Generics

Generics in C# allow you to write flexible and reusable code by creating classes, structures, interfaces, and methods that can work with any data type. They provide type safety without sacrificing flexibility. Here's an overview of how generics work:

Basic Syntax:

```
class ClassName<T>
{
    // Members, methods, properties, etc.
}
```

T is a type parameter, representing any data type. You can use any valid identifier instead of T.

Example:

```
public class GenericList<T>
{
    private T[] _items;
    private int _currentIndex = 0;

    public GenericList(int capacity)
    {
        _items = new T[capacity];
    }

    public void Add(T item)
    {
        if (_currentIndex < _items.Length)
        {
            _items[_currentIndex] = item;
            _currentIndex++;
        }
    }

    public T GetItem(int index)
    {
        if (index >= 0 && index < _items.Length)
        {
            return _items[index];
        }
        throw new IndexOutOfRangeException();
    }
}
```

Usage:

```
GenericList<int> intList = new GenericList<int>(5);
intList.Add(1);
intList.Add(2);
intList.Add(3);

int item = intList.GetItem(1); // item = 2
```

```
GenericList<string> stringList = new GenericList<string>(3);
stringList.Add("apple");
stringList.Add("banana");

string fruit = stringList.GetItem(0); // fruit = "apple"
```

Constraints:

You can apply constraints on generic type parameters to specify capabilities that generic types must have. Common constraints include where T : class, where T : struct, where T : interface, where T : new() (parameterless constructor), and custom constraints.

Example with Constraints:

```
public class MyClass<T> where T : IDisposable
{
    // Methods can use IDisposable methods on T
}
```

Generics are extensively used in .NET framework collections (like List<T>), LINQ, and various other scenarios where type safety and code reuse are essential. They help to avoid code duplication and increase the flexibility and maintainability of your codebase.

Boxing and Unboxing

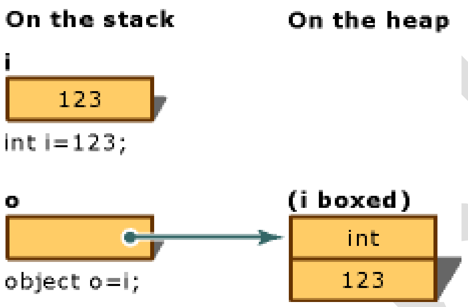
Boxing

Boxing is the process of converting a value type to the type object or to any interface type implemented by this value type. Boxing a value type allocates an object instance on the heap and copies the value into the new object.

The following statement implicitly applies the boxing operation on the variable i:

```
int i = 123;
// The following line boxes i.
object o = i;
```

The result of this statement is creating an object reference o, on the stack, that references a value of the type int, on the heap. This value is a copy of the value-type value assigned to the variable i. The difference between the two variables, i and o, is illustrated in the following image of boxing conversion



It is also possible to perform the boxing explicitly as in the following example, but explicit boxing is never required:

```
int i = 123;
object o = (object)i;  // explicit boxing
```

Unboxing

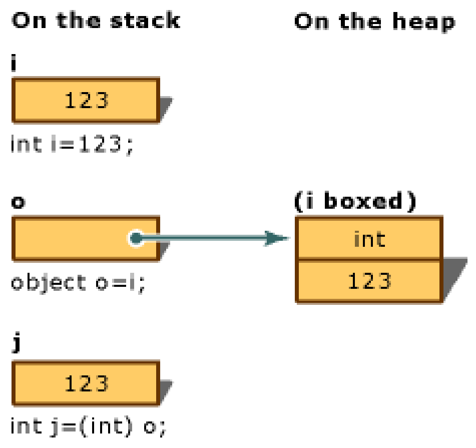
Unboxing is an explicit conversion from the type object to a value type or from an interface type to a value type that implements the interface. An unboxing operation consists of:

- Checking the object instance to make sure that it is a boxed value of the given value type.
- Copying the value from the instance into the value-type variable.

The following statements demonstrate both boxing and unboxing operations:

```
int i = 123;      // a value type
object o = i;     // boxing
```

```
int j = (int)o;    // unboxing
```



Arrays

You can store multiple variables of the same type in an array data structure. You declare an array by specifying the type of its elements. If you want the array to store elements of any type, you can specify object as its type.

```
type[] arrayName;
```

An array has the following properties:

- An array can be single-dimensional, multidimensional, or jagged.
- The number of dimensions are set when an array variable is declared. The length of each dimension is established when the array instance is created. These values can't be changed during the lifetime of the instance.
- A jagged array is an array of arrays, and each member array has the default value of null.
- Arrays are zero indexed: an array with n elements is indexed from 0 to n-1.
- Array elements can be of any type, including an array type.
- Array types are reference types derived from the abstract base type Array. All arrays implement IList and IEnumerable. You can use the foreach statement to iterate through an array. Single-dimensional arrays also implement IList and IEnumerable.

```
// Declare a single-dimensional array of 5 integers.
int[] array1 = new int[5];

// Declare and set array element values.
int[] array2 = [1, 2, 3, 4, 5, 6];

// Declare a two dimensional array.
int[,] multiDimensionalArray1 = new int[2, 3];

// Declare and set array element values.
int[,] multiDimensionalArray2 = { { 1, 2, 3 }, { 4, 5, 6 } };
```

```
// Declare a jagged array.
int[][] jaggedArray = new int[6][];

// Set the values of the first array in the jagged array structure.
jaggedArray[0] = [1, 2, 3, 4];
```

Single-dimensional arrays

A single-dimensional array is a sequence of like elements. You access an element via its index. The index is its ordinal position in the sequence. The first element in the array is at index 0. You create a single-dimensional array using the new operator specifying the array element type and the number of elements.
The following example declares and initializes single-dimensional arrays:

```
int[] array = new int[5];
string[] weekDays = ["Sun", "Mon", "Tue", "Wed", "Thu", "Fri", "Sat"];

Console.WriteLine(weekDays[0]);
Console.WriteLine(weekDays[1]);
Console.WriteLine(weekDays[2]);
Console.WriteLine(weekDays[3]);
Console.WriteLine(weekDays[4]);
Console.WriteLine(weekDays[5]);
Console.WriteLine(weekDays[6]);

/*Output:
Sun
Mon
Tue
Wed
Thu
Fri
Sat
*/
```

he first declaration declares an uninitialized array of five integers, from array[0] to array[4]. The elements of the array are initialized to the default value of the element type, 0 for integers. The second declaration declares an array of strings and initializes all seven values of that array. A series of Console.WriteLine statements prints all the elements of the weekDay array.

Multidimensional arrays

Arrays can have more than one dimension. For example, the following declarations create four arrays: two have two dimensions, two have three dimensions. The first two declarations declare the length of each dimension, but don't initialize the values of the array. The second two declarations use an initializer to set the values of each element in the multidimensional array.

```
int[,] array2DDeclaration = new int[4, 2];

int[,,] array3DDeclaration = new int[4, 2, 3];

// Two-dimensional array.
int[,] array2DInitialization =  { { 1, 2 }, { 3, 4 }, { 5, 6 }, { 7, 8 } };
// Three-dimensional array.
int[,,] array3D = new int[,,] { { { 1, 2, 3 }, { 4,   5,  6 } },
                                { { 7, 8, 9 }, { 10, 11, 12 } } };

// Accessing array elements.
System.Console.WriteLine(array2DInitialization[0, 0]);
System.Console.WriteLine(array2DInitialization[0, 1]);
System.Console.WriteLine(array2DInitialization[1, 0]);
System.Console.WriteLine(array2DInitialization[1, 1]);

System.Console.WriteLine(array2DInitialization[3, 0]);
System.Console.WriteLine(array2DInitialization[3, 1]);
// Output:
// 1
// 2
// 3
// 4
// 7
// 8

System.Console.WriteLine(array3D[1, 0, 1]);
System.Console.WriteLine(array3D[1, 1, 2]);
// Output:
// 8
// 12

// Getting the total count of elements or the length of a given dimension.
var allLength = array3D.Length;
var total = 1;
for (int i = 0; i < array3D.Rank; i++)
{
    total *= array3D.GetLength(i);
}
System.Console.WriteLine($"{allLength} equals {total}");
// Output:
// 12 equals 12
```

For multi-dimensional arrays, elements are traversed such that the indices of the rightmost dimension are incremented first, then the next left dimension, and so on, to the leftmost index. The following example enumerates both a 2D and a 3D array:

```
int[,] numbers2D = { { 9, 99 }, { 3, 33 }, { 5, 55 } };

foreach (int i in numbers2D)
```

```
{
    System.Console.Write($"{i} ");
}
// Output: 9 99 3 33 5 55

int[,,] array3D = new int[,,] { { { 1, 2, 3 }, { 4,   5,  6 } },
                        { { 7, 8, 9 }, { 10, 11, 12 } } };
foreach (int i in array3D)
{
    System.Console.Write($"{i} ");
}
// Output: 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12
```

In a 2D array, you can think of the left index as the row and the right index as the column. However, with multidimensional arrays, using a nested for loop gives you more control over the order in which to process the array elements:

```
int[,,] array3D = new int[,,] { { { 1, 2, 3 }, { 4,   5,  6 } },
                        { { 7, 8, 9 }, { 10, 11, 12 } } };

for (int i = 0; i < array3D.GetLength(0); i++)
{
    for (int j = 0; j < array3D.GetLength(1); j++)
    {
        for (int k = 0; k < array3D.GetLength(2); k++)
        {
            System.Console.Write($"{array3D[i, j, k]} ");
        }
        System.Console.WriteLine();
    }
    System.Console.WriteLine();
}
// Output (including blank lines):
// 1 2 3
// 4 5 6
//
// 7 8 9
// 10 11 12
//
```

Jagged arrays

A jagged array is an array whose elements are arrays, possibly of different sizes. A jagged array is sometimes called an "array of arrays." Its elements are reference types and are initialized to null. The following examples show how to declare, initialize, and access jagged arrays. The first example, jaggedArray, is declared in one statement. Each contained array is created in subsequent statements. The second example, jaggedArray2 is declared and initialized in one statement. It's possible to mix jagged and multidimensional arrays. The final example, jaggedArray3, is a declaration and initialization of a single-dimensional jagged array that contains three two-dimensional array elements of different sizes.

```
int[][] jaggedArray = new int[3][];

jaggedArray[0] = [1, 3, 5, 7, 9];
jaggedArray[1] = [0, 2, 4, 6];
jaggedArray[2] = [11, 22];

int[][] jaggedArray2 =
[
    [1, 3, 5, 7, 9],
    [0, 2, 4, 6],
    [11, 22]
];

// Assign 77 to the second element ([1]) of the first array ([0]):
jaggedArray2[0][1] = 77;

// Assign 88 to the second element ([1]) of the third array ([2]):
jaggedArray2[2][1] = 88;

int[][,] jaggedArray3 =
[
    new int[,] { {1,3}, {5,7} },
    new int[,] { {0,2}, {4,6}, {8,10} },
    new int[,] { {11,22}, {99,88}, {0,9} }
];

Console.Write("{0}", jaggedArray3[0][1, 0]);
Console.WriteLine(jaggedArray3.Length);
```

A jagged array's elements must be initialized before you can use them. Each of the elements is itself an array. It's also possible to use initializers to fill the array elements with values. When you use initializers, you don't need the array size.

Implicitly typed arrays

You can create an implicitly typed array in which the type of the array instance is inferred from the elements specified in the array initializer. The rules for any implicitly typed variable also apply to implicitly typed arrays.

The following examples show how to create an implicitly typed array:

```
int[] a = new[] { 1, 10, 100, 1000 }; // int[]

// Accessing array
Console.WriteLine("First element: " + a[0]);
Console.WriteLine("Second element: " + a[1]);
Console.WriteLine("Third element: " + a[2]);
Console.WriteLine("Fourth element: " + a[3]);
```

```
/* Outputs
First element: 1
Second element: 10
Third element: 100
Fourth element: 1000
*/

var b = new[] { "hello", null, "world" }; // string[]

// Accessing elements of an array using 'string.Join' method
Console.WriteLine(string.Join(" ", b));
/* Output
hello  world
*/

// single-dimension jagged array
int[][] c =
[
    [1,2,3,4],
    [5,6,7,8]
];
// Looping through the outer array
for (int k = 0; k < c.Length; k++)
{
    // Looping through each inner array
    for (int j = 0; j < c[k].Length; j++)
    {
        // Accessing each element and printing it to the console
        Console.WriteLine($"Element at c[{k}][{j}] is: {c[k][j]}");
    }
}
/* Outputs
Element at c[0][0] is: 1
Element at c[0][1] is: 2
Element at c[0][2] is: 3
Element at c[0][3] is: 4
Element at c[1][0] is: 5
Element at c[1][1] is: 6
Element at c[1][2] is: 7
Element at c[1][3] is: 8
*/

// jagged array of strings
string[][] d =
[
    ["Luca", "Mads", "Luke", "Dinesh"],
    ["Karen", "Suma", "Frances"]
];

// Looping through the outer array
int i = 0;
foreach (var subArray in d)
{
    // Looping through each inner array
```

```
        int j = 0;
        foreach (var element in subArray)
        {
            // Accessing each element and printing it to the console
            Console.WriteLine($"Element at d[{i}][{j}] is: {element}");
            j++;
        }
        i++;
    }
    /* Outputs
    Element at d[0][0] is: Luca
    Element at d[0][1] is: Mads
    Element at d[0][2] is: Luke
    Element at d[0][3] is: Dinesh
    Element at d[1][0] is: Karen
    Element at d[1][1] is: Suma
    Element at d[1][2] is: Frances
    */
```

In the previous example, notice that with implicitly typed arrays, no square brackets are used on the left side of the initialization statement. Also, jagged arrays are initialized by using new [] just like single-dimensional arrays.

When you create an anonymous type that contains an array, the array must be implicitly typed in the type's object initializer. In the following example, contacts is an implicitly typed array of anonymous types, each of which contains an array named PhoneNumbers. The var keyword isn't used inside the object initializers.

```
var contacts = new[]
{
    new
    {
        Name = "Eugene Zabokritski",
        PhoneNumbers = new[] { "206-555-0108", "425-555-0001" }
    },
    new
    {
        Name = "Hanying Feng",
        PhoneNumbers = new[] { "650-555-0199" }
    }
};
```

Collections and Data Structures

All collections provide methods for adding, removing, or finding items in the collection. In addition, all collections that directly or indirectly implement the ICollection interface or the ICollection interface share these features:

- The ability to enumerate the collection

.NET collections either implement System.Collections.IEnumerable or System.Collections.Generic.IEnumerable to enable the collection to be iterated through. An enumerator can be thought of as a movable pointer to any element in the collection. The foreach, in statement and the For Each...Next Statement use the enumerator exposed by the GetEnumerator method and hide the complexity of manipulating the enumerator. In addition, any collection that implements System.Collections.Generic.IEnumerable is considered a queryable type and can be queried with LINQ. LINQ queries provide a common pattern for accessing data. They're typically more concise and readable than standard foreach loops and provide filtering, ordering, and grouping capabilities. LINQ queries can also improve performance. For more information, see LINQ to Objects (C#), LINQ to Objects (Visual Basic), Parallel LINQ (PLINQ), Introduction to LINQ Queries (C#), and Basic Query Operations (Visual Basic).

- The ability to copy the collection contents to an array

All collections can be copied to an array using the CopyTo method. However, the order of the elements in the new array is based on the sequence in which the enumerator returns them. The resulting array is always one-dimensional with a lower bound of zero.

In addition, many collection classes contain the following features:

- Capacity and Count properties

The capacity of a collection is the number of elements it can contain. The count of a collection is the number of elements it actually contains. Some collections hide the capacity or the count or both.

Most collections automatically expand in capacity when the current capacity is reached. The memory is reallocated, and the elements are copied from the old collection to the new one. This design reduces the code required to use the collection. However, the performance of the collection might be negatively affected. For example, for List, if Count is less than Capacity, adding an item is an O(1) operation. If the capacity needs to be increased to accommodate the new element, adding an item becomes an O(n) operation, where n is Count. The best way to avoid poor performance caused by multiple reallocations is to set the initial capacity to be the estimated size of the collection.

A BitArray is a special case; its capacity is the same as its length, which is the same as its count.

- A consistent lower bound

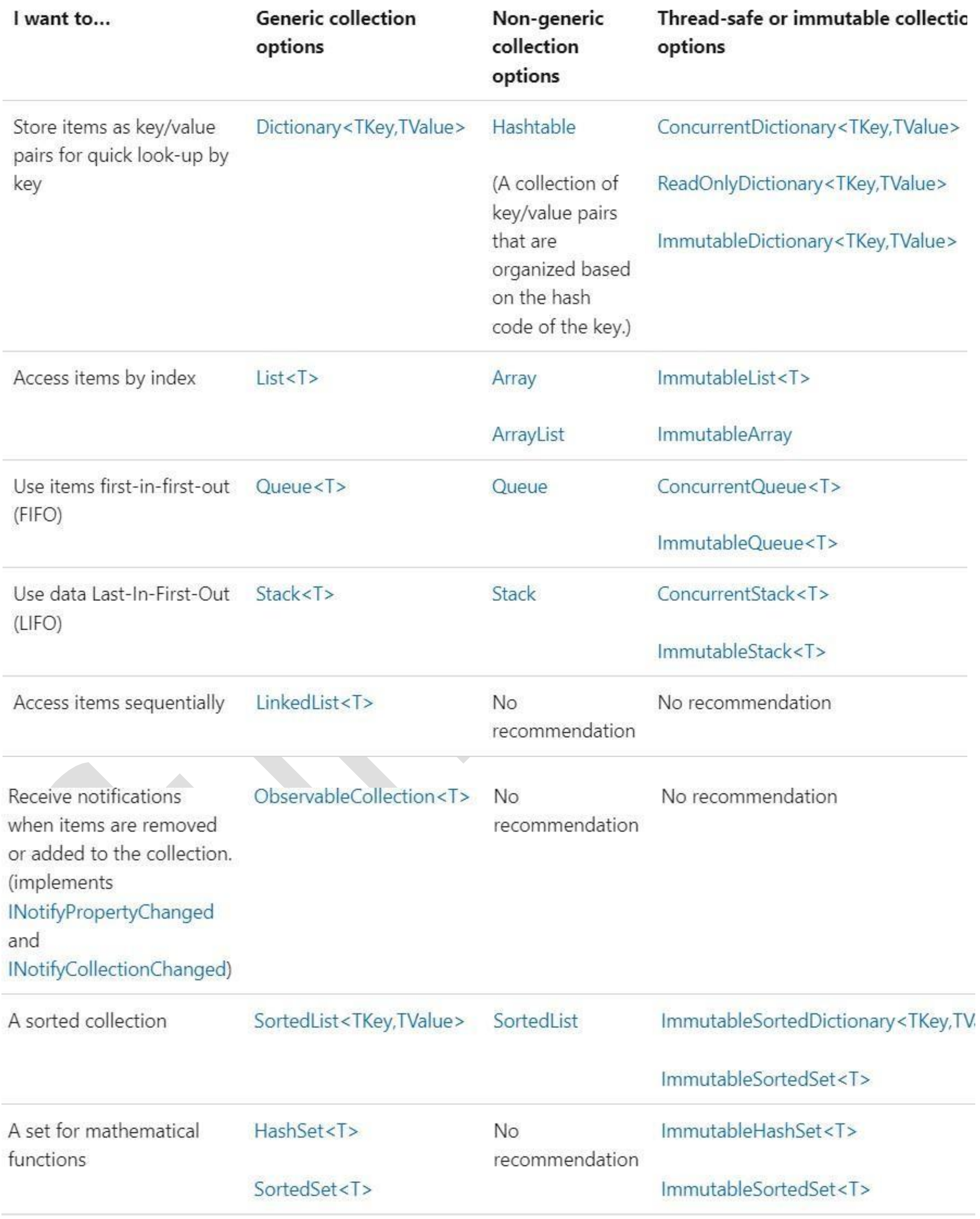
The lower bound of a collection is the index of its first element. All indexed collections in the System.Collections namespaces have a lower bound of zero, meaning they're 0-indexed. Array has a lower bound of zero by default, but a different lower bound can be defined when creating an instance of the Array class using Array.CreateInstance.

