**Generic**is a class which allows the user to define classes and methods with the placeholder. Generics were added to version 2.0 of the C# language. The basic idea behind using Generic is to allow type (Integer, String, … etc and user-defined types) to be a parameter to methods, classes, and interfaces. A primary limitation of collections is the absence of effective type checking. This means that you can put any object in a collection because all classes in the C# programming language extend from the object base class. This compromises type safety and contradicts the basic definition of C# as a type-safe language. In addition, using collections involves a significant performance overhead in the form of implicit and explicit type casting that is required to add or retrieve objects from a collection.  
To address the type safety issue, the [**.NET**](https://www.geeksforgeeks.org/c-net-framework-basic-architecture-component-stack/)framework provides generics to create classes, structures, interfaces, and methods that have placeholders for the types they use. Generics are commonly used to create type-safe collections for both reference and value types. The [**.NET**](https://www.geeksforgeeks.org/c-net-framework-basic-architecture-component-stack/) framework provides an extensive set of interfaces and classes in the System.Collections.Generic namespace for implementing generic collections.

**Generic Class**

Generics in C# is its most powerful feature. It allows you to define the type-safe data structures. This out-turn in a remarkable performance boost and high-grade code, because it helps to reuse data processing algorithms without replicating type-specific code. Generics are similar to templates in C++ but are different in implementation and capabilities. Generics introduces the concept of type parameters, because of which it is possible to create methods and classes that defers the framing of data type until the class or method is declared and is instantiated by client code. Generic types perform better than normal system types because they reduce the need for boxing, unboxing, and type casting the variables or objects.  
Parameter types are specified in generic class creation.

Boxing and Unboxing

.Net defines two major categories of data type termed value type and reference type to represent a variable. This is where boxing and unboxing are needed. Boxing is a mechanism to explicitly convert a value type to a reference type by storing the variable into System.Object; when you box the value the CLR allocates a new object into the heap and copies the value type's value into that instance. For example you have created a variable of int type as:

1. **int** a = 20;
2. **object** b = a; //boxing

The opposite operation is Unboxing which is the process of converting back the reference type into the value type. This process verifies that the receiving data type is equivalent to the boxed type as;

1. **int** c = (**int**)b; // unboxing

The C# compiler sees the assignment from int to object and vice-versa. When this program is compiled and you examine the IL generated code via IL dissembler, you notice that the program respond by inserting a box instruction in the IL automatically when b is assigned the value of a and an unbox instruction when c is assigned the value b as in the following;

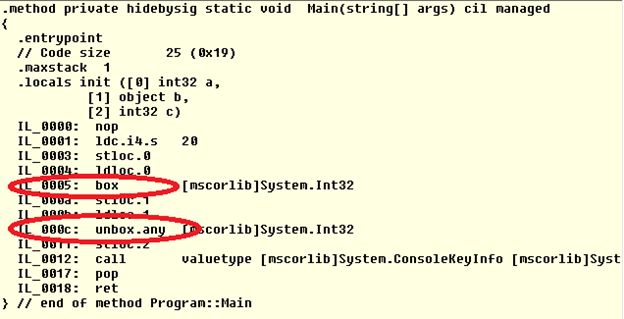


Figure 1.1 - IL opcode

The code loads the constant 20 and stores it in the local slot; it the loads the value 20 onto the stack and boxes it. Finally it loads the boxed 20 back onto the stack and unboxes it into an int.

There are  series of operations performed by .NET CLR, such as, first an object is allocated in the managed heap, then in boxing the value is transformed into the memory location and during unboxing the value is stored on the heap and must be transferred back to the stack. So the Boxing and Unboxing process has a significant importance in Generics from the performance point of view because this process is more resource-intensive rather than using Generics.

Generic Classes

The Generic class can be defined by putting the <T> sign after the class name. It isn't mandatory to put the "T" word in the Generic type definition. You can use any word in the TestClass<> class declaration.

1. **public** **class** TestClass<T> { }

The System.Collection.Generic namespace also defines a number of classes that implement many of these key interfaces. The following table describes the core class types of this namespace.

