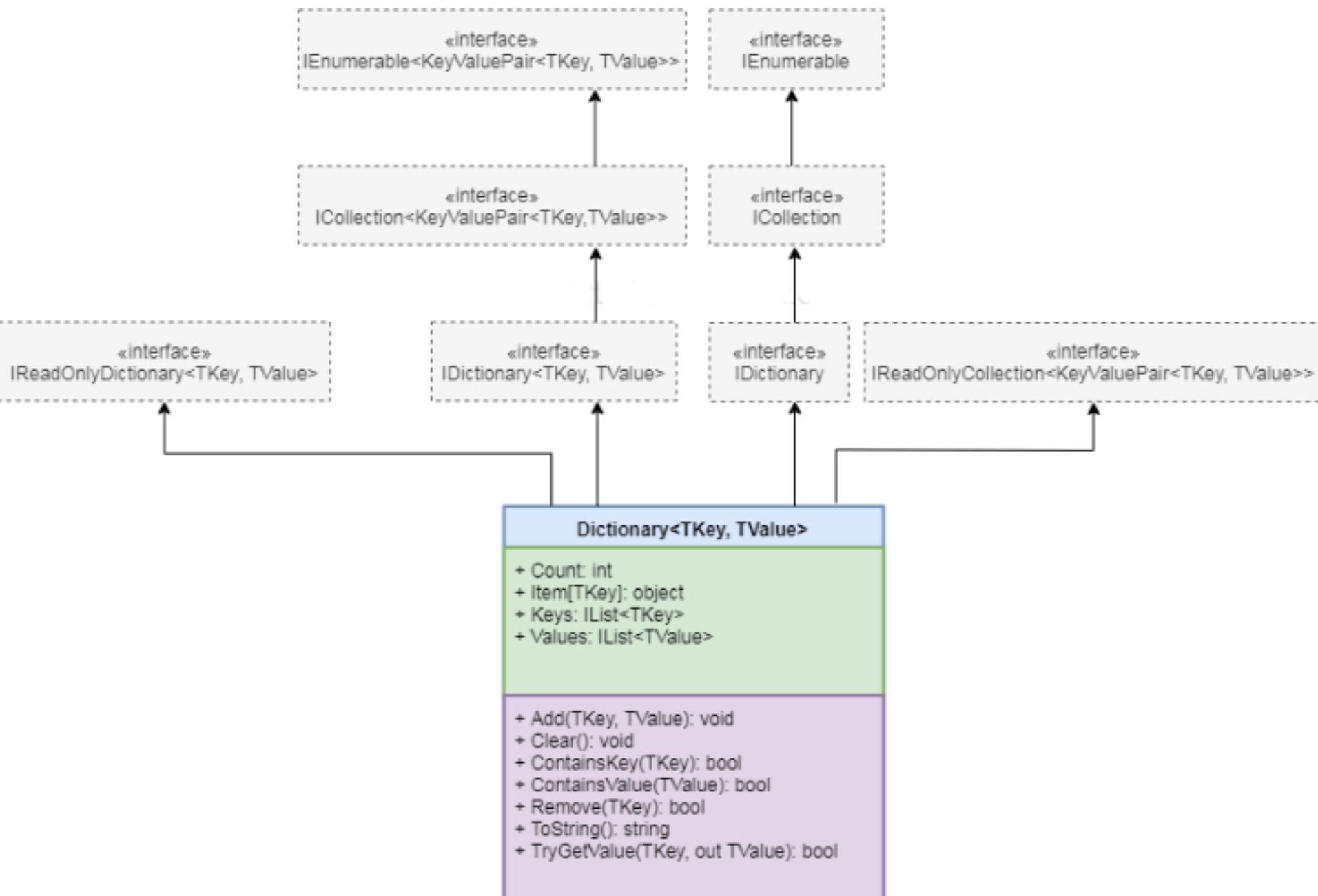
- 1) `List<T>` stores elements of the specified type and it grows automatically.
- 2) `List<T>` can store multiple null and duplicate elements.
- 3) `List<T>` can be assigned to **`ICollection<T>`** or **`IList<T>`** type of variable. It provides more helper method When assigned to `List<T>` variable
- 4) `List<T>` can be access using indexer, for loop or foreach statement.
- 5) LINQ can be use to query `List<T>` collection.
- 6) `List<T>` is ideal for storing and retrieving large number of elements.