- Synchronization for access from multiple threads (System.Collections classes only).

Non-generic collection types in the System.Collections namespace provide some thread safety with synchronization; typically exposed through the SyncRoot and IsSynchronized members. These collections aren't thread-safe by default. If you require scalable and efficient multi-threaded access to a collection, use one of the classes in the System.Collections.Concurrent namespace or consider using an immutable collection. For more information, see Thread-Safe Collections.

Choose a collection

In general, you should use generic collections. The following table describes some common collection scenarios and the collection classes you can use for those scenarios. If you're new to generic collections, the following table will help you choose the generic collection that works best for your task:



I want to...	Generic collection options	Non-generic collection options	Thread-safe or immutable collectic options
Store items as key/value pairs for quick look-up by key	Dictionary<TKey,TValue>	Hashtable (A collection of key/value pairs that are organized based on the hash code of the key.)	ConcurrentDictionary<TKey,TValue> ReadOnlyDictionary<TKey,TValue> ImmutableDictionary<TKey,TValue>
Access items by index	List<T>	Array ArrayList	ImmutableList<T> ImmutableArray
Use items first-in-first-out (FIFO)	Queue<T>	Queue	ConcurrentQueue<T> ImmutableQueue<T>
Use data Last-In-First-Out (LIFO)	Stack<T>	Stack	ConcurrentStack<T> ImmutableStack<T>
Access items sequentially	LinkedList<T>	No recommendation	No recommendation
Receive notifications when items are removed or added to the collection. (implements INotifyPropertyChanged and INotifyCollectionChanged)	ObservableCollection<T>	No recommendation	No recommendation
A sorted collection	SortedList<TKey,TValue>	SortedList	ImmutableSortedDictionary<TKey,TV ImmutableSortedSet<T>
A set for mathematical functions	HashSet<T> SortedSet<T>	No recommendation	ImmutableHashSet<T> ImmutableSortedSet<T>

ArrayList

Definition

- Namespace: System.Collections
- Assembly: System.Runtime.dll

Implements the IList interface using an array whose size is dynamically increased as required.

```
public class ArrayList : ICloneable, System.Collections.IList
```

Derived System.Windows.Forms.DomainUpDown.DomainUpDownItemCollection Implements ICollection IEnumerable IList ICloneable

Examples

The following example shows how to create and initialize an ArrayList and how to display its values.

```
using System;
using System.Collections;
public class SamplesArrayList  {

   public static void Main()  {

      // Creates and initializes a new ArrayList.
      ArrayList myAL = new ArrayList();
      myAL.Add("Hello");
      myAL.Add("World");
      myAL.Add("!");

      // Displays the properties and values of the ArrayList.
      Console.WriteLine( "myAL" );
      Console.WriteLine( "    Count:    {0}", myAL.Count );
      Console.WriteLine( "    Capacity: {0}", myAL.Capacity );
      Console.Write( "    Values:" );
      PrintValues( myAL );
   }

   public static void PrintValues( IEnumerable myList )  {
      foreach ( Object obj in myList )
         Console.Write( "   {0}", obj );
      Console.WriteLine();
   }
}


/*
This code produces output similar to the following:

myAL
    Count:    3
    Capacity: 4
```

```
     Values:    Hello    World    !

 */
```

Remarks

The ArrayList is not guaranteed to be sorted. You must sort the ArrayList by calling its Sort method prior to performing operations (such as BinarySearch) that require the ArrayList to be sorted. To maintain a collection that is automatically sorted as new elements are added, you can use the SortedSet class.
The capacity of an ArrayList is the number of elements the ArrayList can hold. As elements are added to an ArrayList, the capacity is automatically increased as required through reallocation. The capacity can be decreased by calling TrimToSize or by setting the Capacity property explicitly.

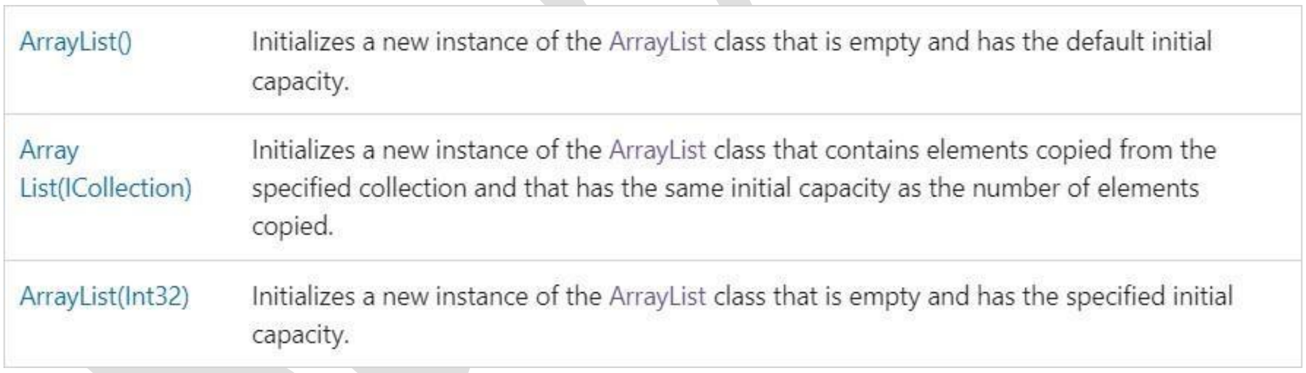
.NET Framework only: For very large ArrayList objects, you can increase the maximum capacity to 2 billion elements on a 64-bit system by setting the enabled attribute of the configuration element to true in the run-time environment.

Elements in this collection can be accessed using an integer index. Indexes in this collection are zero-based.

The ArrayList collection accepts null as a valid value. It also allows duplicate elements.

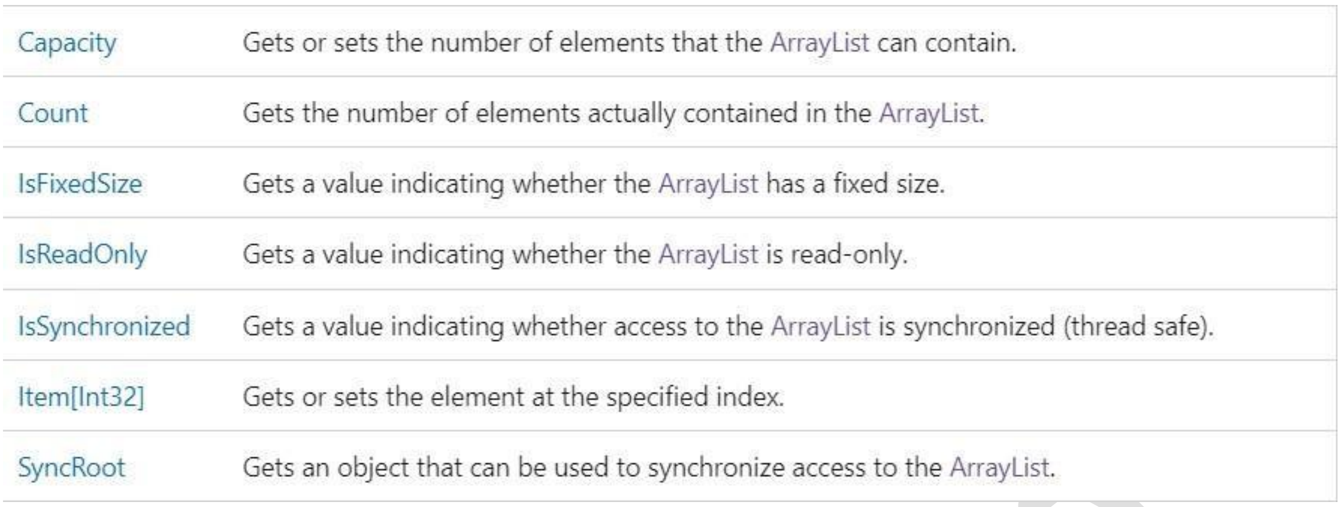
Using multidimensional arrays as elements in an ArrayList collection is not supported.

Constructors



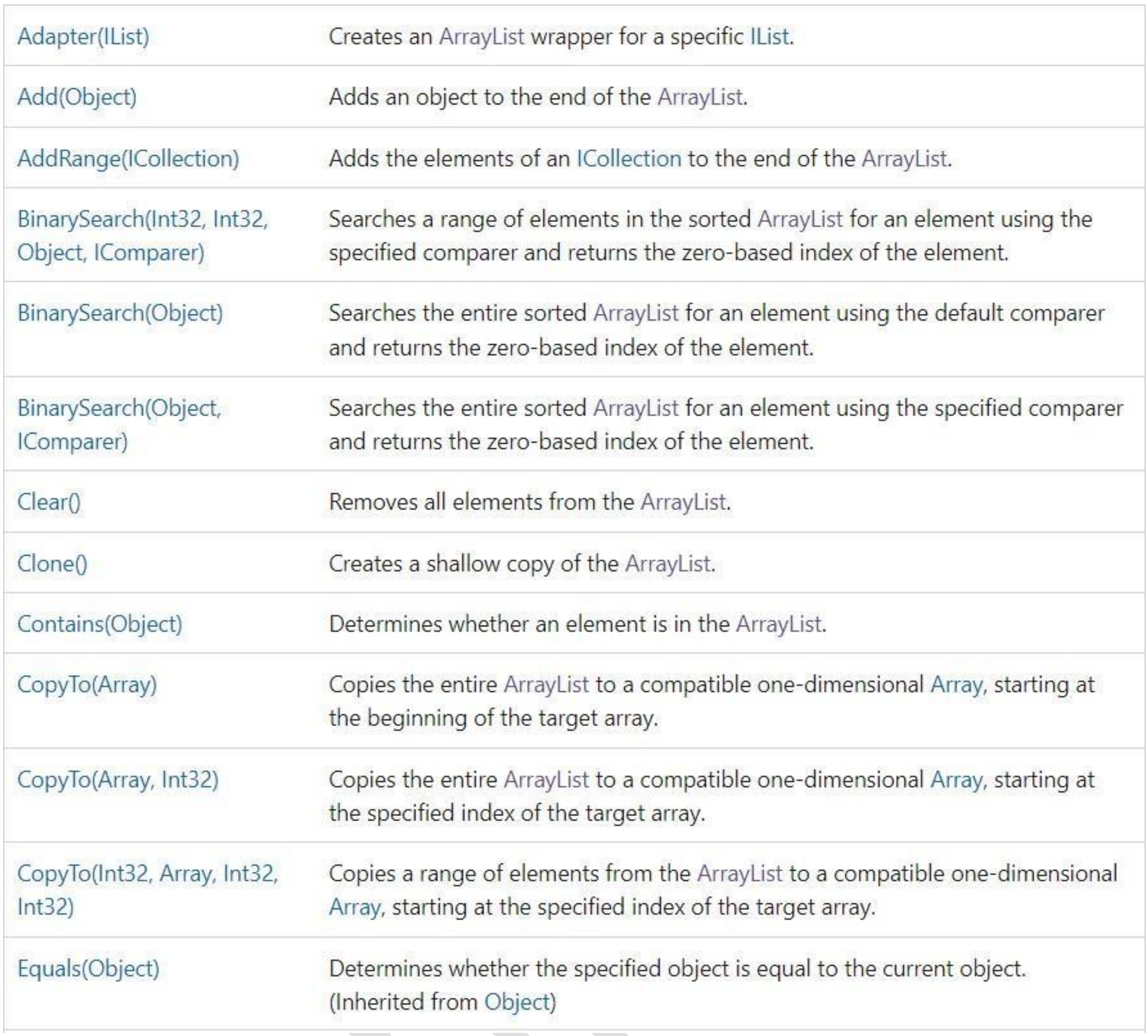
ArrayList()	Initializes a new instance of the ArrayList class that is empty and has the default initial capacity.
Array List(ICollection)	Initializes a new instance of the ArrayList class that contains elements copied from the specified collection and that has the same initial capacity as the number of elements copied.
ArrayList(Int32)	Initializes a new instance of the ArrayList class that is empty and has the specified initial capacity.

Properties

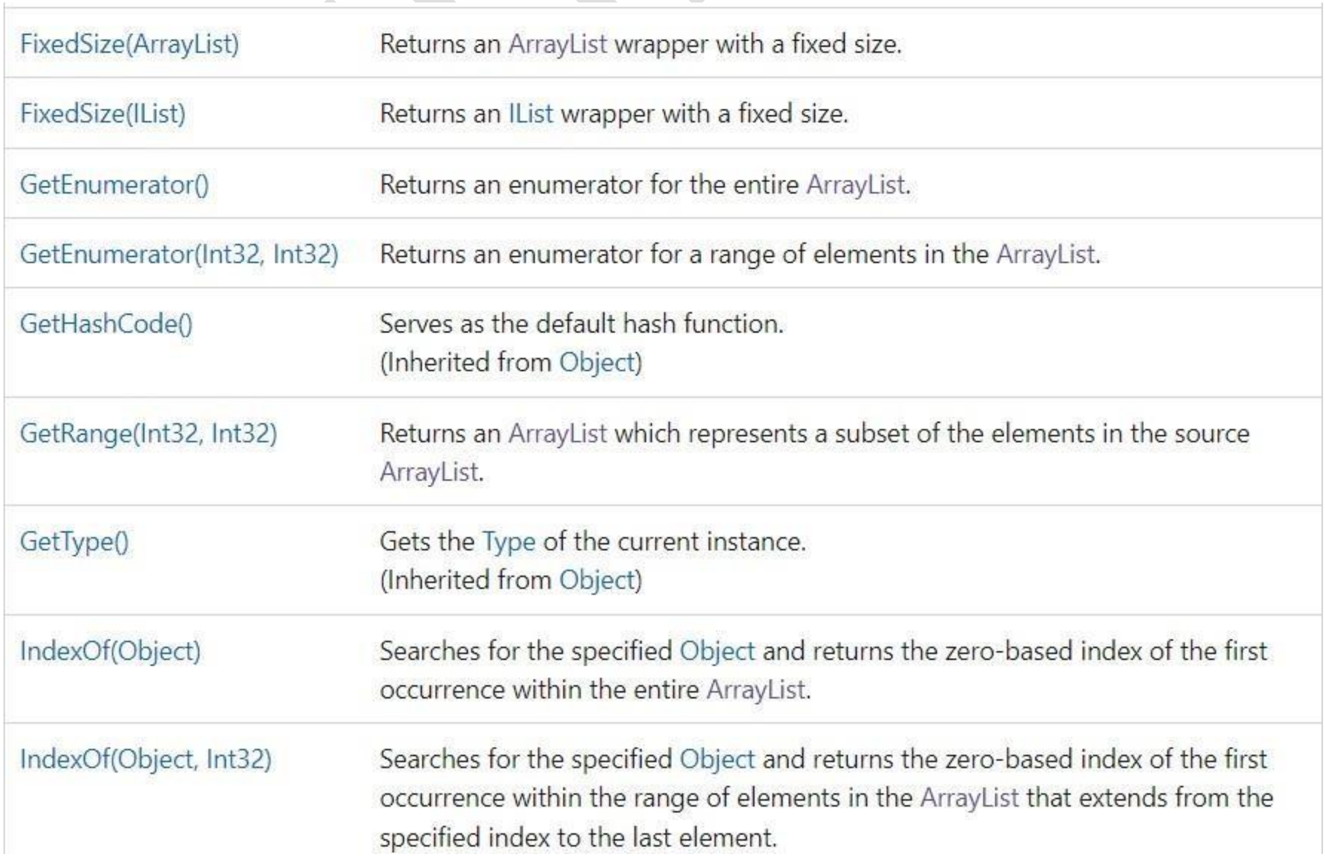


Capacity	Gets or sets the number of elements that the ArrayList can contain.
Count	Gets the number of elements actually contained in the ArrayList.
IsFixedSize	Gets a value indicating whether the ArrayList has a fixed size.
IsReadOnly	Gets a value indicating whether the ArrayList is read-only.
IsSynchronized	Gets a value indicating whether access to the ArrayList is synchronized (thread safe).
Item[Int32]	Gets or sets the element at the specified index.
SyncRoot	Gets an object that can be used to synchronize access to the ArrayList.