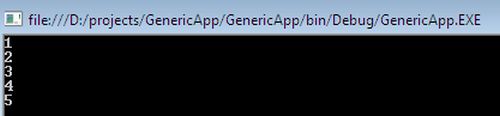
|  |  |
| --- | --- |
| Generic class | Description |
| Collection<T> | The basis for a generic collection Comparer compares two generic objects for equality |
| Dictionary<TKey, TValue> | A generic collection of name/value pairs |
| List<T> | A dynamically resizable list of Items |
| Queue<T> | A generic implementation of a first-in, first-out (FIFO) list |
| Stack<T> | A generic implementation of a last-in, first-out (LIFO) list |

**Simple Generic Class Example**

The following example shows a simple Generic type manipulation. The TestClass<T> defines an array of generic type with length 5. The Add() method is responsible for adding any type of objects into the collection and the Indexer property is an implementation of foreach statement iteration. Finally in the main class we instantiated the TestClass<T> class with an Integer type reference and adds some integer type elements into the collection using the Add() method.

1. **using** System;
2. **using** System.Collections.Generic;
4. **namespace** GenericApp
5. {
6. **public** **class** TestClass<T>
7. {
8. // define an Array of Generic type with length 5
9. T[] obj = **new** T[5];
10. **int** count = 0;
12. // adding items mechanism into generic type
13. **public** **void**  Add(T item)
14. {
15. //checking length
16. **if** (count + 1 < 6)
17. {
18. obj[count] = item;
20. }
21. count++;
22. }
23. //indexer for foreach statement iteration
24. **public** T **this**[**int** index]
25. {
26. **get** { **return** obj[index]; }
27. **set** { obj[index] = value; }
28. }
29. }
30. **class** Program
31. {
32. **static** **void** Main(**string**[] args)
33. {
34. //instantiate generic with Integer
35. TestClass<**int**> intObj = **new** TestClass<**int**>();
37. //adding integer values into collection
38. intObj.Add(1);
39. intObj.Add(2);
40. intObj.Add(3);     //No boxing
41. intObj.Add(4);
42. intObj.Add(5);
44. //displaying values
45. **for** (**int** i = 0; i < 5; i++)
46. {
47. Console.WriteLine(intObj[i]);   //No unboxing
48. }
49. Console.ReadKey();
50. }
51. }
52. }

After building and running this program, the output of the program is as shown in the following;



There are some significant characteristics of Generic types that make them special to the conventional non-generics type as follows;

* Type Safety
* Performance
* Binary Code reuse

**Type Safety**

One of the most significant features of Generics is Type Safety. In the case of the non-generic ArrayList class, if objects are used, any type can be added to the collections that can sometimes result in a great disaster. The following example shows adding an integer, string and object to the collection of an ArrayList type;

1. ArrayList obj = **new** ArrayList();
2. obj.Add(50);
3. obj.Add("Dog");
4. obj.Add(**new** TestClass());

Now, if the collection is iterated through the foreach statement using integer elements, the compiler accepts the code but because all the elements in the collection are not an integer, a runtime exception occurs;

1. **foreach**(**int** i **in** obj)
2. {
3. Console.WriteLine(i);
4. }

The rule of thumb in programming is that Errors should be detected as early as possible. With the generic class Test<T>, the generic type T defines what types are allowed. With the definition of Test<int>, only an integer type can be added to the collection. The compiler doesn't compile the code because the Add() method has invalid arguments as follows;

1. Test<**int**> obj = **new** Test<**int**>();
2. obj.Add(50);
3. obj.Add("Dog");            //compiler error
4. obj.Add(**new** TestClass());  //compiler error

**Performance**

Another feature of Generics is performance. Using value types with non-generic collection classes result in boxing and unboxing overhead when a value type is converted to reference type and vice-versa.

In the following example, the ArrayList class stores objects and the Add() method is defined to store some integer type argument. So an integer type is boxed. When the value from ArrayList is read using the foreach statement, unboxing occurrs.