C# - Dictionary<TKey, TValue>

The `Dictionary<TKey, TValue>` collection in C# is same as English dictionary. English dictionary is a collection of words and their definitions, often listed alphabetically in one or more specific languages. In the same way, the Dictionary in C# is a collection of Keys and Values, where key is like word and value is like definition.

The `Dictionary<TKey, TValue>` class is a generic collection class in the `System.Collections.Generic` namespace. TKey denotes the type of key and TValue is the type of TValue.

The following diagram illustrates the generic Dictionary class hierarchy.



A Dictionary object can be assigned to a variable of `IDictionary<Tkey, TValue>` or `Dictionary<TKey, TValue>` class.

Example: Dictionary Initialization

```
IDictionary<int, string> dict = new Dictionary<int, string>();  
  
//or  
  
Dictionary<int, string> dict = new Dictionary<int, string>();
```

In the above example, we have specified types of key and value while declaring a dictionary object. An int is a type of key and string is a type of value that will be stored into a dictionary object named `dict`. You can use any valid C# data type for keys and values.

It is recommended to program to the interface rather than to the class. So, use `IDictionary<TKey, TValue>` type variable to initialize a dictionary object.

Note:

Dictionary cannot include duplicate or null keys, where as values can be duplicated or set as null. Keys must be unique otherwise it will throw a runtime exception.

Important Properties and Methods of IDictionary

Property	Description
Count	Gets the total number of elements exists in the Dictionary<TKey,TValue>.
IsReadOnly	Returns a boolean indicating whether the Dictionary<TKey,TValue> is read-only.
Item	Gets or sets the element with the specified key in the Dictionary<TKey,TValue>.
Keys	Returns collection of keys of Dictionary<TKey,TValue>.
Values	Returns collection of values in Dictionary<TKey,TValue>.

Method	Description
Add	Adds an item to the Dictionary collection.
Add	Add key-value pairs in Dictionary<TKey, TValue> collection.
Remove	Removes the first occurrence of specified item from the Dictionary<TKey, TValue>.
Remove	Removes the element with the specified key.
ContainsKey	Checks whether the specified key exists in Dictionary<TKey, TValue>.
ContainsValue	Checks whether the specified key exists in Dictionary<TKey, TValue>.
Clear	Removes all the elements from Dictionary<TKey, TValue>.
TryGetValue	Returns true and assigns the value with specified key, if key does not exists then return false.

Add Elements into Dictionary

Use Add() method to add the key-value pair in dictionary.

Add() Signature: `void Add(TKey, TValue)`

Example: Add elements in dictionary

```
IDictionary<int, string> dict = new Dictionary<int, string>();  
dict.Add(1, "One");  
dict.Add(2, "Two");  
dict.Add(3, "Three");
```

Try it

The IDictionary type instance has one more overload for the Add() method. It accepts a `KeyValuePair<TKey, TValue>` struct as a parameter.

Add() Signature: `void Add(KeyValuePair<TKey, TValue> item);`

TIPS

Check whether a dictionary already stores specified key before adding a key-value pair.

Example: Add key-value pair in dictionary

```
IDictionary<int, string> dict = new Dictionary<int, string>();  
  
dict.Add(new KeyValuePair<int, string>(1, "One"));  
dict.Add(new KeyValuePair<int, string>(2, "Two"));  
  
//The following is also valid  
dict.Add(3, "Three");
```

It can also be initialized using collection initializer syntax with keys and values as shown below.

Example: Dictionary Initialization

```
IDictionary<int, string> dict = new Dictionary<int, string>()  
    {  
        {1, "One"},  
        {2, "Two"},  
        {3, "Three"}  
    };
```

Accessing Dictionary Elements

Dictionary elements can be accessed by many ways e.g. foreach, for loop or indexer.

Use foreach or for loop to iterate access all the elements of dictionary. The dictionary stores key-value pairs. So you can use a KeyValuePair<TKey, TValue> type or an implicitly typed variable var in foreach loop as shown below.

Example: Access elements using foreach

```
Dictionary<int, string> dict = new Dictionary<int, string>()
{
    {1, "One"},
    {2, "Two"},
    {3, "Three"}
};

foreach (KeyValuePair<int, string> item in dict)
{
    Console.WriteLine("Key: {0}, Value: {1}", item.Key, item.Value);
}
```

Use for loop to access all the elements. Use Count property of dictionary to get the total number of elements in the dictionary.