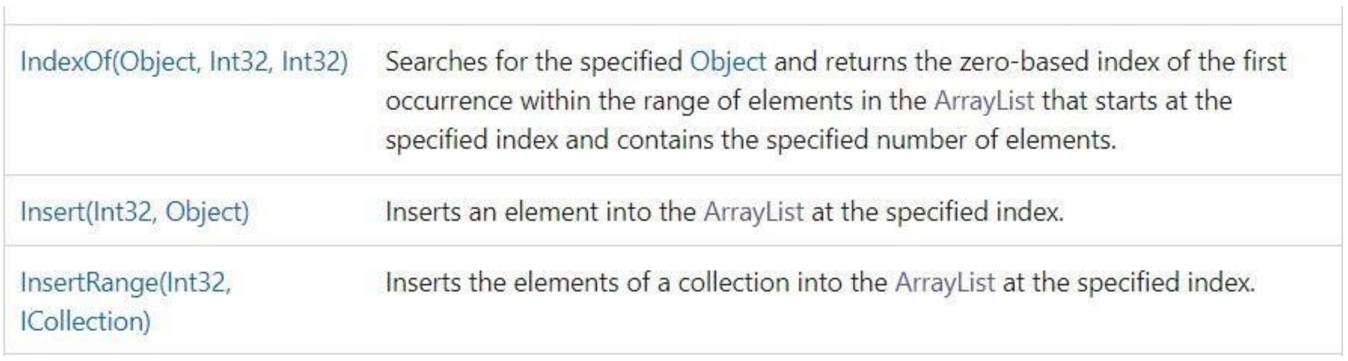
Methods



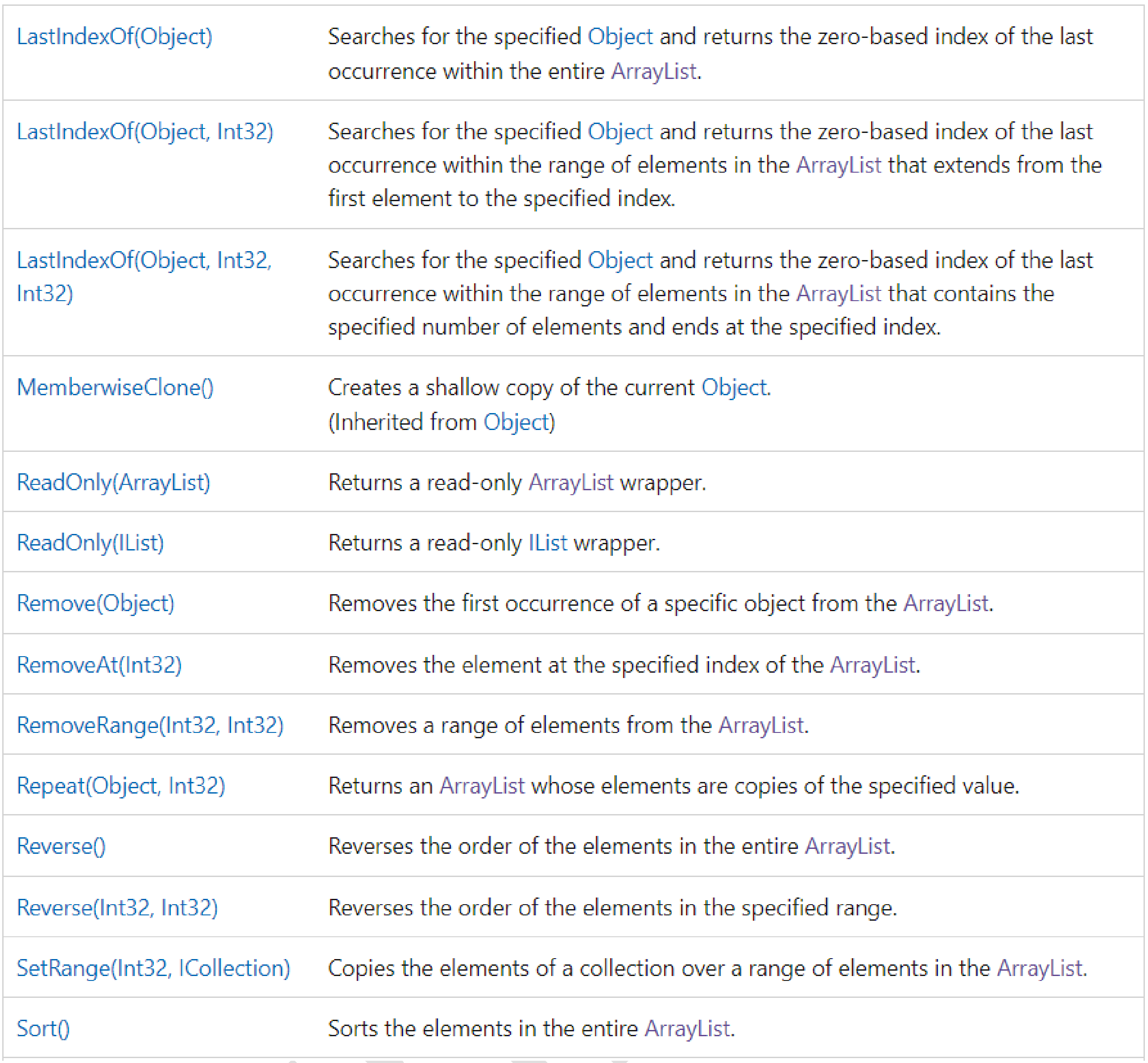
Adapter(IList)	Creates an ArrayList wrapper for a specific IList.
Add(Object)	Adds an object to the end of the ArrayList.
AddRange(ICollection)	Adds the elements of an ICollection to the end of the ArrayList.
BinarySearch(Int32, Int32, Object, IComparer)	Searches a range of elements in the sorted ArrayList for an element using the specified comparer and returns the zero-based index of the element.
BinarySearch(Object)	Searches the entire sorted ArrayList for an element using the default comparer and returns the zero-based index of the element.
BinarySearch(Object, IComparer)	Searches the entire sorted ArrayList for an element using the specified comparer and returns the zero-based index of the element.
Clear()	Removes all elements from the ArrayList.
Clone()	Creates a shallow copy of the ArrayList.
Contains(Object)	Determines whether an element is in the ArrayList.
CopyTo(Array)	Copies the entire ArrayList to a compatible one-dimensional Array, starting at the beginning of the target array.
CopyTo(Array, Int32)	Copies the entire ArrayList to a compatible one-dimensional Array, starting at the specified index of the target array.
CopyTo(Int32, Array, Int32, Int32)	Copies a range of elements from the ArrayList to a compatible one-dimensional Array, starting at the specified index of the target array.
Equals(Object)	Determines whether the specified object is equal to the current object. (Inherited from Object)



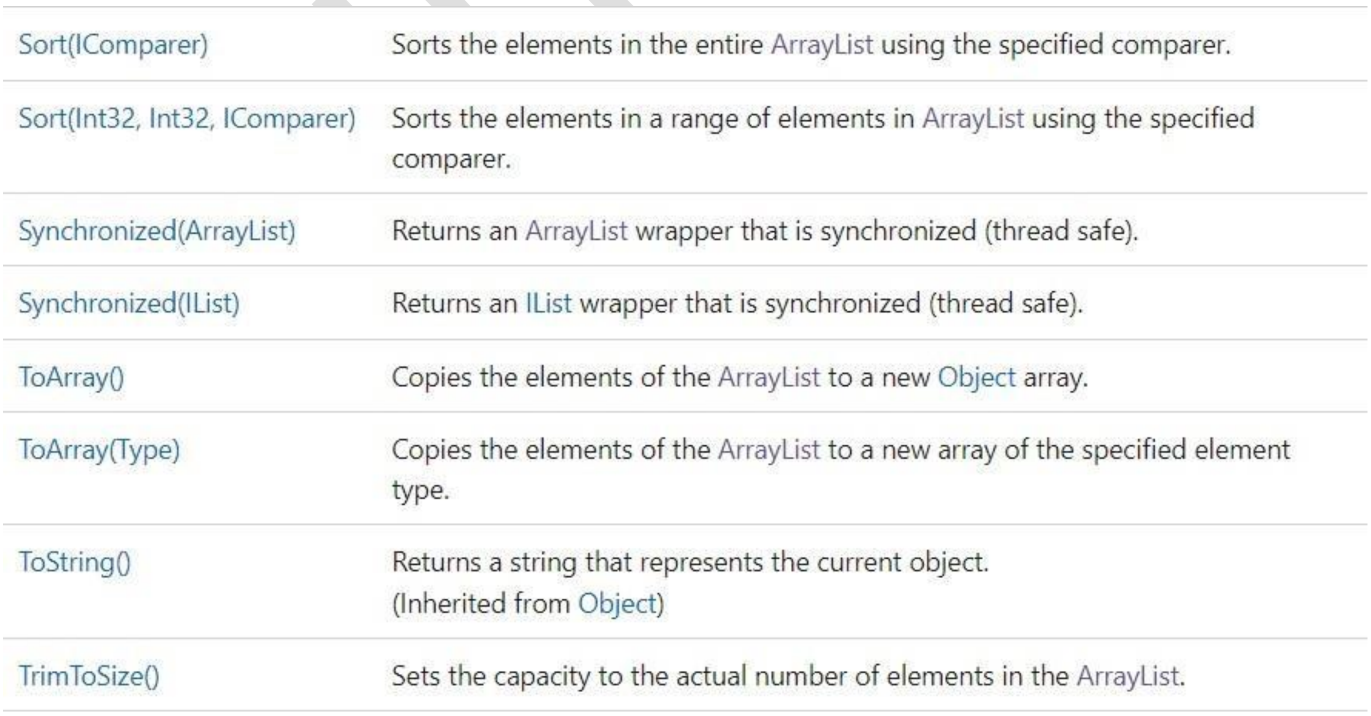
FixedSize(ArrayList)	Returns an ArrayList wrapper with a fixed size.
FixedSize(IList)	Returns an IList wrapper with a fixed size.
GetEnumerator()	Returns an enumerator for the entire ArrayList.
GetEnumerator(Int32, Int32)	Returns an enumerator for a range of elements in the ArrayList.
GetHashCode()	Serves as the default hash function. (Inherited from Object)
GetRange(Int32, Int32)	Returns an ArrayList which represents a subset of the elements in the source ArrayList.
GetType()	Gets the Type of the current instance. (Inherited from Object)
IndexOf(Object)	Searches for the specified Object and returns the zero-based index of the first occurrence within the entire ArrayList.
IndexOf(Object, Int32)	Searches for the specified Object and returns the zero-based index of the first occurrence within the range of elements in the ArrayList that extends from the specified index to the last element.



IndexOf(Object, Int32, Int32)	Searches for the specified Object and returns the zero-based index of the first occurrence within the range of elements in the ArrayList that starts at the specified index and contains the specified number of elements.
Insert(Int32, Object)	Inserts an element into the ArrayList at the specified index.
InsertRange(Int32, ICollection)	Inserts the elements of a collection into the ArrayList at the specified index.



LastIndexOf(Object)	Searches for the specified Object and returns the zero-based index of the last occurrence within the entire ArrayList.
LastIndexOf(Object, Int32)	Searches for the specified Object and returns the zero-based index of the last occurrence within the range of elements in the ArrayList that extends from the first element to the specified index.
LastIndexOf(Object, Int32, Int32)	Searches for the specified Object and returns the zero-based index of the last occurrence within the range of elements in the ArrayList that contains the specified number of elements and ends at the specified index.
MemberwiseClone()	Creates a shallow copy of the current Object. (Inherited from Object)
ReadOnly(ArrayList)	Returns a read-only ArrayList wrapper.
ReadOnly(IList)	Returns a read-only IList wrapper.
Remove(Object)	Removes the first occurrence of a specific object from the ArrayList.
RemoveAt(Int32)	Removes the element at the specified index of the ArrayList.
RemoveRange(Int32, Int32)	Removes a range of elements from the ArrayList.
Repeat(Object, Int32)	Returns an ArrayList whose elements are copies of the specified value.
Reverse()	Reverses the order of the elements in the entire ArrayList.
Reverse(Int32, Int32)	Reverses the order of the elements in the specified range.
SetRange(Int32, ICollection)	Copies the elements of a collection over a range of elements in the ArrayList.
Sort()	Sorts the elements in the entire ArrayList.



Sort(IComparer)	Sorts the elements in the entire ArrayList using the specified comparer.
Sort(Int32, Int32, IComparer)	Sorts the elements in a range of elements in ArrayList using the specified comparer.
Synchronized(ArrayList)	Returns an ArrayList wrapper that is synchronized (thread safe).
Synchronized(IList)	Returns an IList wrapper that is synchronized (thread safe).
ToArray()	Copies the elements of the ArrayList to a new Object array.
ToArray(Type)	Copies the elements of the ArrayList to a new array of the specified element type.
ToString()	Returns a string that represents the current object. (Inherited from Object)
TrimToSize()	Sets the capacity to the actual number of elements in the ArrayList.

Hashtable

- Namespace: System.Collections
- Assembly: System.Runtime.dll

Represents a collection of key/value pairs that are organized based on the hash code of the key.

```
public class Hashtable : ICloneable, System.Collections.IDictionary,
System.Runtime.Serialization.IDeserializationCallback,
System.Runtime.Serialization.ISerializable
```

- Derived : System.Configuration.SettingsAttributeDictionary System.Configuration.SettingsContext System.Data.PropertyCollection System.Printing.IndexedProperties.PrintPropertyDictionary

- Implements: ICollection IDictionary IEnumerable ICloneable IDeserializationCallback ISerializable

The following example shows how to create, initialize and perform various functions to a Hashtable and how to print out its keys and values.

```
using System;
using System.Collections;

class Example
{
    public static void Main()
    {
        // Create a new hash table.
        //
        Hashtable openWith = new Hashtable();

        // Add some elements to the hash table. There are no
        // duplicate keys, but some of the values are duplicates.
        openWith.Add("txt", "notepad.exe");
        openWith.Add("bmp", "paint.exe");
        openWith.Add("dib", "paint.exe");
        openWith.Add("rtf", "wordpad.exe");

        // The Add method throws an exception if the new key is
        // already in the hash table.
        try
        {
            openWith.Add("txt", "winword.exe");
        }
        catch
        {
            Console.WriteLine("An element with Key = \"txt\" already exists.");
        }

        // The Item property is the default property, so you
        // can omit its name when accessing elements.
        Console.WriteLine("For key = \"rtf\", value = {0}.", openWith["rtf"]);
        // The default Item property can be used to change the value
        // associated with a key.
        openWith["rtf"] = "winword.exe";
        Console.WriteLine("For key = \"rtf\", value = {0}.", openWith["rtf"]);

        // If a key does not exist, setting the default Item property
        // for that key adds a new key/value pair.
        openWith["doc"] = "winword.exe";

        // ContainsKey can be used to test keys before inserting
        // them.
        if (!openWith.ContainsKey("ht"))
        {
            openWith.Add("ht", "hypertrm.exe");
            Console.WriteLine("Value added for key = \"ht\": {0}",
openWith["ht"]);
        }

        // When you use foreach to enumerate hash table elements,
        // the elements are retrieved as KeyValuePair objects.
        Console.WriteLine();
        foreach( DictionaryEntry de in openWith )
        {
            Console.WriteLine("Key = {0}, Value = {1}", de.Key, de.Value);
        }

        // To get the values alone, use the Values property.
        ICollection valueColl = openWith.Values;

        // The elements of the ValueCollection are strongly typed
        // with the type that was specified for hash table values.
        Console.WriteLine();
        foreach( string s in valueColl )
        {
            Console.WriteLine("Value = {0}", s);
        }

        // To get the keys alone, use the Keys property.
        ICollection keyColl = openWith.Keys;

        // The elements of the KeyCollection are strongly typed
        // with the type that was specified for hash table keys.
        Console.WriteLine();
        foreach( string s in keyColl )
        {
            Console.WriteLine("Key = {0}", s);
        }

        // Use the Remove method to remove a key/value pair.
        Console.WriteLine("\nRemove(\"doc\")");
        openWith.Remove("doc");

        if (!openWith.ContainsKey("doc"))
        {
            Console.WriteLine("Key \"doc\" is not found.");
        }
    }
}

/* This code example produces the following output:

An element with Key = "txt" already exists.
For key = "rtf", value = wordpad.exe.
For key = "rtf", value = winword.exe.
Value added for key = "ht": hypertrm.exe

Key = dib, Value = paint.exe
Key = txt, Value = notepad.exe
Key = ht, Value = hypertrm.exe
Key = bmp, Value = paint.exe
Key = rtf, Value = winword.exe
Key = doc, Value = winword.exe

Value = paint.exe
Value = notepad.exe
Value = hypertrm.exe
Value = paint.exe
Value = winword.exe
Value = winword.exe

Key = dib
Key = txt
Key = ht
Key = bmp
Key = rtf
Key = doc

Remove("doc")
Key "doc" is not found.
 */
```