1. ArrayList  obj = **new** ArrayList();
2. obj.Add(50);    //boxing- convert value type to reference type
3. **int** x= (**int**)obj[0]; //unboxing
4. **foreach**(**int** i **in** obj)
5. {
6. Console.WriteLine(i);   // unboxing
7. }

**Note:** Generics are faster than other collections such as ArrayList.

Instead of using objects, a Generics type of the TestClass<T> class is defined as an int, so an int type is used inside the class that is generated dynamically from the compiler. Therefore boxing and unboxing no longer occurs as in the following;

1. TestClass<**int**> obj = **new** TestClass<**int**>();
2. obj.Add(50);    //No boxing
3. **int** x= obj[0]; // No unboxing
4. **foreach**(**int** i **in** obj)
5. {
6. Console.WriteLine(i);   //No unboxing
7. }

**Binary Code reuse**

Generics provide a kind of source code protection. A Generic class can be defined once and can be instantiated with many different types. Generics can be defined in one CLR supported language and used from another .NET language. The following TestClass<T> is instantiated with an int and string types:

1. TestClass<**int**> obj = **new** TestClass<**int**>();
2. obj.Add(50);
4. TestClass<**string**> obj1 = **new** TestClass<**string**>();
5. Obj1.Add("hello");

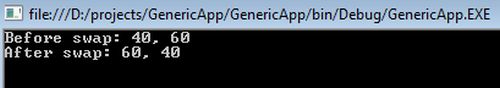
Generic Methods

While most developers will typically use the existing generic types within the base class libraries, it is certainly possible to build your own generic members and custom generic types.

The objective of this example is to build a swap method that can operate on any possible data type (value-based or reference-based) using a single type parameter. Due to the nature of swapping algorithms, the incoming parameters will be sent by reference via ref keyword.

1. **using** System;
2. **using** System.Collections.Generic;
4. **namespace** GenericApp
5. {
6. **class** Program
7. {
8. //Generic method
9. **static** **void** Swap<T>(**ref** T a, **ref** T b)
10. {
11. T temp;
12. temp = a;
13. a = b;
14. b = temp;
15. }
16. **static** **void** Main(**string**[] args)
17. {
18. // Swap of two integers.
19. **int** a = 40, b = 60;
20. Console.WriteLine("Before swap: {0}, {1}", a, b);
22. Swap<**int**>(**ref** a, **ref** b);
24. Console.WriteLine("After swap: {0}, {1}", a, b);
26. Console.ReadLine();
27. }
28. }
29. }

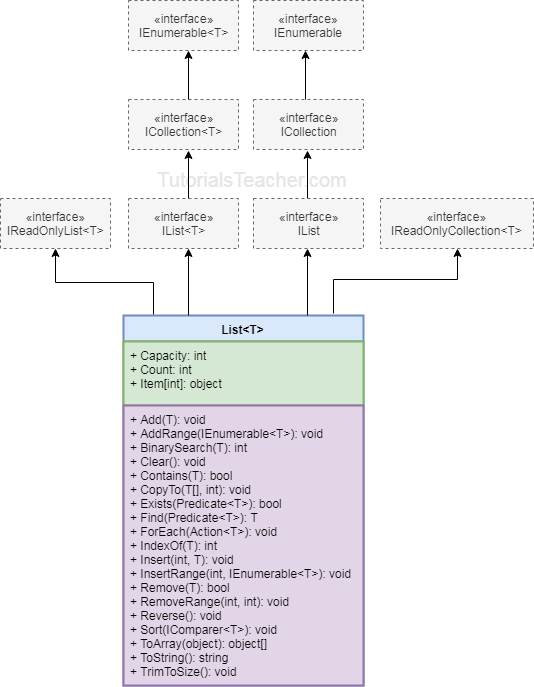
After compiling this Generic method implementation program, the output is as in the following;



# 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.

[](https://www.tutorialsteacher.com/Content/images/csharp/list.png)

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 concreate implementation of IList<T> interface. In the object-oriented programming, it is advisable to program to interface rather than concreate class. So use [IList<T>](http://msdn.microsoft.com/en-us/library/5y536ey6(v=vs.110).aspx)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 List<T> |

| 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 speciied 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 occurence 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 theÂ 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());