Example: Access Elements using for Loop

```
Dictionary<int, string> dict = new Dictionary<int, string>()
{
    {1, "One"},
    {2, "Two"},
    {3, "Three"}
};

for (int i = 0; i < dict.Count; i++)
{
    Console.WriteLine("Key: {0}, Value: {1}",
        dict.Keys.ElementAt(i),
        dict[ dict.Keys.ElementAt(i)]);
}
```

Dictionary can be used like an array to access its individual elements. Specify key (not index) to get a value from a dictionary using indexer like an array.

Example: Access Individual Element

```
Dictionary<int, string> dict = new Dictionary<int, string>()
{
    {1, "One"},
    {2, "Two"},
    {3, "Three"}
};

Console.WriteLine(dict[1]); //returns One
Console.WriteLine(dict[2]); // returns Two
```

Note:

Indexer takes the key as a parameter. If the specified key does not exist then a `KeyNotFoundException` will be thrown.

If you are not sure about the key then use the `TryGetValue()` method. The `TryGetValue()` method will return false if it could not find keys instead of throwing an exception.

`TryGetValue()` Signature: `bool TryGetValue(TKey key, out TValue value)`

Example: TryGetValue()

```
Dictionary<int, string> dict = new Dictionary<int, string>()
{
    {1, "One"},
    {2, "Two"},
    {3, "Three"}
};

string result;

if(dict.TryGetValue(4, out result))
{
    Console.WriteLine(result);
}
else
{
    Console.WriteLine("Could not find the specified key.");
}
```

Check for an Existing Elements

Dictionary includes various methods to determine whether a dictionary contains specified elements or keys. Use the `ContainsKey()` method to check whether a specified key exists in the dictionary or not.

Use the `Contains()` method to check whether a specified Key and Value pair exists in the dictionary or not.

`ContainsKey()` Signature: `bool ContainsKey(TKey key)`

`Contains()` signature: `bool Contains(KeyValuePair<TKey, TValue> item)`

Example: `ContainsKey()` & `Contains()`

```
Dictionary<int, string> dict = new Dictionary<int, string>()
{
    {1, "One"},
    {2, "Two"},
    {3, "Three"}
};

dict.ContainsKey(1); // returns true
dict.ContainsKey(4); // returns false

dict.Contains(new KeyValuePair<int, string>(1, "One")); // returns true
```

Another overload of the `Contains()` method takes `IEqualityComparer` as a second parameter. An instance of `IEqualityComparer` is used when you want to customize the equality comparison. For example, consider the following example of a dictionary that stores a `Student` objects.

Custom Comparer

```
public class Student
{
    public int StudentID { get; set; }
    public string StudentName { get; set; }
}

class StudentDictionaryComparer : IEqualityComparer<KeyValuePair<int, Student>>
{
    public bool Equals(KeyValuePair<int, Student> x, KeyValuePair<int, Student> y)
    {
        if (x.Key == y.Key && (x.Value.StudentID == y.Value.StudentID) && (x.Value.StudentName == y.Value.StudentName))
            return true;

        return false;
    }

    public int GetHashCode(KeyValuePair<int, Student> obj)
    {
        return obj.Key.GetHashCode();
    }
}
```

```
class Program
{
    static void Main(string[] args)
    {
        IDictionary<int, Student> studentDict = new Dictionary<int, Student>()
        {
            { 1, new Student(){ StudentID =1, StudentName = "Bill"}},
            { 2, new Student(){ StudentID =2, StudentName = "Steve"}},
            { 3, new Student(){ StudentID =3, StudentName = "Ram"}}
        };

        Student std = new Student(){ StudentID = 1, StudentName = "Bill"};

        KeyValuePair<int, Student> elementToFind = new KeyValuePair<int, Student>(1, std);

        bool result = studentDict.Contains(elementToFind, new StudentDictionaryComparer()); // returns true

        Console.WriteLine(result);
    }
}
```


In the above example, we have used `StudentDictionaryComparer` which derives `IEqualityComparer` to compare `Student` objects in the dictionary. The default comparer will only work with primitive data types.