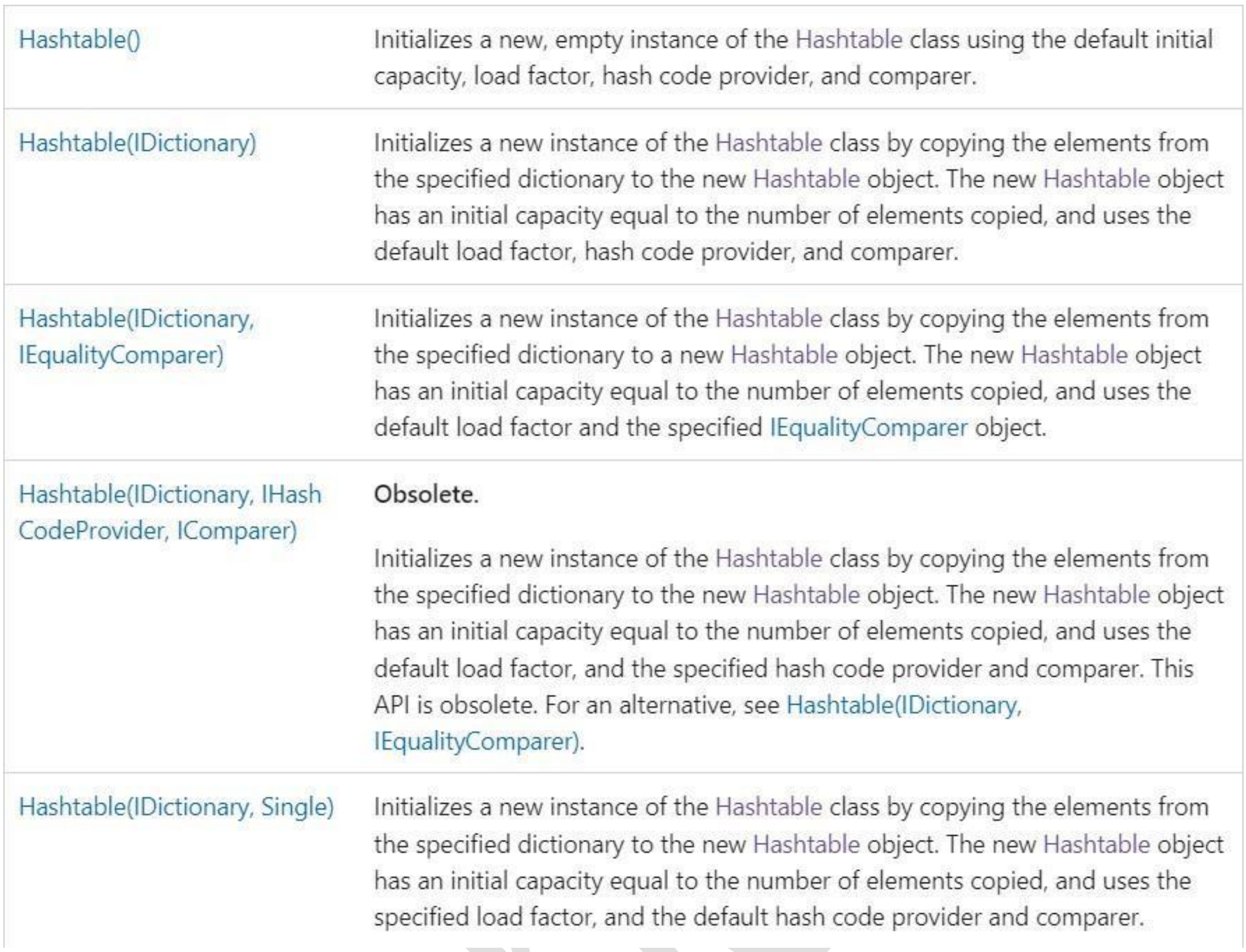
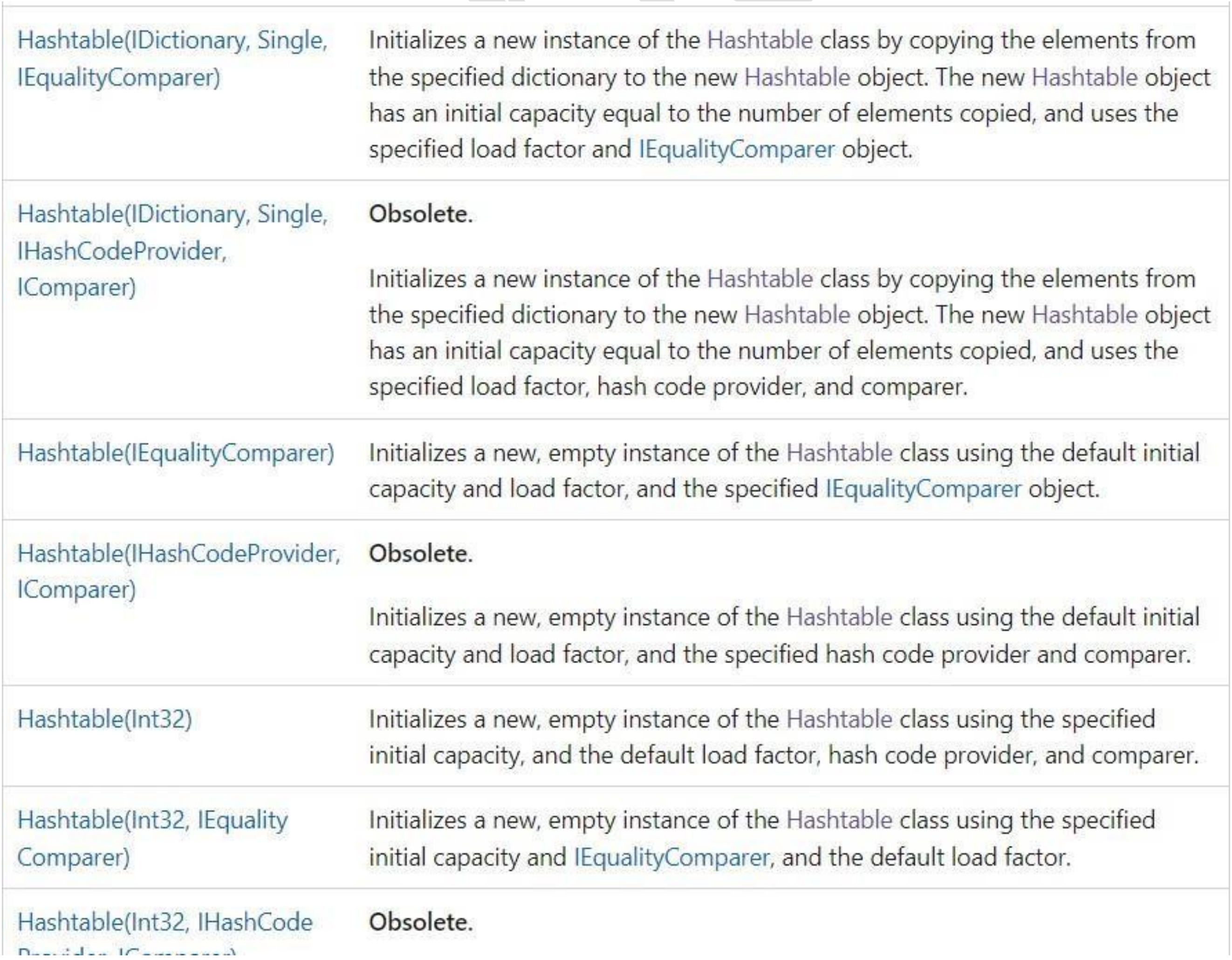
- Each element is a key/value pair stored in a DictionaryEntry object. A key cannot be null, but a value can be.
- The objects used as keys by a Hashtable are required to override the Object.GetHashCode method (or the IHashCodeProvider interface) and the Object.Equals method (or the IComparer interface). The implementation of both methods and interfaces must handle case sensitivity the same way; otherwise, the Hashtable might behave incorrectly. For example, when creating a Hashtable, you must use the CaseInsensitiveHashCodeProvider class (or any case-insensitive IHashCodeProvider implementation) with the CaseInsensitiveComparer class (or any case-insensitive IComparer implementation).
- Furthermore, these methods must produce the same results when called with the same parameters while the key exists in the Hashtable. An alternative is to use a Hashtable constructor with an IEqualityComparer parameter. If key equality were simply reference equality, the inherited implementation of Object.GetHashCode and Object.Equals would suffice.

- Key objects must be immutable as long as they are used as keys in the Hashtable.
- When an element is added to the Hashtable, the element is placed into a bucket based on the hash code of the key. Subsequent lookups of the key use the hash code of the key to search in only one particular bucket, thus substantially reducing the number of key comparisons required to find an element.
- The load factor of a Hashtable determines the maximum ratio of elements to buckets. Smaller load factors cause faster average lookup times at the cost of increased memory consumption. The default load factor of 1.0 generally provides the best balance between speed and size. A different load factor can also be specified when the Hashtable is created.
- As elements are added to a Hashtable, the actual load factor of the Hashtable increases. When the actual load factor reaches the specified load factor, the number of buckets in the Hashtable is automatically increased to the smallest prime number that is larger than twice the current number of Hashtable buckets.
- Each key object in the Hashtable must provide its own hash function, which can be accessed by calling GetHash. However, any object implementing IHashCodeProvider can be passed to a Hashtable constructor, and that hash function is used for all objects in the table.
- The capacity of a Hashtable is the number of elements the Hashtable can hold. As elements are added to a Hashtable, the capacity is automatically increased as required through reallocation.
- .NET Framework only: For very large Hashtable objects, you can increase the maximum capacity to 2 billion elements on a 64-bit system by setting the enabled attribute of the configuration element to true in the run-time environment.
- The foreach statement of the C# language (For Each in Visual Basic) returns an object of the type of the elements in the collection. Since each element of the Hashtable is a key/value pair, the element type is not the type of the key or the type of the value. Instead, the element type is DictionaryEntry.

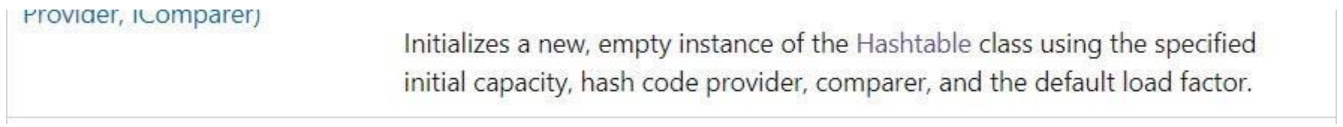
```
foreach(DictionaryEntry de in myHashtable)
{
    // ...
}
```

- The foreach statement is a wrapper around the enumerator, which only allows reading from, not writing to, the collection.
- Because serializing and deserializing an enumerator for a Hashtable can cause the elements to become reordered, it is not possible to continue enumeration without calling the Reset method.

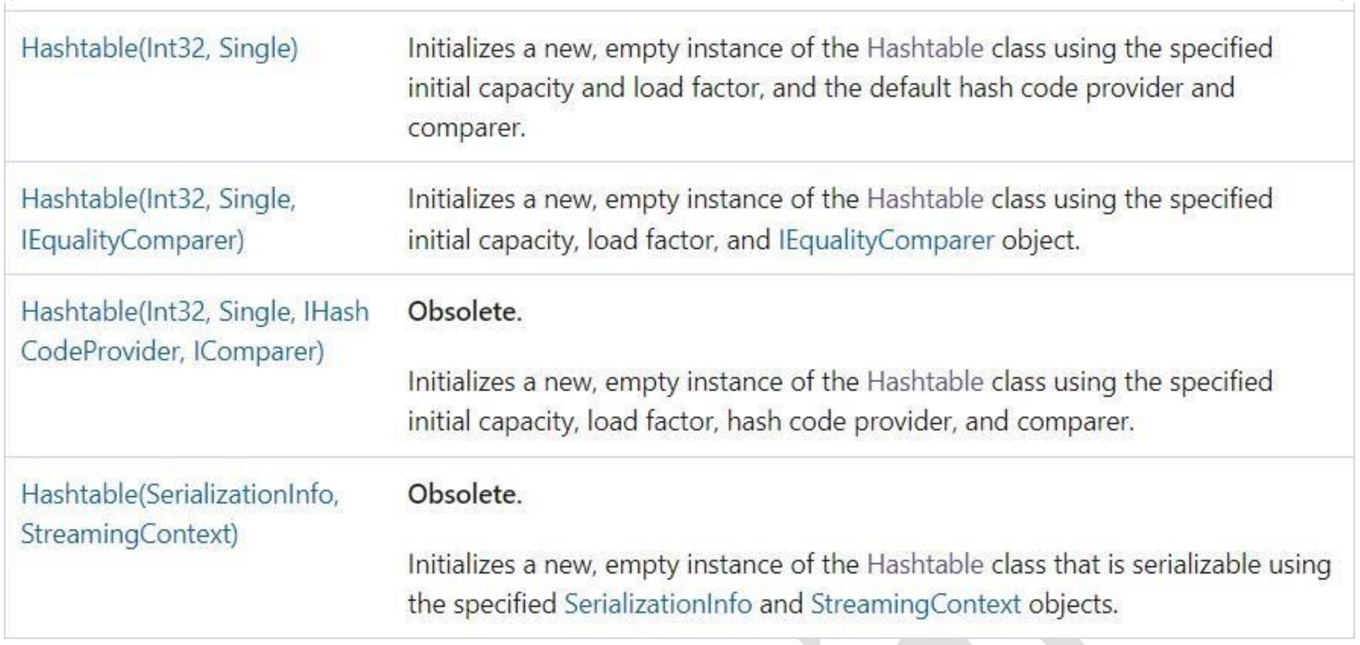
Constructors

Hashtable()	Initializes a new, empty instance of the Hashtable class using the default initial capacity, load factor, hash code provider, and comparer.
Hashtable(IDictionary)	Initializes a new instance of the Hashtable class by copying the elements from the specified dictionary to the new Hashtable object. The new Hashtable object has an initial capacity equal to the number of elements copied, and uses the default load factor, hash code provider, and comparer.
Hashtable(IDictionary, IEqualityComparer)	Initializes a new instance of the Hashtable class by copying the elements from the specified dictionary to a new Hashtable object. The new Hashtable object has an initial capacity equal to the number of elements copied, and uses the default load factor and the specified IEqualityComparer object.
Hashtable(IDictionary, IHash CodeProvider, IComparer)	**Obsolete.** Initializes a new instance of the Hashtable class by copying the elements from the specified dictionary to the new Hashtable object. The new Hashtable object has an initial capacity equal to the number of elements copied, and uses the default load factor, and the specified hash code provider and comparer. This API is obsolete. For an alternative, see Hashtable(IDictionary, IEqualityComparer).
Hashtable(IDictionary, Single)	Initializes a new instance of the Hashtable class by copying the elements from the specified dictionary to the new Hashtable object. The new Hashtable object has an initial capacity equal to the number of elements copied, and uses the specified load factor, and the default hash code provider and comparer.
Hashtable(IDictionary, Single, IEqualityComparer)	Initializes a new instance of the Hashtable class by copying the elements from the specified dictionary to the new Hashtable object. The new Hashtable object has an initial capacity equal to the number of elements copied, and uses the specified load factor and IEqualityComparer object.
Hashtable(IDictionary, Single, IHashCodeProvider, IComparer)	**Obsolete.** Initializes a new instance of the Hashtable class by copying the elements from the specified dictionary to the new Hashtable object. The new Hashtable object has an initial capacity equal to the number of elements copied, and uses the specified load factor, hash code provider, and comparer.
Hashtable(IEqualityComparer)	Initializes a new, empty instance of the Hashtable class using the default initial capacity and load factor, and the specified IEqualityComparer object.
Hashtable(IHashCodeProvider, IComparer)	**Obsolete.** Initializes a new, empty instance of the Hashtable class using the default initial capacity and load factor, and the specified hash code provider and comparer.
Hashtable(Int32)	Initializes a new, empty instance of the Hashtable class using the specified initial capacity, and the default load factor, hash code provider, and comparer.
Hashtable(Int32, IEquality Comparer)	Initializes a new, empty instance of the Hashtable class using the specified initial capacity and IEqualityComparer, and the default load factor.
Hashtable(Int32, IHashCode	**Obsolete.**

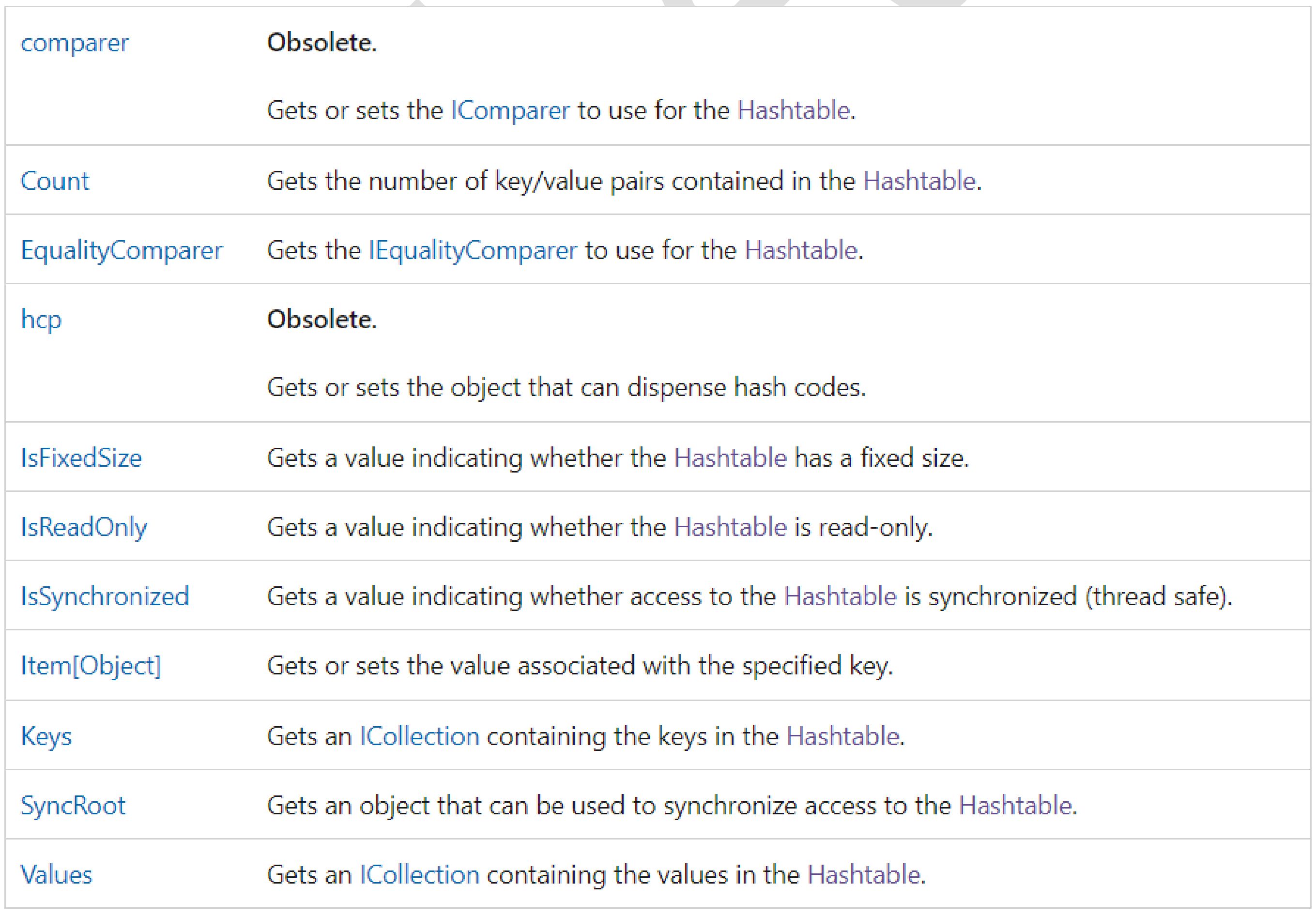


Provider, IComparer)	Initializes a new, empty instance of the Hashtable class using the specified initial capacity, hash code provider, comparer, and the default load factor.