AddRange

IList<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

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

intList.ForEach(el => Console.WriteLine(el));

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

foreach (var el in intList)

Console.WriteLine(el);

Accessing List

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

int elem = intList[1]; // returns 20

Access List elements

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

Console.Write("Total elements: {0}", intList.Count);

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]);

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);

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](https://www.tutorialsteacher.com/csharp/csharp-predicate) type deligate or the [lambda expression](https://www.tutorialsteacher.com/linq/linq-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(el => el%2 == 0);// returns true

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);

}

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 **IList<T>** or **List<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 class provides functionality to work with a dictionary. A dictionary is a collection of a key and value pair objects. Dictionary<TKey,TValue> in C# is a generic class and can be used to create a collection of objects. Dictionary.Keys and Dictionary.Values represents the collection of keys and values respectively. The Dictionary type represents a collection of keys and values pair of data.

The Dictionary class defined in the System.Collections.Generic namespace is a generic class and can store any data types in a form of keys and values. Each key must be unique in the collection.

Dictionary Initialization

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

//or

Dictionary<int, string> dict = new Dictionary<int, string>();

Add elements in dictionary

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

dict.Add(1,"One");

dict.Add(2,"Two");

dict.Add(3,"Three");

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");

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);

}

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)]);

}

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

Example: TryGetValue()

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

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

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

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

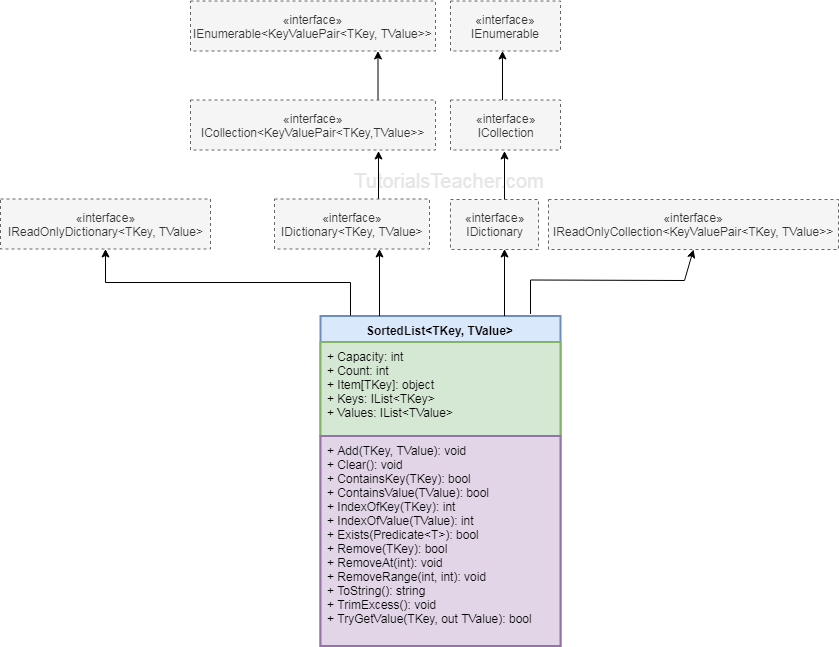
1. A Dictionary stores Key-Value pairs where the key must be unique.
2. Before adding a KeyValuePair into a dictionary, check that the key does not exist using the ContainsKey() method.
3. Use the TryGetValue() method to get the value of a key to avoid possible runtime exceptions.
4. Use a foreach or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

# 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>](https://msdn.microsoft.com/en-us/library/8ehhxeaf(v=vs.110).aspx). 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](https://www.tutorialsteacher.com/csharp/csharp-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.

[](https://www.tutorialsteacher.com/Content/images/csharp/generic-sortedlist-hierarchy.png)

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);

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"]);

Access Key and Value 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);

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](http://msdn.microsoft.com/en-us/library/5tbh8a42(v=vs.110).aspx) 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

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.