Remove Elements in Dictionary

Use the `Remove()` method to remove an existing item from the dictionary. `Remove()` has two overloads, one overload method accepts a key and the other overload method accepts a `KeyValuePair<>` as a parameter.

`Remove()` signature:

- `bool Remove(TKey key)`
- `bool Remove(KeyValuePair<TKey, TValue>)`

Example: Remove elements from Dictionary


```
Dictionary<int, string> dict = new Dictionary<int, string>()
{
    {1, "One"},
    {2, "Two"},
    {3, "Three"}
};

dict.Remove(1); // removes the item which has 1 as a key
```

Both Key and Value must match to remove an item. The item will not be removed if both are not matched. For example, the following example will not remove any item:


```
// removes nothing because value One1 is not matching
dict.Remove(new KeyValuePair<int, string>(2, "Two1"));
```

Use generic [SortedDictionary](#) collection if you want to sort a dictionary collection based on the keys.

 **Points to Remember :**

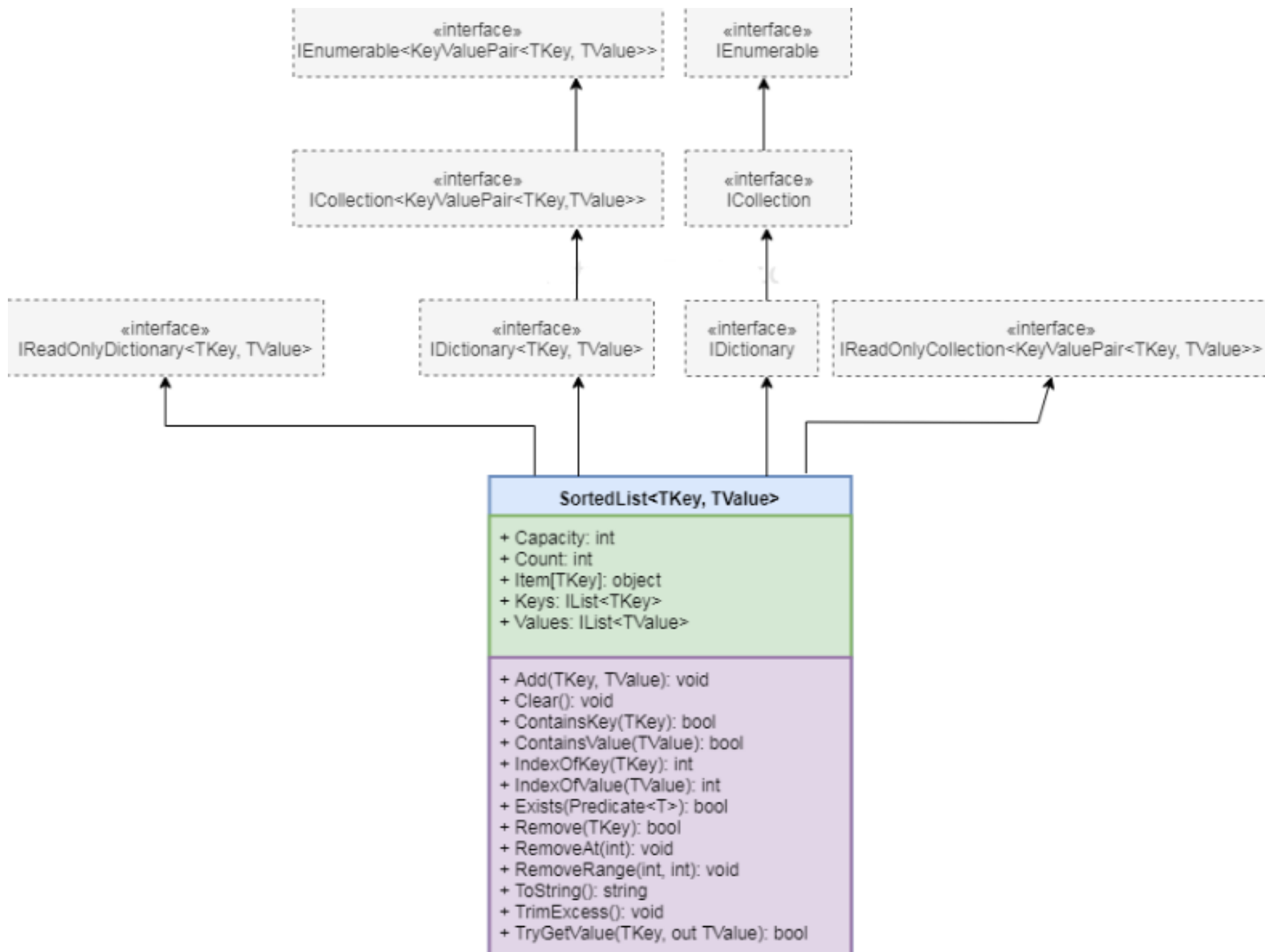
- 1) A Dictionary stores Key-Value pairs where the key must be unique.
- 2) Before adding a KeyValuePair into a dictionary, check that the key does not exist using the ContainsKey() method.
- 3) Use the TryGetValue() method to get the value of a key to avoid possible runtime exceptions.
- 4) Use a foreach or for loop to iterate a dictionary.
- 5) Use dictionary indexer to access individual item.
- 6) Use custom class that derives IEqualityComparer to compare object of custom class with Contains() method.