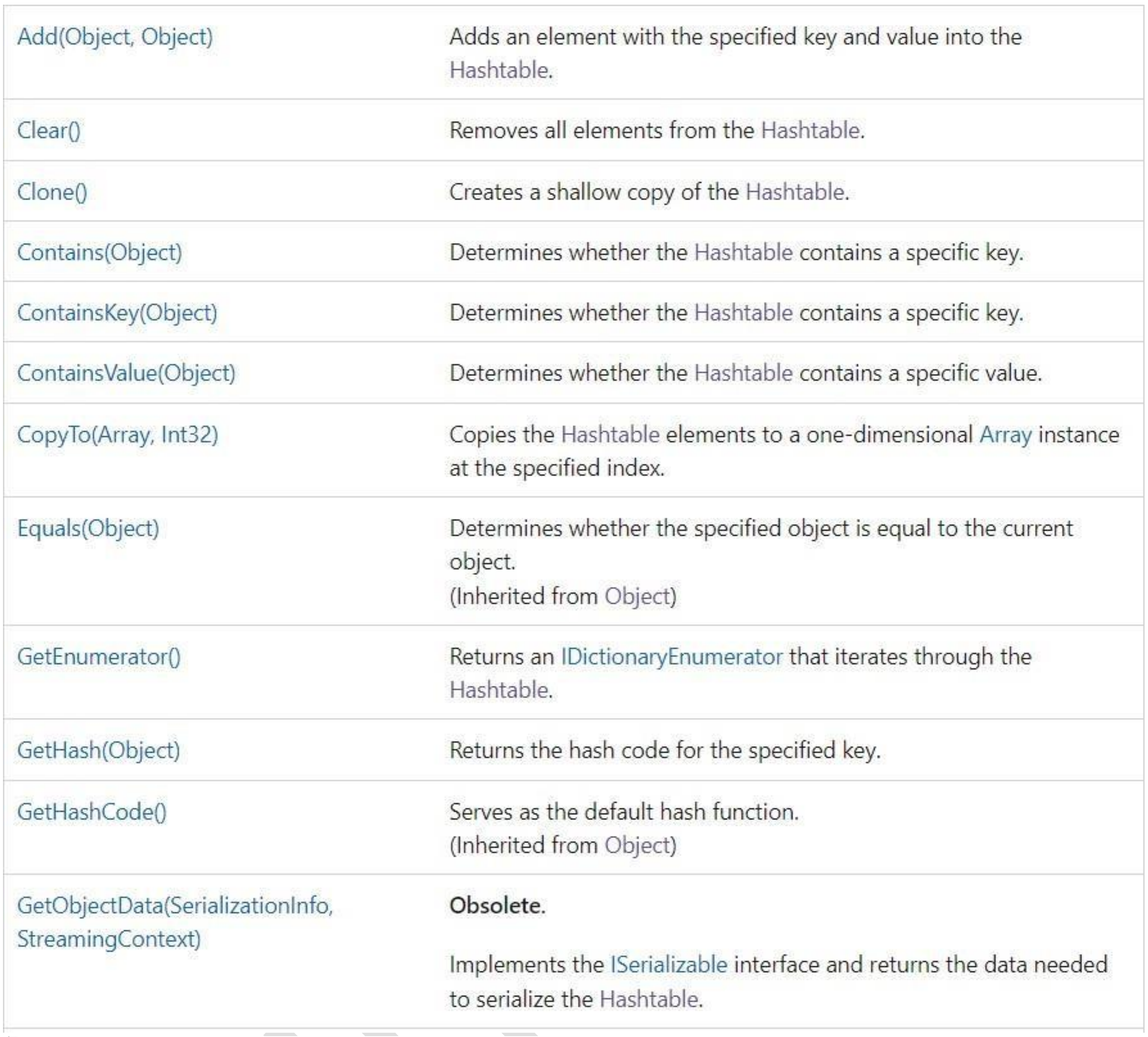
Hashtable(Int32, Single)	Initializes a new, empty instance of the Hashtable class using the specified initial capacity and load factor, and the default hash code provider and comparer.
Hashtable(Int32, Single, IEqualityComparer)	Initializes a new, empty instance of the Hashtable class using the specified initial capacity, load factor, and IEqualityComparer object.
Hashtable(Int32, Single, IHashCodeProvider, IComparer)	**Obsolete.** Initializes a new, empty instance of the Hashtable class using the specified initial capacity, load factor, hash code provider, and comparer.
Hashtable(SerializationInfo, StreamingContext)	**Obsolete.** Initializes a new, empty instance of the Hashtable class that is serializable using the specified SerializationInfo and StreamingContext objects.

Properties

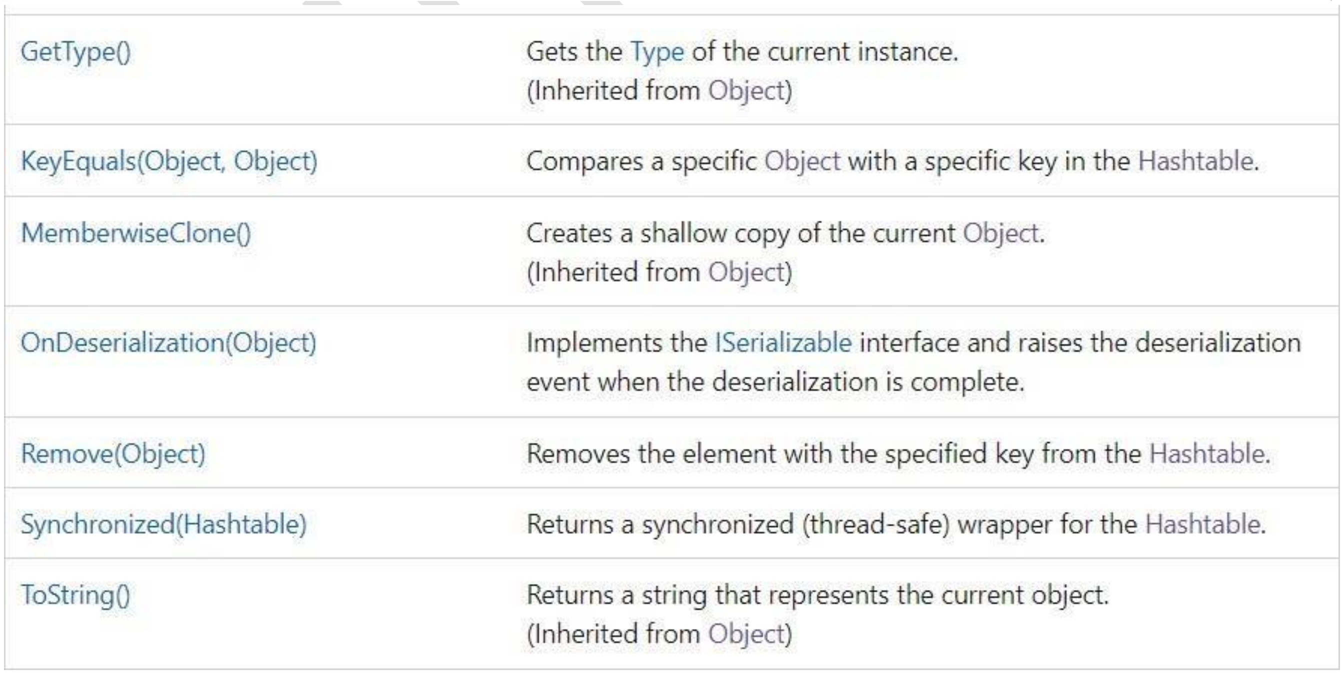


comparer	**Obsolete.** Gets or sets the IComparer to use for the Hashtable.
Count	Gets the number of key/value pairs contained in the Hashtable.
EqualityComparer	Gets the IEqualityComparer to use for the Hashtable.
hcp	**Obsolete.** Gets or sets the object that can dispense hash codes.
IsFixedSize	Gets a value indicating whether the Hashtable has a fixed size.
IsReadOnly	Gets a value indicating whether the Hashtable is read-only.
IsSynchronized	Gets a value indicating whether access to the Hashtable is synchronized (thread safe).
Item[Object]	Gets or sets the value associated with the specified key.
Keys	Gets an ICollection containing the keys in the Hashtable.
SyncRoot	Gets an object that can be used to synchronize access to the Hashtable.
Values	Gets an ICollection containing the values in the Hashtable.

Methods



Add(Object, Object)	Adds an element with the specified key and value into the Hashtable.
Clear()	Removes all elements from the Hashtable.
Clone()	Creates a shallow copy of the Hashtable.
Contains(Object)	Determines whether the Hashtable contains a specific key.
ContainsKey(Object)	Determines whether the Hashtable contains a specific key.
ContainsValue(Object)	Determines whether the Hashtable contains a specific value.
CopyTo(Array, Int32)	Copies the Hashtable elements to a one-dimensional Array instance at the specified index.
Equals(Object)	Determines whether the specified object is equal to the current object. (Inherited from Object)
GetEnumerator()	Returns an IDictionaryEnumerator that iterates through the Hashtable.
GetHash(Object)	Returns the hash code for the specified key.
GetHashCode()	Serves as the default hash function. (Inherited from Object)
GetObjectData(SerializationInfo, StreamingContext)	**Obsolete.** Implements the ISerializable interface and returns the data needed to serialize the Hashtable.



GetType()	Gets the Type of the current instance. (Inherited from Object)
KeyEquals(Object, Object)	Compares a specific Object with a specific key in the Hashtable.
MemberwiseClone()	Creates a shallow copy of the current Object. (Inherited from Object)
OnDeserialization(Object)	Implements the ISerializable interface and raises the deserialization event when the deserialization is complete.
Remove(Object)	Removes the element with the specified key from the Hashtable.
Synchronized(Hashtable)	Returns a synchronized (thread-safe) wrapper for the Hashtable.
ToString()	Returns a string that represents the current object. (Inherited from Object)

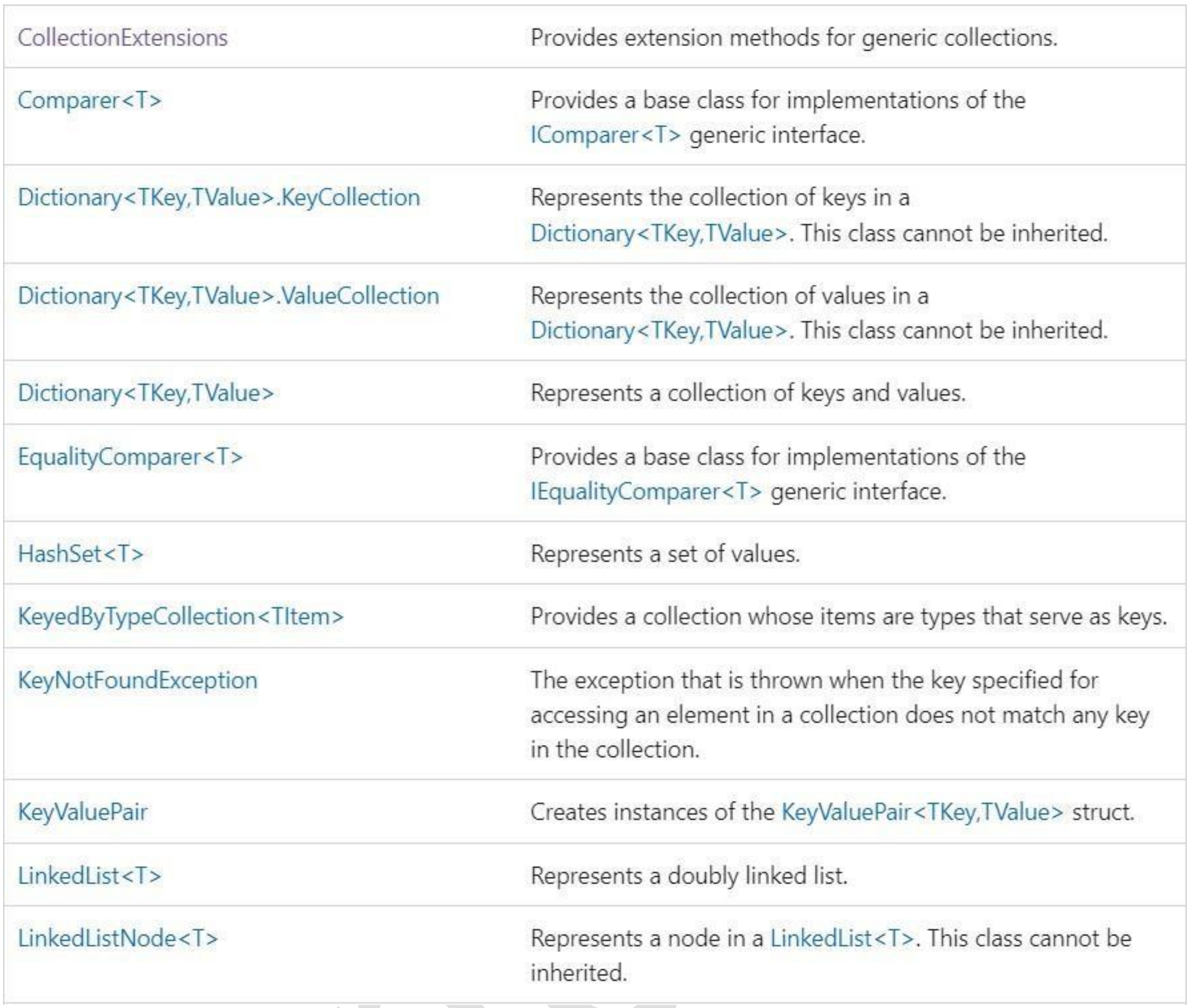
Thread Safety

Hashtable is thread safe for use by multiple reader threads and a single writing thread. It is thread safe for multi-thread use when only one of the threads perform write (update) operations, which allows for lock-free reads provided that the writers are serialized to the Hashtable. To support multiple writers all operations on the Hashtable must be done through the wrapper returned by the Synchronized(Hashtable) method, provided that there are no threads reading the Hashtable object.

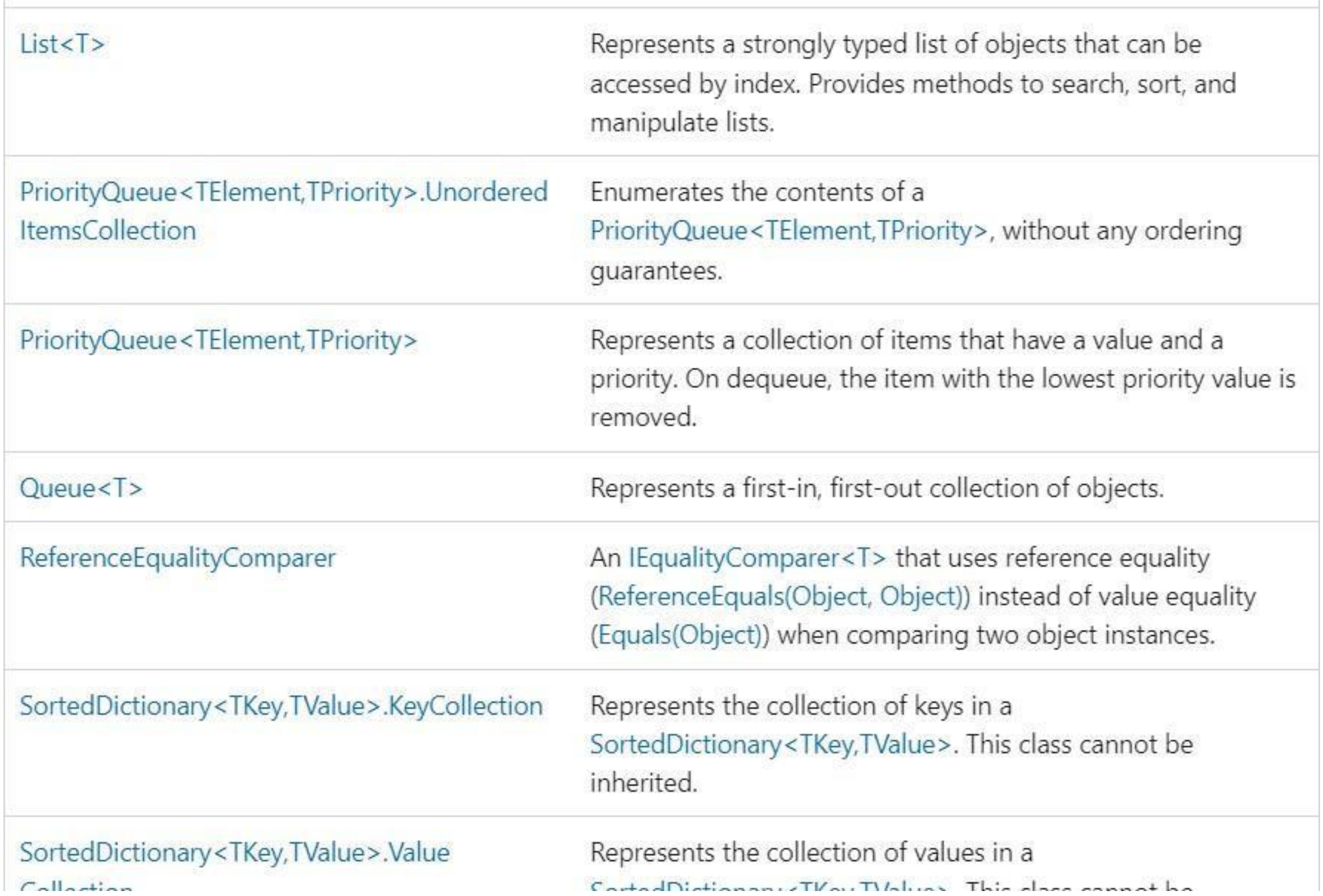
Generic Collections

- Namespace: System.Collections.Generic
- Contains interfaces and classes that define generic collections, which allow users to create strongly typed collections that provide better type safety and performance than non-generic strongly typed collections.