C# - SortedList<TKey, TValue>

The generic SortedList `SortedList<TKey, TValue>` represents a collection of key-value pairs that are sorted by key based on associated [IComparer<T>](#) . A SortedList collection stores key and value pairs in ascending order of key by default.

C# includes two type of SortedList, generic SortedList and [non-generic SortedList](#). Generic SortedList denotes with angel bracket: `SortedList<TKey, TValue>` where `TKey` is for type of key and `TValue` is for type of value. Non-generic type do not specify the type of key and values.

The following diagram illustrates the generic SortedList hierarchy.



Internally, SortedList maintains a two object[] array, one for keys and another for values. So when you add key-value pair, it does binary search using key to find an appropriate index to store a key and value in respective arrays. It also re-arranges the elements when you remove the elements from it.

You can instantiate `SortedList<TKey, TValue>` by specifying type for key and value, as shown below.

Example: Instantiate Generic SortedList

```
SortedList<int, string> mySortedList = new SortedList<int, string>();
```

In the above example, `mySortedList` will store the keys of int type and the values of string type.

Important Properties and Methods of Generic SortedList

Property	Description
Capacity	Gets or sets the number of elements that the SortedList<TKey,TValue> can store.
Count	Gets the total number of elements exists in the SortedList<TKey,TValue>.
IsReadOnly	Returns a boolean indicating whether the SortedList<TKey,TValue> is read-only.
Item	Gets or sets the element with the specified key in the SortedList<TKey,TValue>.
Keys	Get list of keys of SortedList<TKey,TValue>.
Values	Get list of values in SortedList<TKey,TValue>.

Method	Description
Add	Add key-value pairs into SortedList<TKey, TValue>.
Remove	Removes element with the specified key.
RemoveAt	Removes element at the specified index.
ContainsKey	Checks whether the specified key exists in SortedList<TKey, TValue>.
ContainsValue	Checks whether the specified key exists in SortedList<TKey, TValue>.
Clear	Removes all the elements from SortedList<TKey, TValue>.
IndexOfKey	Returns an index of specified key stored in internal array of SortedList<TKey, TValue>.
IndexOfValue	Returns an index of specified value stored in internal array of SortedList<TKey, TValue>
TryGetValue	Returns true and assigns the value with specified key, if key does not exists then return false.

Add Elements into SortedList

Use the `Add()` method to add key value pairs into a `SortedList`. The key cannot be null, but the value can be null. Also, the datatype of key and value must be same as specified, otherwise it will give compile time error.

Add() method signature: `void Add(TKey key, TValue value)`

The following example shows how to add key-value pair in the generic `SortedList` collection.