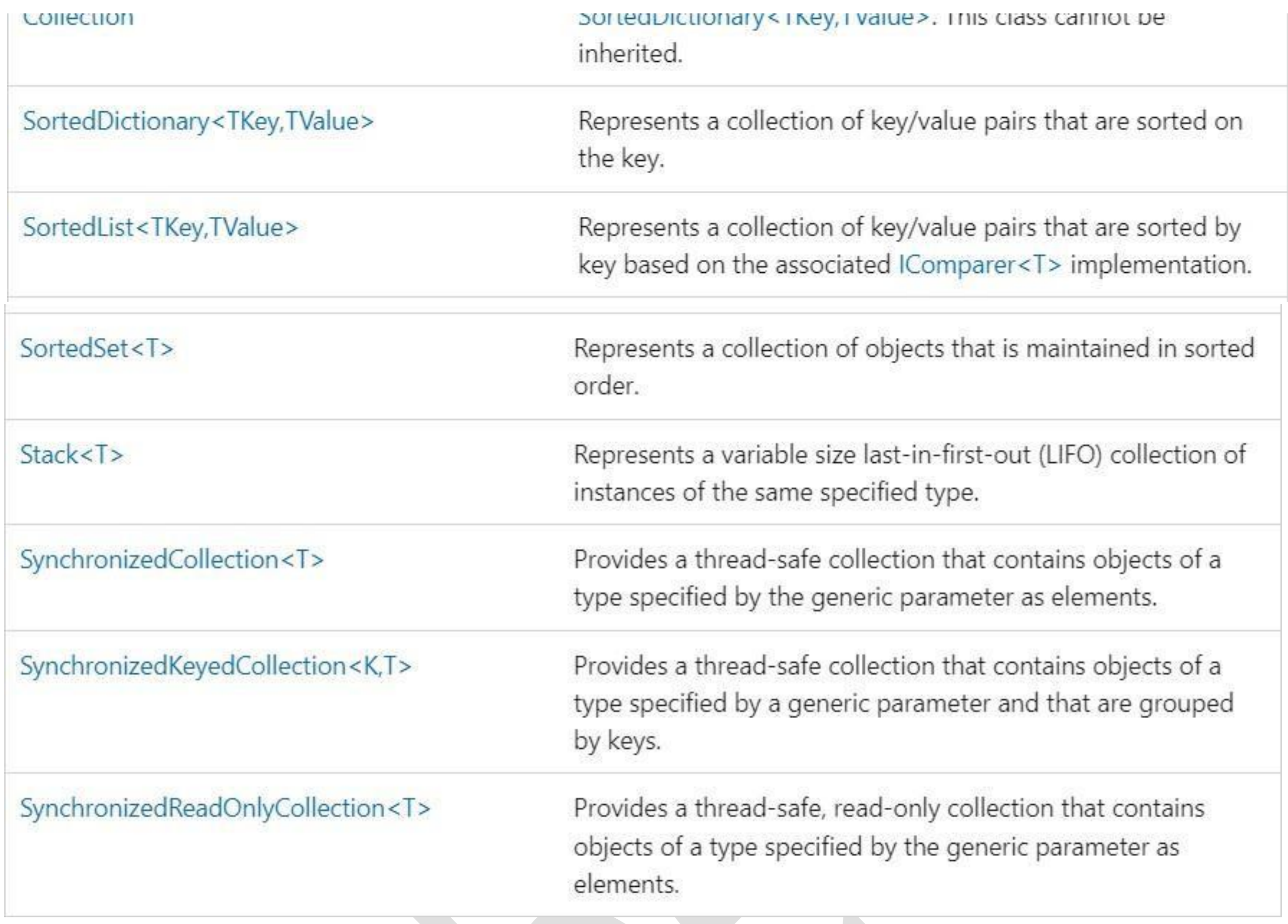
Classes



CollectionExtensions	Provides extension methods for generic collections.
Comparer<T>	Provides a base class for implementations of the IComparer<T> generic interface.
Dictionary<TKey,TValue>.KeyCollection	Represents the collection of keys in a Dictionary<TKey,TValue>. This class cannot be inherited.
Dictionary<TKey,TValue>.ValueCollection	Represents the collection of values in a Dictionary<TKey,TValue>. This class cannot be inherited.
Dictionary<TKey,TValue>	Represents a collection of keys and values.
EqualityComparer<T>	Provides a base class for implementations of the IEqualityComparer<T> generic interface.
HashSet<T>	Represents a set of values.
KeyedByTypeCollection<TItem>	Provides a collection whose items are types that serve as keys.
KeyNotFoundException	The exception that is thrown when the key specified for accessing an element in a collection does not match any key in the collection.
KeyValuePair	Creates instances of the KeyValuePair<TKey,TValue> struct.
LinkedList<T>	Represents a doubly linked list.
LinkedListNode<T>	Represents a node in a LinkedList<T>. This class cannot be inherited.

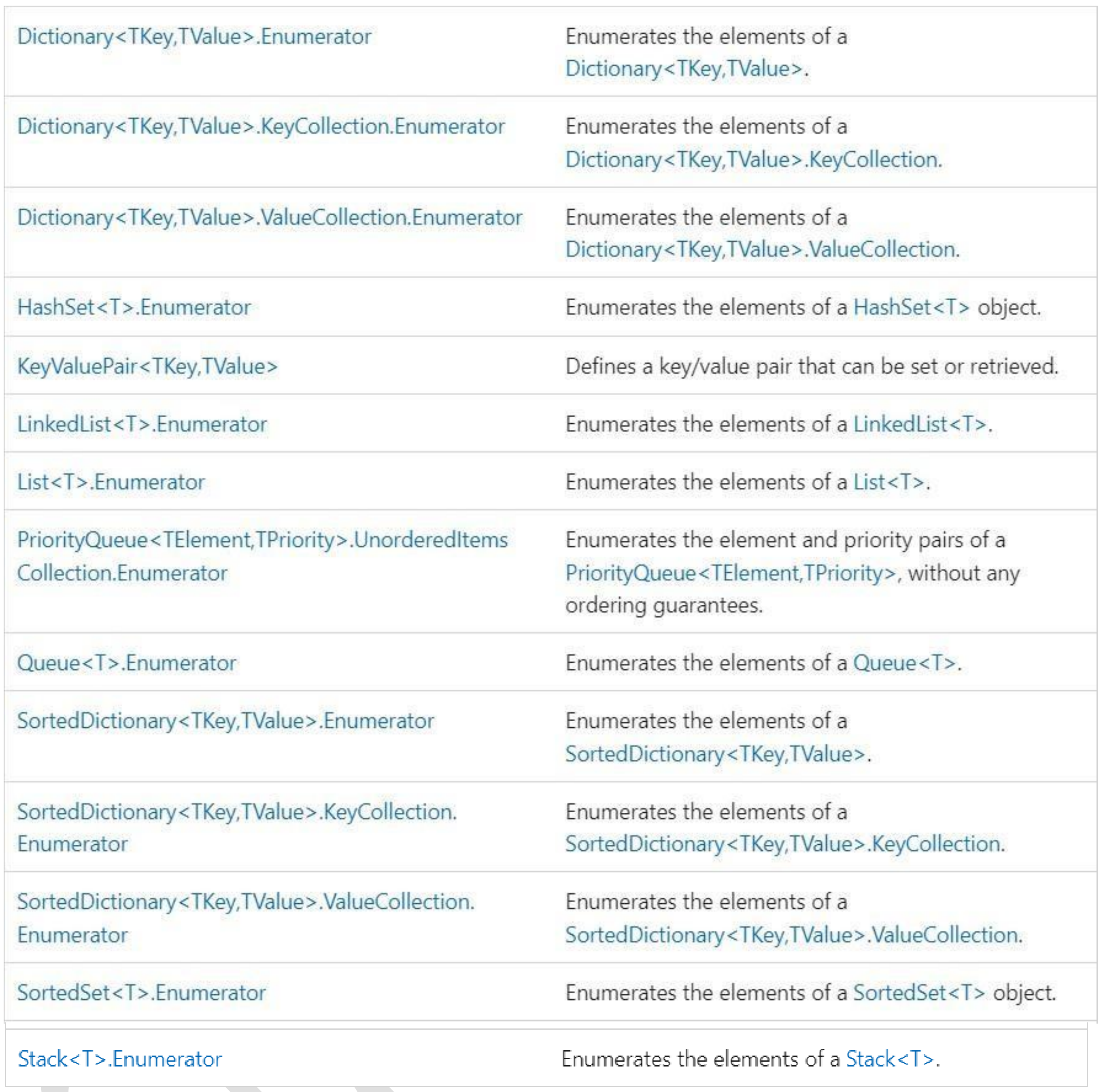


List<T>	Represents a strongly typed list of objects that can be accessed by index. Provides methods to search, sort, and manipulate lists.
PriorityQueue<TElement,TPriority>.UnorderedItemsCollection	Enumerates the contents of a PriorityQueue<TElement,TPriority>, without any ordering guarantees.
PriorityQueue<TElement,TPriority>	Represents a collection of items that have a value and a priority. On dequeue, the item with the lowest priority value is removed.
Queue<T>	Represents a first-in, first-out collection of objects.
ReferenceEqualityComparer	An IEqualityComparer<T> that uses reference equality (ReferenceEquals(Object, Object)) instead of value equality (Equals(Object)) when comparing two object instances.
SortedDictionary<TKey,TValue>.KeyCollection	Represents the collection of keys in a SortedDictionary<TKey,TValue>. This class cannot be inherited.
SortedDictionary<TKey,TValue>.Value Collection	Represents the collection of values in a SortedDictionary<TKey,TValue>. This class cannot be



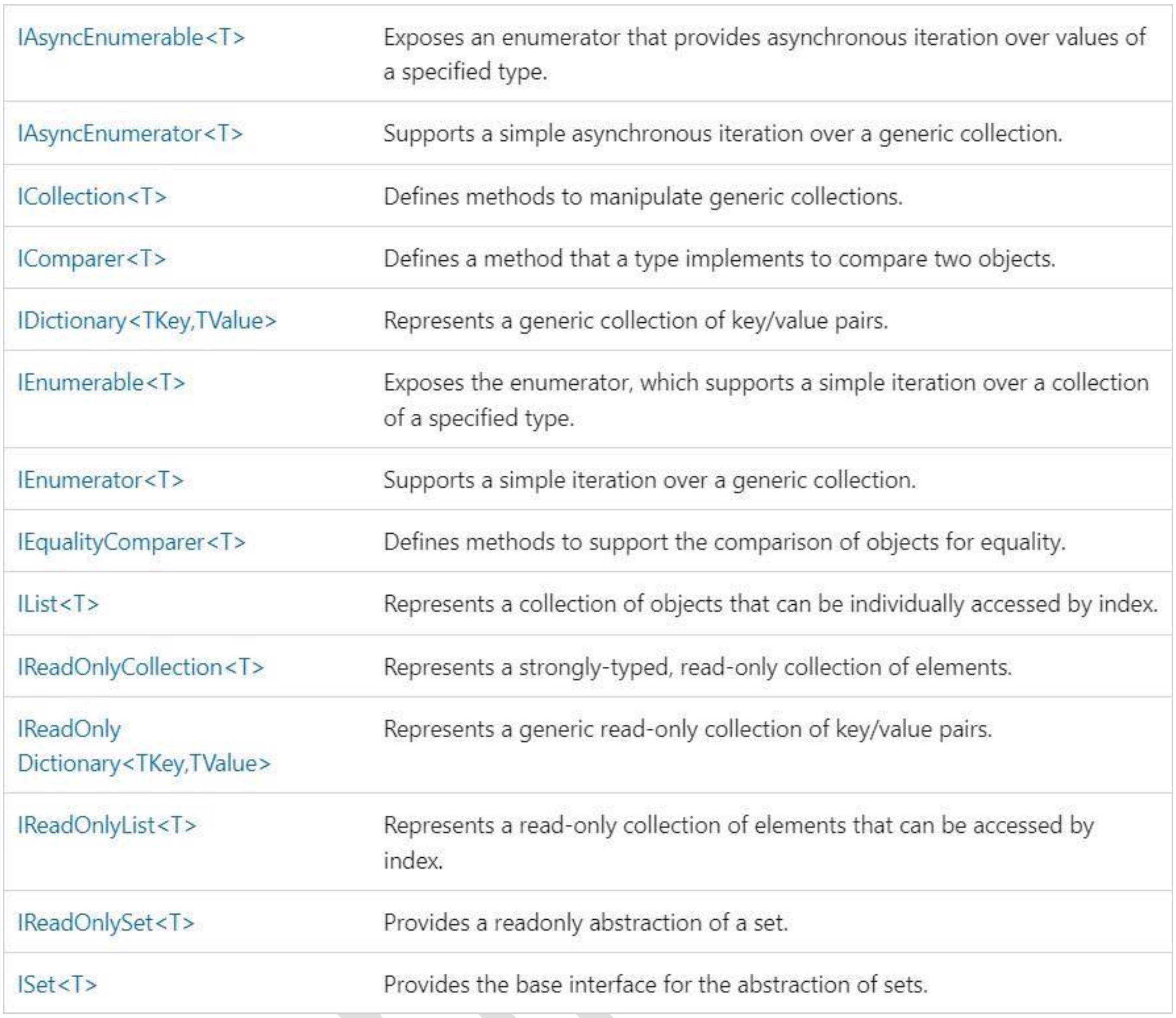
Collection	SortedDictionary<TKey,TValue>. This class cannot be inherited.
SortedDictionary<TKey,TValue>	Represents a collection of key/value pairs that are sorted on the key.
SortedList<TKey,TValue>	Represents a collection of key/value pairs that are sorted by key based on the associated IComparer<T> implementation.
SortedSet<T>	Represents a collection of objects that is maintained in sorted order.
Stack<T>	Represents a variable size last-in-first-out (LIFO) collection of instances of the same specified type.
SynchronizedCollection<T>	Provides a thread-safe collection that contains objects of a type specified by the generic parameter as elements.
SynchronizedKeyedCollection<K,T>	Provides a thread-safe collection that contains objects of a type specified by a generic parameter and that are grouped by keys.
SynchronizedReadOnlyCollection<T>	Provides a thread-safe, read-only collection that contains objects of a type specified by the generic parameter as elements.

Structs



Dictionary<TKey,TValue>.Enumerator	Enumerates the elements of a Dictionary<TKey,TValue>.
Dictionary<TKey,TValue>.KeyCollection.Enumerator	Enumerates the elements of a Dictionary<TKey,TValue>.KeyCollection.
Dictionary<TKey,TValue>.ValueCollection.Enumerator	Enumerates the elements of a Dictionary<TKey,TValue>.ValueCollection.
HashSet<T>.Enumerator	Enumerates the elements of a HashSet<T> object.
KeyValuePair<TKey,TValue>	Defines a key/value pair that can be set or retrieved.
LinkedList<T>.Enumerator	Enumerates the elements of a LinkedList<T>.
List<T>.Enumerator	Enumerates the elements of a List<T>.
PriorityQueue<TElement,TPriority>.UnorderedItems Collection.Enumerator	Enumerates the element and priority pairs of a PriorityQueue<TElement,TPriority>, without any ordering guarantees.
Queue<T>.Enumerator	Enumerates the elements of a Queue<T>.
SortedDictionary<TKey,TValue>.Enumerator	Enumerates the elements of a SortedDictionary<TKey,TValue>.
SortedDictionary<TKey,TValue>.KeyCollection. Enumerator	Enumerates the elements of a SortedDictionary<TKey,TValue>.KeyCollection.
SortedDictionary<TKey,TValue>.ValueCollection. Enumerator	Enumerates the elements of a SortedDictionary<TKey,TValue>.ValueCollection.
SortedSet<T>.Enumerator	Enumerates the elements of a SortedSet<T> object.
Stack<T>.Enumerator	Enumerates the elements of a Stack<T>.

Interfaces



IAsyncEnumerable<T>	Exposes an enumerator that provides asynchronous iteration over values of a specified type.
IAsyncEnumerator<T>	Supports a simple asynchronous iteration over a generic collection.
ICollection<T>	Defines methods to manipulate generic collections.
IComparer<T>	Defines a method that a type implements to compare two objects.
IDictionary<TKey,TValue>	Represents a generic collection of key/value pairs.
IEnumerable<T>	Exposes the enumerator, which supports a simple iteration over a collection of a specified type.
IEnumerator<T>	Supports a simple iteration over a generic collection.
IEqualityComparer<T>	Defines methods to support the comparison of objects for equality.
IList<T>	Represents a collection of objects that can be individually accessed by index.
IReadOnlyCollection<T>	Represents a strongly-typed, read-only collection of elements.
IReadOnly Dictionary<TKey,TValue>	Represents a generic read-only collection of key/value pairs.
IReadOnlyList<T>	Represents a read-only collection of elements that can be accessed by index.
IReadOnlySet<T>	Provides a readonly abstraction of a set.
ISet<T>	Provides the base interface for the abstraction of sets.

- Many of the generic collection types are direct analogs of nongeneric types. Dictionary<TKey,TValue> is a generic version of Hashtable; it uses the generic structure KeyValuePair<TKey,TValue> for enumeration instead of DictionaryEntry. List is a generic version of ArrayList. There are generic Queue and Stack classes that correspond to the nongeneric versions. There are generic and nongeneric versions of SortedList<TKey,TValue>. Both versions are hybrids of a dictionary and a list. The SortedDictionary<TKey,TValue> generic class is a pure dictionary and has no nongeneric counterpart. The LinkedList generic class is a true linked list and has no nongeneric counterpart.

List Class

- Namespace: System.Collections.Generic
- Assembly: System.Collections.dll

Represents a strongly typed list of objects that can be accessed by index. Provides methods to search, sort, and manipulate lists.

```
public class List<T> : System.Collections.Generic.ICollection<T>,
System.Collections.Generic.IEnumerable<T>, System.Collections.Generic.IList<T>,
```

Type Parameters

T :The type of elements in the list

Inheritance : Object -> List

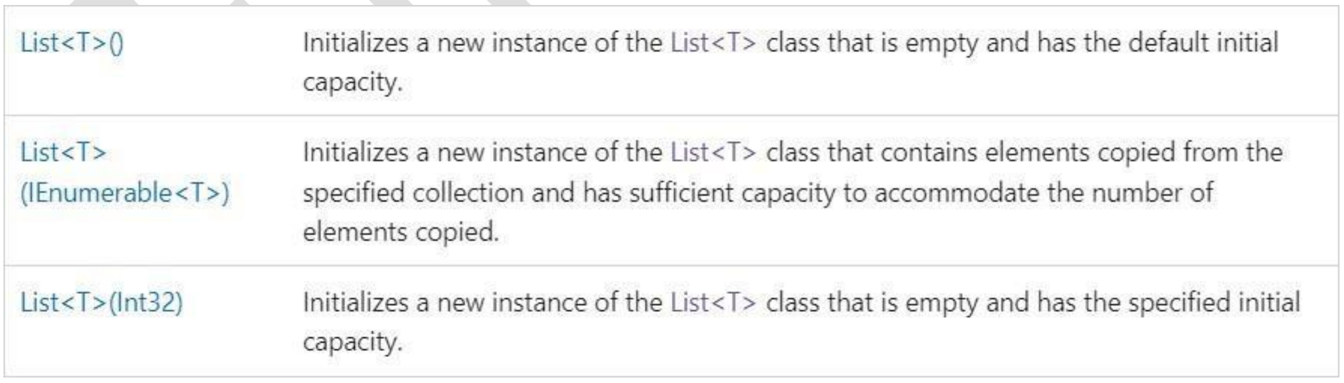
Implements : ICollection IEnumerable IList IReadOnlyCollection IReadOnlyList ICollection IEnumerable IList

- List class represents the list of objects which can be accessed by index. It comes under the System.Collections.Generic namespace. List class can be used to create a collection of different types like integers, strings etc. List class also provides the methods to search, sort, and manipulate lists.

Characteristics:

- It is different from the arrays. A List can be resized dynamically but arrays cannot.
- List class can accept null as a valid value for reference types and it also allows duplicate elements.
- If the Count becomes equals to Capacity, then the capacity of the List increased automatically by reallocating the internal array. The existing elements will be copied to the new array before the addition of the new element.
- List class is the generic equivalent of ArrayList class by implementing the IList generic interface.
- This class can use both equality and ordering comparer.
- List class is not sorted by default and elements are accessed by zero-based index.
- For very large List objects, you can increase the maximum capacity to 2 billion elements on a 64-bit system by setting the enabled attribute of the configuration element to true in the run-time environment.

Constructors

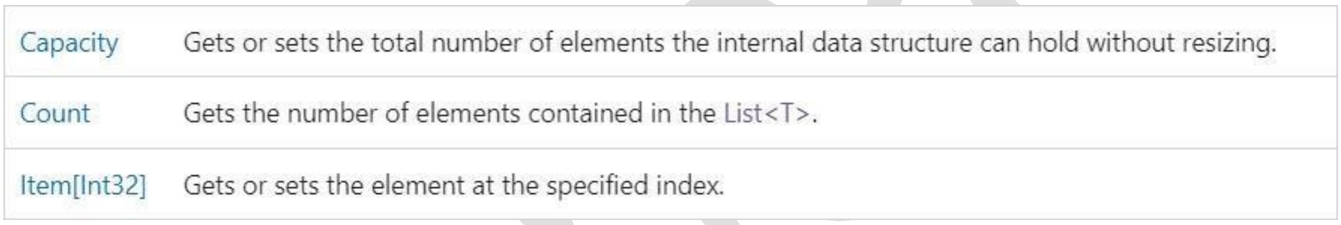


List<T>()	Initializes a new instance of the List<T> class that is empty and has the default initial capacity.
List<T>(IEnumerable<T>)	Initializes a new instance of the List<T> class that contains elements copied from the specified collection and has sufficient capacity to accommodate the number of elements copied.
List<T>(Int32)	Initializes a new instance of the List<T> class that is empty and has the specified initial capacity.

Example

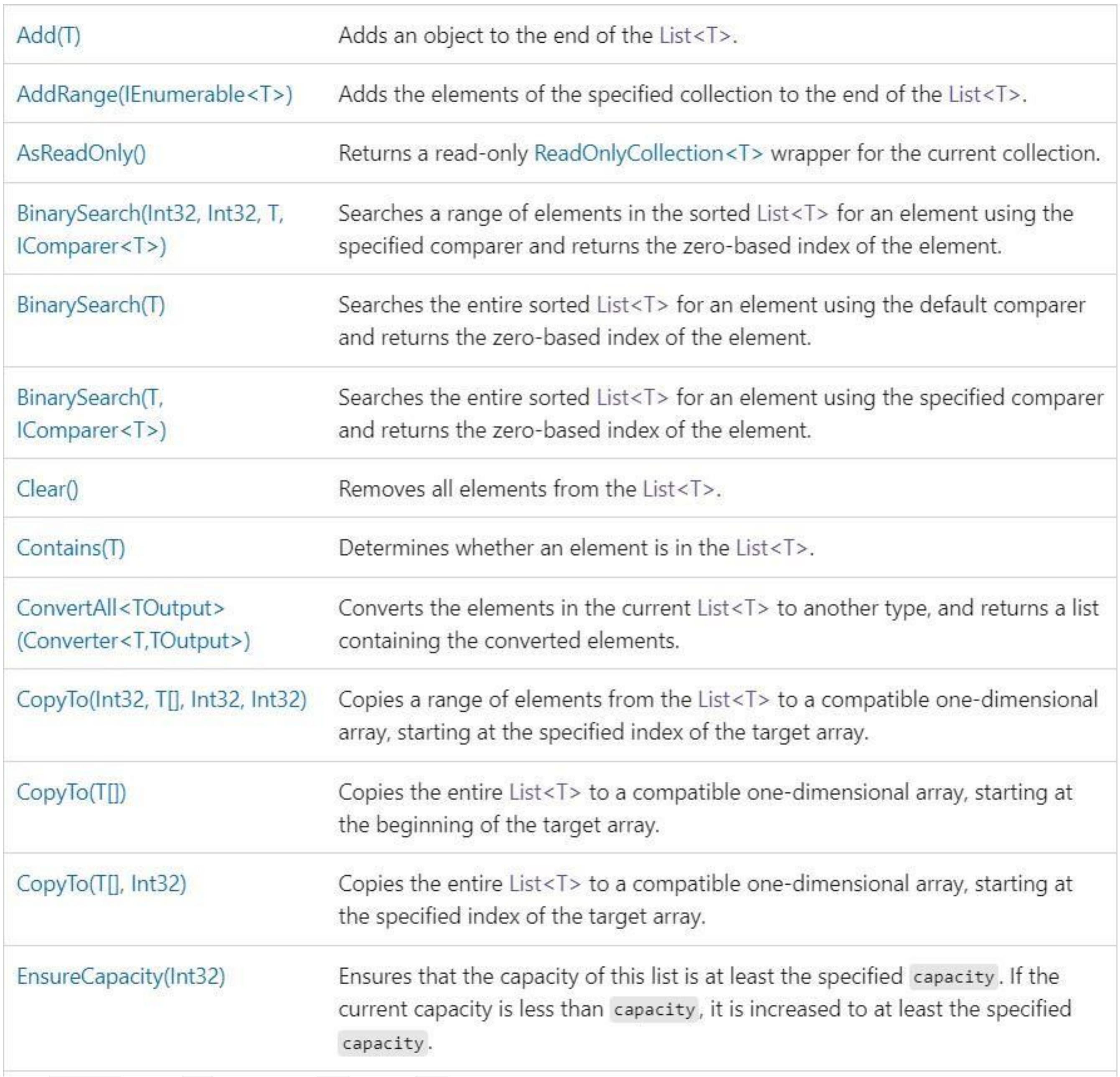
```
// C# program to create a List<T>
using System;
using System.Collections.Generic;
```

Properties

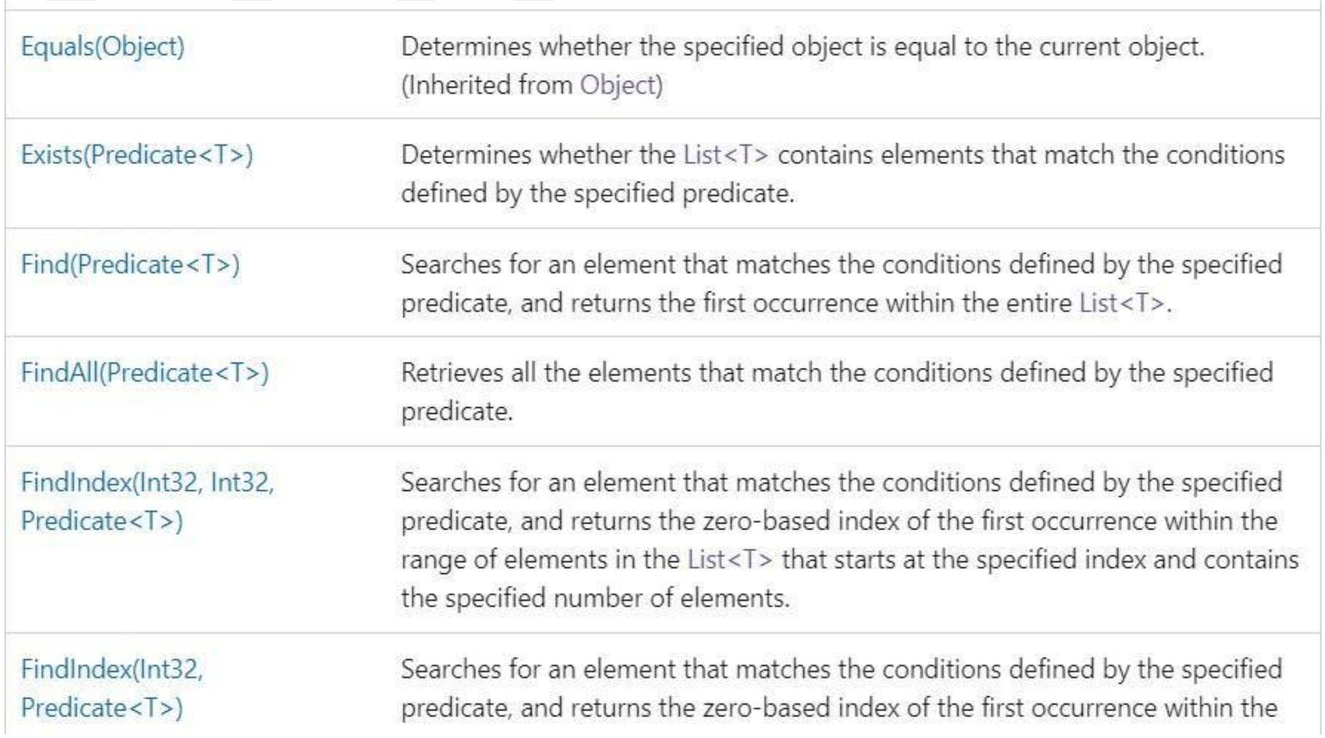


Capacity	Gets or sets the total number of elements the internal data structure can hold without resizing.
Count	Gets the number of elements contained in the List<T>.
Item[Int32]	Gets or sets the element at the specified index.

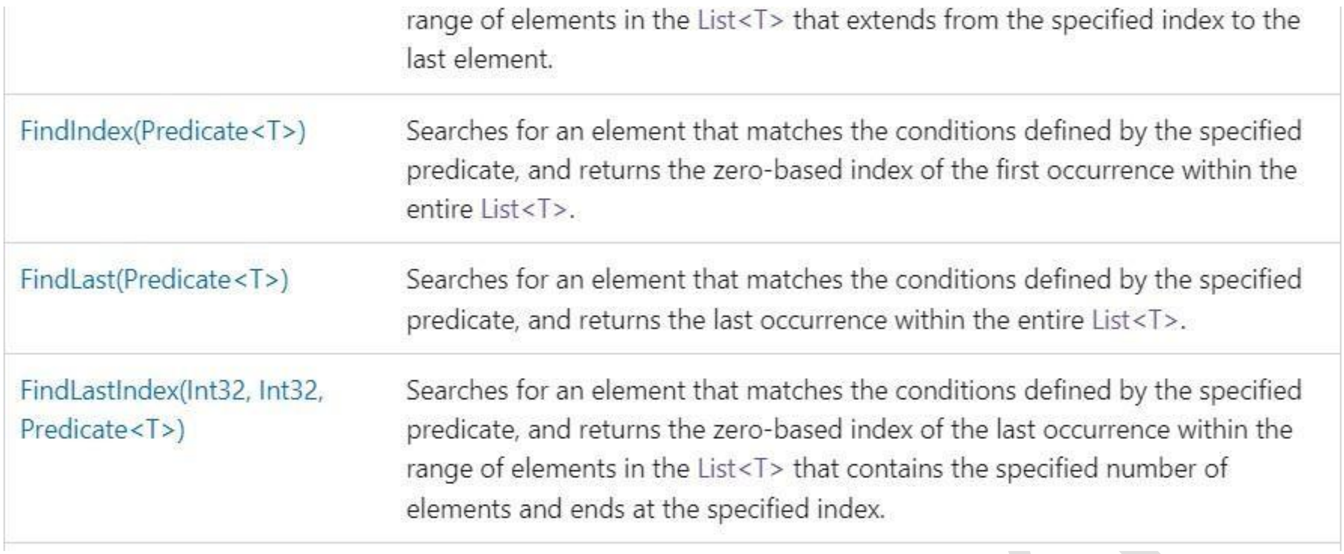
Methods



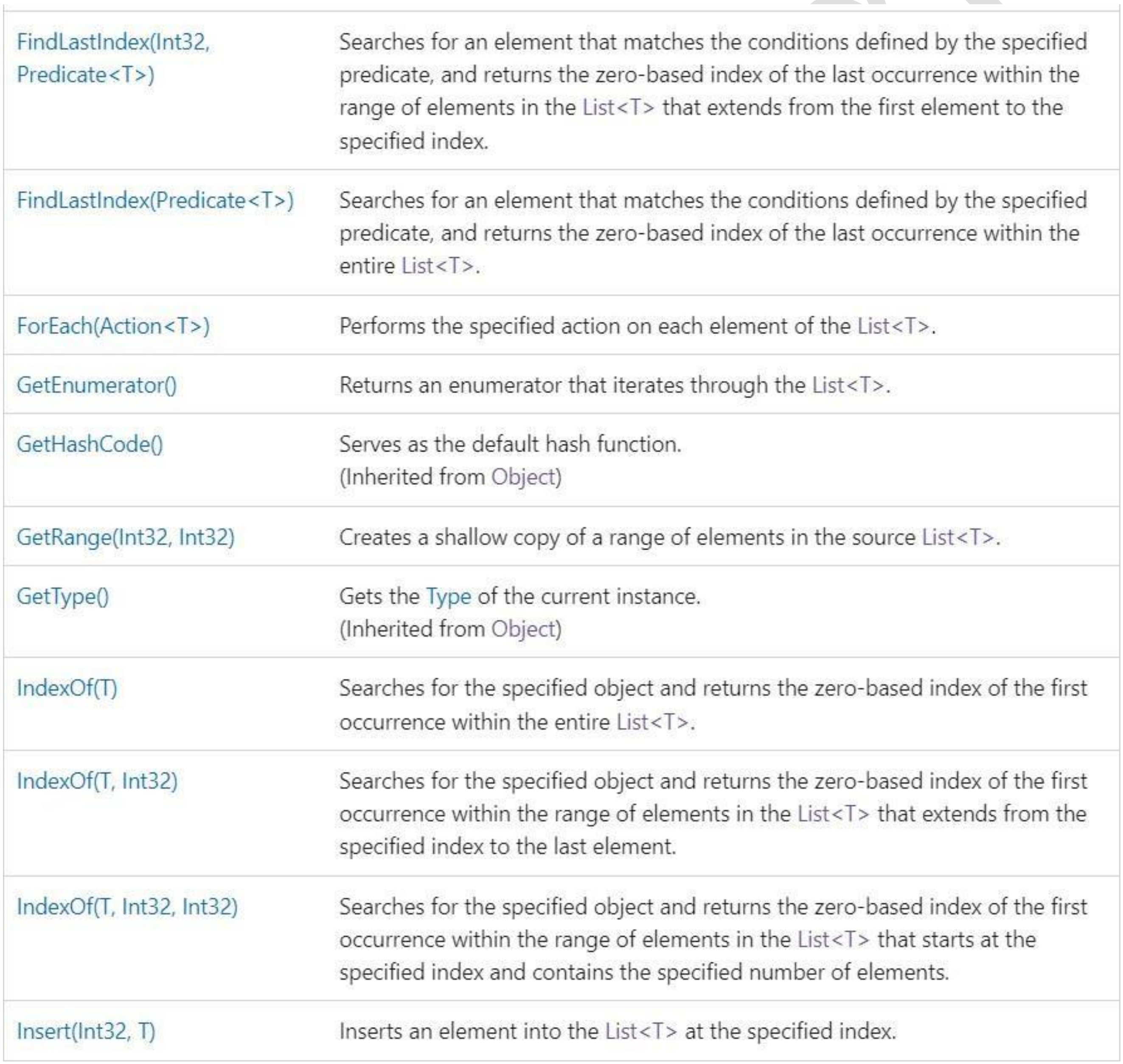
Add(T)	Adds an object to the end of the List<T>.
AddRange(IEnumerable<T>)	Adds the elements of the specified collection to the end of the List<T>.
AsReadOnly()	Returns a read-only ReadOnlyCollection<T> wrapper for the current collection.
BinarySearch(Int32, Int32, T, IComparer<T>)	Searches a range of elements in the sorted List<T> for an element using the specified comparer and returns the zero-based index of the element.
BinarySearch(T)	Searches the entire sorted List<T> for an element using the default comparer and returns the zero-based index of the element.
BinarySearch(T, IComparer<T>)	Searches the entire sorted List<T> for an element using the specified comparer and returns the zero-based index of the element.
Clear()	Removes all elements from the List<T>.
Contains(T)	Determines whether an element is in the List<T>.
ConvertAll<TOutput>(Converter<T,TOutput>)	Converts the elements in the current List<T> to another type, and returns a list containing the converted elements.
CopyTo(Int32, T[], Int32, Int32)	Copies a range of elements from the List<T> to a compatible one-dimensional array, starting at the specified index of the target array.
CopyTo(T[])	Copies the entire List<T> to a compatible one-dimensional array, starting at the beginning of the target array.
CopyTo(T[], Int32)	Copies the entire List<T> to a compatible one-dimensional array, starting at the specified index of the target array.
EnsureCapacity(Int32)	Ensures that the capacity of this list is at least the specified `capacity`. If the current capacity is less than `capacity`, it is increased to at least the specified `capacity`.