Example:Add Elements into SortedList<TKey, TValue>

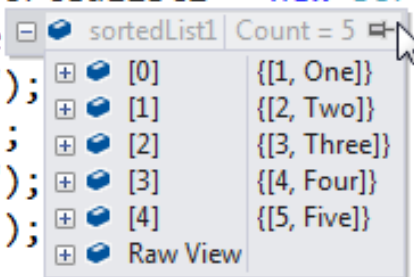
```
SortedList<int,string> sortedList1 = new SortedList<int,string>();
sortedList1.Add(3, "Three");
sortedList1.Add(4, "Four");
sortedList1.Add(1, "One");
sortedList1.Add(5, "Five");
sortedList1.Add(2, "Two");

SortedList<string,int> sortedList2 = new SortedList<string,int>();
sortedList2.Add("one", 1);
sortedList2.Add("two", 2);
sortedList2.Add("three", 3);
sortedList2.Add("four", 4);
// Compile time error: cannot convert from <null> to <int>
// sortedList2.Add("Five", null);

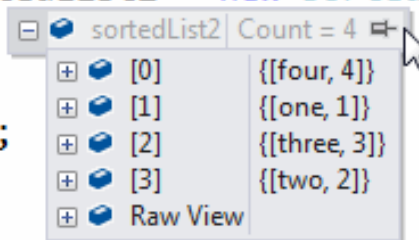
SortedList<double,int?> sortedList3 = new SortedList<double,int?>();
sortedList3.Add(1.5, 100);
sortedList3.Add(3.5, 200);
sortedList3.Add(2.4, 300);
sortedList3.Add(2.3, null);
sortedList3.Add(1.1, null);
```

SortedList collection sorts the elements everytime you add the elements. So if you debug the above example, you will keys in ascending order even if they are added randomly. The following image shows SortedList in debug view.

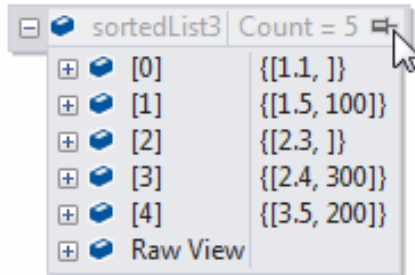
```
SortedList<int, string> sortedList1 = new SortedList<int, string>();
sortedList1.Add(3, "Three");
sortedList1.Add(4, "Four");
sortedList1.Add(1, "One");
sortedList1.Add(5, "Five");
sortedList1.Add(2, "Two");
```



```
SortedList<string, int> sortedList2 = new SortedList<string, int>();
sortedList2.Add("one", 1);
sortedList2.Add("two", 2);
sortedList2.Add("three", 3);
sortedList2.Add("four", 4);
```



```
SortedList<double, int?> sortedList3 = new SortedList<double, int?>();
sortedList3.Add(1.5, 100);
sortedList3.Add(3.5, 200);
sortedList3.Add(2.4, 300);
sortedList3.Add(2.3, null);
sortedList3.Add(1.1, null);
```



SortedList<TKey, TValue> in debug view

As you can see in the above image, sortedList1 stores key-value pairs in ascending order of key and sortedList2 stores items in alphabetical order of key even if they are not added in that order. sortedList3 includes nullable int so that it includes null as a value.

Key/value pairs can also be added using object-initializer syntax, as shown below.

Example: Initialize SortedList<TKey, TValue>

```
SortedList<int,string> sortedList1 = new SortedList<int,string>()  
    {  
        {3, "Three"},  
        {4, "Four"},  
        {1, "One"},  
        {5, "Five"},  
        {2, "Two"}}  
    };
```

Accessing Generic SortedList

The SortedList can be accessed by the index or key. Unlike other collection types, Indexer of SortedList requires key and returns value for that key. However, please make sure that key exists in the SortedList, otherwise it will throw KeyNotFoundException.

Example: Access SortedList<TKey, TValue> using indexer

```
SortedList<string,int> sortedList2 = new SortedList<string,int>();
sortedList2.Add("one", 1);
sortedList2.Add("two", 2);
sortedList2.Add("three", 3);
sortedList2.Add("four", 4);

Console.WriteLine(sortedList2["one"]);
Console.WriteLine(sortedList2["two"]);
Console.WriteLine(sortedList2["three"]);

//Following will throw runtime exception: KeyNotFoundException
Console.WriteLine(sortedList2["ten"]);
```

Keys and Values indexers can use the access key and value of SortedList using for loop as shown below:

0 references

```
SortedList<string,int> sortedList2 = new SortedList<string,int>();
sortedList2.Add("one", 1);
sortedList2.Add("two", 2);
sortedList2.Add("three", 3);
sortedList2.Add("four", 4);

for (int i = 0; i < sortedList2.Count; i++)
{
    Console.WriteLine("key: {0}, value: {1}", sortedList2.Keys[i], sortedList2.Values[i]);
}
```

Accessing Elements using foreach loop

The foreach statement in C# can be used to access the SortedList collection. SortedList element includes both key and value pair. so, the type of element would be `KeyValuePair` structure rather than type of key or value.

foreach statement to access generic SortedList:

```
SortedList<string,int> sortedList2 = new SortedList<string,int>();
sortedList2.Add("one", 1);
sortedList2.Add("two", 2);
sortedList2.Add("three", 3);
sortedList2.Add("four", 4);

foreach(KeyValuePair<string,int> kvp in sortedList2 )
    Console.WriteLine("key: {0}, value: {1}", kvp.Key , kvp.Value );
```

Accessing key

If you are not sure that particular key exists or not than use TryGetValue method to retrieve the value of specified key. If key doesn't exists than it will return false instead of throwing exception.

Example: TryGetValue

```
SortedList<string,int> sortedList2 = new SortedList<string,int>();
sortedList2.Add("one", 1);
sortedList2.Add("two", 2);
sortedList2.Add("three", 3);
sortedList2.Add("four", 4);

int val;

if (sortedList2.TryGetValue("ten",out val))
    Console.WriteLine("value: {0}", val);
else
    Console.WriteLine("Key is not valid.");

if (sortedList2.TryGetValue("one",out val))
    Console.WriteLine("value: {0}", val);
```

Remove Elements from Generic SortedList

Use the Remove(key) and RemoveAt(index) methods to remove values from a SortedList.

Remove() signature: `bool Remove(TKey key)`

RemoveAt() signature: `void RemoveAt(int index)`

Example: Remove Elements

```
SortedList<string,int> sortedList2 = new SortedList<string,int>();
sortedList2.Add("one", 1);
sortedList2.Add("two", 2);
sortedList2.Add("three", 3);
sortedList2.Add("four", 4);

sortedList2.Remove("one");//removes the element whose key is 'one'
sortedList2.RemoveAt(0);//removes the element at zero index i.e first element: four

foreach(KeyValuePair<string,int> kvp in sortedList2 )
    Console.WriteLine("key: {0}, value: {1}", kvp.Key , kvp.Value );
```


ContainsKey() and ContainsValue()

The ContainsKey() checks whether the specified key exists in the SortedList or not.

ContainsKey() signature: `bool ContainsKey(object key)`

The ContainsValue() method determines whether the specified value exists in the SortedList or not.

ContainValue() signature: `bool ContainValue(object value)`

Example: Contain()

```
SortedList<string,int> sortedList = new SortedList<string,int>();
sortedList.Add("one", 1);
sortedList.Add("two", 2);
sortedList.Add("three", 3);
sortedList.Add("four", 4);
sortedList.Add("five", 5);

sortedList.ContainsKey("One"); // returns true
sortedList.ContainsKey("Ten"); // returns false

sortedList.ContainsValue(2); // returns true
sortedList.ContainsValue(6); // returns false
```

Accessing SortedList using LINQ

You can use LINQ query syntax or method syntax to access SortedList collection using different criterias.

Example: Access SortedList using LINQ Method Syntax

```
SortedList<string,int> sortedList = new SortedList<string,int>();
sortedList.Add("one", 1);
sortedList.Add("two", 2);
sortedList.Add("three", 3);
sortedList.Add("four", 4);
sortedList.Add("five", 5);

var result = sortedList.Where(kvp => kvp.Key == "two").FirstOrDefault();

Console.WriteLine("key: {0}, value: {1}", result.Key, result.Value);
```

Example: Accessing Generic SortedList using LINQ Query Syntax

```
SortedList<string,int> sortedList = new SortedList<string,int>();
sortedList.Add("one", 1);
sortedList.Add("two", 2);
sortedList.Add("three", 3);
sortedList.Add("four", 4);
sortedList.Add("five", 5);

var query = from kvp in sortedList
            where kvp.Key == "two"
            select kvp;

var result = query.FirstOrDefault();

Console.WriteLine("key: {0}, value: {1}", result.Key, result.Value);
```



Points to Remember :

- 1) C# has a generic and non-generic SortedList.
- 2) SortedList stores the key-value pairs in ascending order of the key. The key must be unique and cannot be null whereas value can be null or duplicate.
- 3) Generic SortedList stores keys and values of specified data types. So no need for casting.
- 4) Key-value pair can be cast to a `KeyValuePair<TKey,TValue>`.
- 5) An individual value can be accessed using an indexer. SortedList indexer accepts key to return value associated with it.