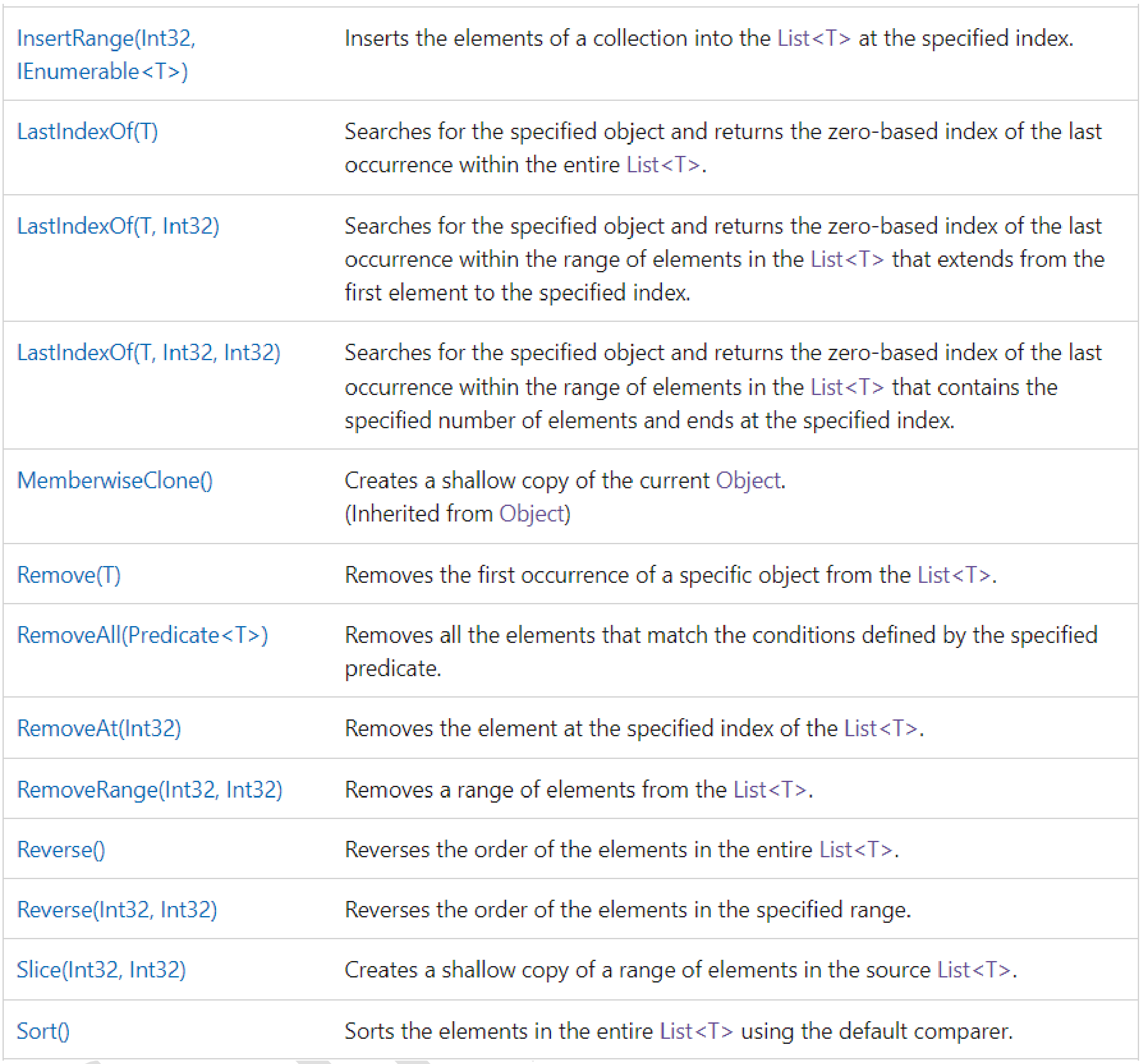
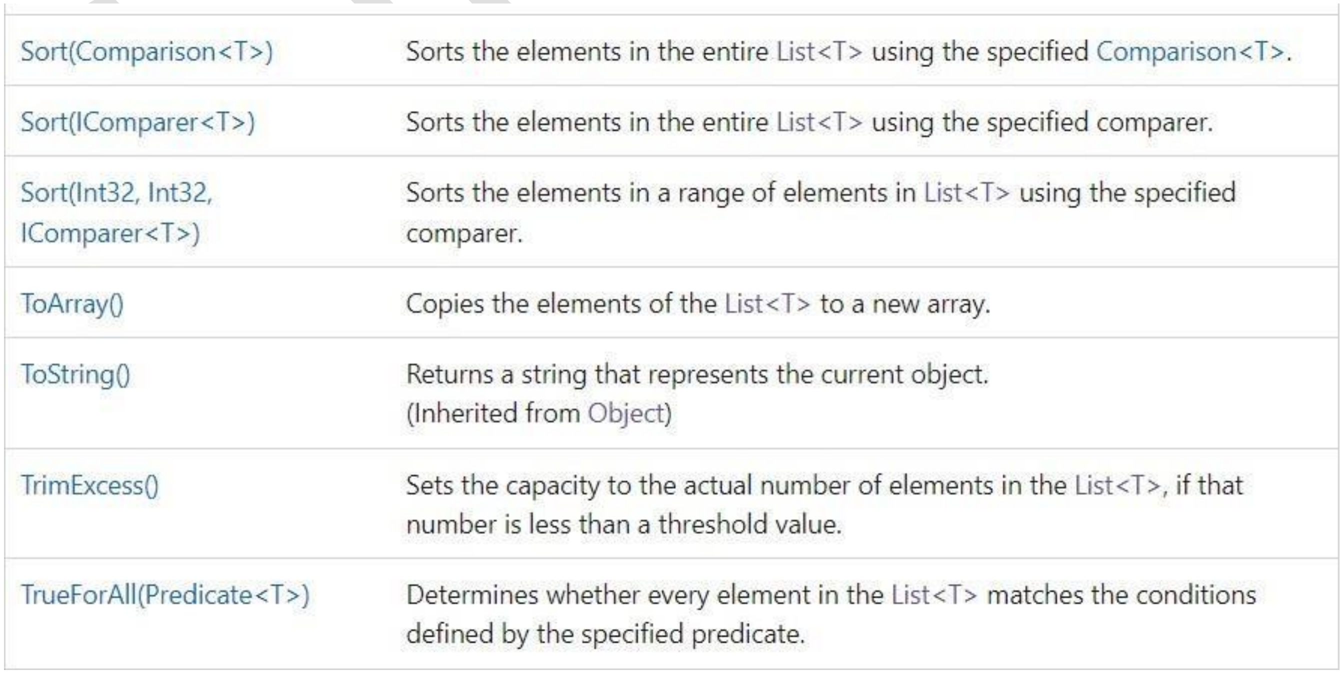
Equals(Object)	Determines whether the specified object is equal to the current object. (Inherited from Object)
Exists(Predicate<T>)	Determines whether the List<T> contains elements that match the conditions defined by the specified predicate.
Find(Predicate<T>)	Searches for an element that matches the conditions defined by the specified predicate, and returns the first occurrence within the entire List<T>.
FindAll(Predicate<T>)	Retrieves all the elements that match the conditions defined by the specified predicate.
FindIndex(Int32, Int32, Predicate<T>)	Searches for an element that matches the conditions defined by the specified predicate, and returns the zero-based index of the first occurrence within the range of elements in the List<T> that starts at the specified index and contains the specified number of elements.
FindIndex(Int32, Predicate<T>)	Searches for an element that matches the conditions defined by the specified predicate, and returns the zero-based index of the first occurrence within the



	range of elements in the List<T> that extends from the specified index to the last element.
FindIndex(Predicate<T>)	Searches for an element that matches the conditions defined by the specified predicate, and returns the zero-based index of the first occurrence within the entire List<T>.
FindLast(Predicate<T>)	Searches for an element that matches the conditions defined by the specified predicate, and returns the last occurrence within the entire List<T>.
FindLastIndex(Int32, Int32, Predicate<T>)	Searches for an element that matches the conditions defined by the specified predicate, and returns the zero-based index of the last occurrence within the range of elements in the List<T> that contains the specified number of elements and ends at the specified index.



FindLastIndex(Int32, Predicate<T>)	Searches for an element that matches the conditions defined by the specified predicate, and returns the zero-based index of the last occurrence within the range of elements in the List<T> that extends from the first element to the specified index.
FindLastIndex(Predicate<T>)	Searches for an element that matches the conditions defined by the specified predicate, and returns the zero-based index of the last occurrence within the entire List<T>.
ForEach(Action<T>)	Performs the specified action on each element of the List<T>.
GetEnumerator()	Returns an enumerator that iterates through the List<T>.
GetHashCode()	Serves as the default hash function. (Inherited from Object)
GetRange(Int32, Int32)	Creates a shallow copy of a range of elements in the source List<T>.
GetType()	Gets the Type of the current instance. (Inherited from Object)
IndexOf(T)	Searches for the specified object and returns the zero-based index of the first occurrence within the entire List<T>.
IndexOf(T, Int32)	Searches for the specified object and returns the zero-based index of the first occurrence within the range of elements in the List<T> that extends from the specified index to the last element.
IndexOf(T, Int32, Int32)	Searches for the specified object and returns the zero-based index of the first occurrence within the range of elements in the List<T> that starts at the specified index and contains the specified number of elements.
Insert(Int32, T)	Inserts an element into the List<T> at the specified index.

InsertRange(Int32, IEnumerable<T>)	Inserts the elements of a collection into the List<T> at the specified index.
LastIndexOf(T)	Searches for the specified object and returns the zero-based index of the last occurrence within the entire List<T>.
LastIndexOf(T, Int32)	Searches for the specified object and returns the zero-based index of the last occurrence within the range of elements in the List<T> that extends from the first element to the specified index.
LastIndexOf(T, Int32, Int32)	Searches for the specified object and returns the zero-based index of the last occurrence within the range of elements in the List<T> that contains the specified number of elements and ends at the specified index.
MemberwiseClone()	Creates a shallow copy of the current Object. (Inherited from Object)
Remove(T)	Removes the first occurrence of a specific object from the List<T>.
RemoveAll(Predicate<T>)	Removes all the elements that match the conditions defined by the specified predicate.
RemoveAt(Int32)	Removes the element at the specified index of the List<T>.
RemoveRange(Int32, Int32)	Removes a range of elements from the List<T>.
Reverse()	Reverses the order of the elements in the entire List<T>.
Reverse(Int32, Int32)	Reverses the order of the elements in the specified range.
Slice(Int32, Int32)	Creates a shallow copy of a range of elements in the source List<T>.
Sort()	Sorts the elements in the entire List<T> using the default comparer.
Sort(Comparison<T>)	Sorts the elements in the entire List<T> using the specified Comparison<T>.
Sort(IComparer<T>)	Sorts the elements in the entire List<T> using the specified comparer.
Sort(Int32, Int32, IComparer<T>)	Sorts the elements in a range of elements in List<T> using the specified comparer.
ToArray()	Copies the elements of the List<T> to a new array.
ToString()	Returns a string that represents the current object. (Inherited from Object)
TrimExcess()	Sets the capacity to the actual number of elements in the List<T>, if that number is less than a threshold value.
TrueForAll(Predicate<T>)	Determines whether every element in the List<T> matches the conditions defined by the specified predicate.

Example

```
// C# Program to remove the element at
// the specified index of the List<T>
using System;
using System.Collections.Generic;

class Demo {

    // Main Method
    public static void Main(String[] args)
    {

        // Creating an List<T> of Integers
        List<int> firstlist = new List<int>();

        // Adding elements to List
        firstlist.Add(17);
        firstlist.Add(19);
        firstlist.Add(21);
        firstlist.Add(9);
        firstlist.Add(75);
        firstlist.Add(19);
        firstlist.Add(73);

        Console.WriteLine("Elements Present in List:\n");

        int p = 0;

        // Displaying the elements of List
        foreach(int k in firstlist)
        {
            Console.Write("At Position {0}: ", p);
            Console.WriteLine(k);
            p++;
        }

        Console.WriteLine(" ");

        // removing the element at index 3
        Console.WriteLine("Removing the element at index 3\n");

        // 9 will remove from the List
        // and 75 will come at index 3
        firstlist.RemoveAt(3);

        int p1 = 0;

        // Displaying the elements of List
        foreach(int n in firstlist)
        {
            Console.Write("At Position {0}: ", p1);
            Console.WriteLine(n);
            p1++;
        }
    }
}
/*
Elements Present in List:

At Position 0: 17
At Position 1: 19
At Position 2: 21
At Position 3: 9
At Position 4: 75
At Position 5: 19
At Position 6: 73

Removing the element at index 3

At Position 0: 17
At Position 1: 19
At Position 2: 21
At Position 3: 75
At Position 4: 19
At Position 5: 73
*/
```

Reference

Stack class

- Stack represents a last-in, first out collection of object. It is used when you need a last-in, first-out access to items. When you add an item in the list, it is called pushing the item and when you remove it, it is called popping the item. This class comes under System.Collections.Generic namespace.

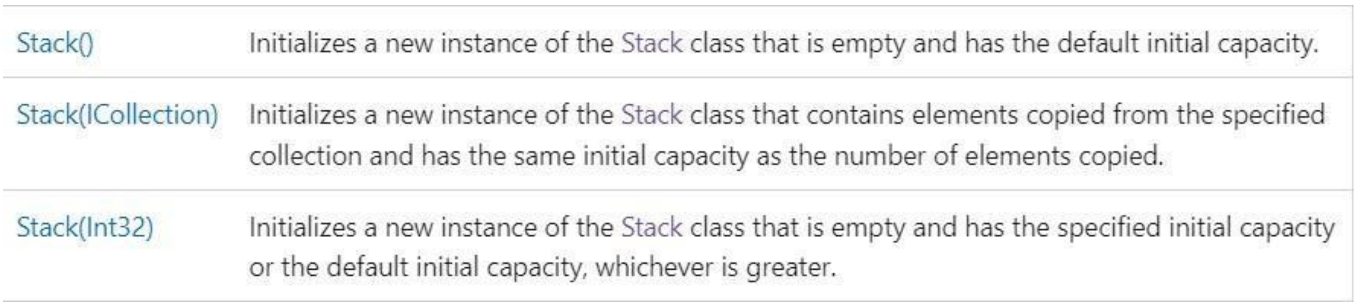
Characteristics of Stack Class:

- Stack is implemented as an array.
- Stacks and queues are useful when you need temporary storage for information; that is, when you might want to discard an element after retrieving its value. Use Queue if you need to access the information in the same order that it is stored in the collection. Use System.Collections.Generic.Stack if you need to access the information in reverse order.
- Use the System.Collections.Concurrent.ConcurrentStack and System.Collections.Concurrent.ConcurrentQueue types when you need to access the collection from multiple threads concurrently.
- A common use for System.Collections.Generic.Stack is to preserve variable states during calls to other procedures.
- Three main operations can be performed on a System.Collections.Generic.Stack and its elements: - Push inserts an element at the top of the Stack. - Pop removes an element from the top of the Stack. - Peek returns an element that is at the top of the Stack but does not remove it from the Stack.

- The capacity of a Stack is the number of elements the Stack can hold. As elements are added to a Stack, the capacity is automatically increased as required by reallocating the internal array. The capacity can be decreased by calling TrimExcess.
- Stack accepts null as a valid value for reference types and allows duplicate elements.
- The following example shows how to create and add values to a Stack and how to display its values.

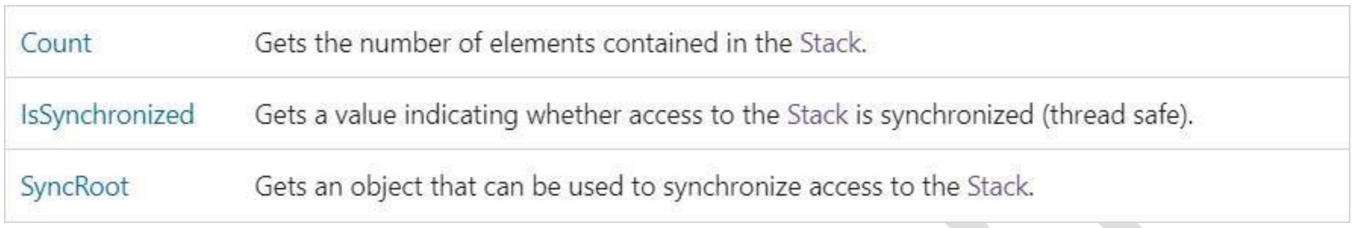
```
using System;
using System.Collections.Generic;

class Example
{
    public static void Main()
    {
        Stack<string> numbers = new Stack<string>();
        numbers.Push("one");
        numbers.Push("two");
        numbers.Push("three");
        numbers.Push("four");
        numbers.Push("five");

        // A stack can be enumerated without disturbing its contents.
        foreach( string number in numbers )
        {
            Console.WriteLine(number);
        }

        Console.WriteLine("\nPopping '{0}'", numbers.Pop());
        Console.WriteLine("Peek at next item to destack: {0}",
            numbers.Peek());
        Console.WriteLine("Popping '{0}'", numbers.Pop());

        // Create a copy of the stack, using the ToArray method and the
        // constructor that accepts an IEnumerable<T>.
        Stack<string> stack2 = new Stack<string>(numbers.ToArray());

        Console.WriteLine("\nContents of the first copy:");
        foreach( string number in stack2 )
        {
            Console.WriteLine(number);
        }

        // Create an array twice the size of the stack and copy the
        // elements of the stack, starting at the middle of the
        // array.
        string[] array2 = new string[numbers.Count * 2];
        numbers.CopyTo(array2, numbers.Count);

        // Create a second stack, using the constructor that accepts an
        // IEnumerable(Of T).
        Stack<string> stack3 = new Stack<string>(array2);

        Console.WriteLine("\nContents of the second copy, with duplicates and
nulls:");
        foreach( string number in stack3 )
        {
            Console.WriteLine(number);
        }

        Console.WriteLine("\nstack2.Contains(\"four\") = {0}",
            stack2.Contains("four"));

        Console.WriteLine("\nstack2.Clear()");
        stack2.Clear();
        Console.WriteLine("\nstack2.Count = {0}", stack2.Count);
    }
}

/* This code example produces the following output:

five
four
three
two
one

Popping 'five'
Peek at next item to destack: four
Popping 'four'

Contents of the first copy:
one
two
three

Contents of the second copy, with duplicates and nulls:
one
two
three




stack2.Contains("four") = False

stack2.Clear()

stack2.Count = 0
 */
```

Constructors



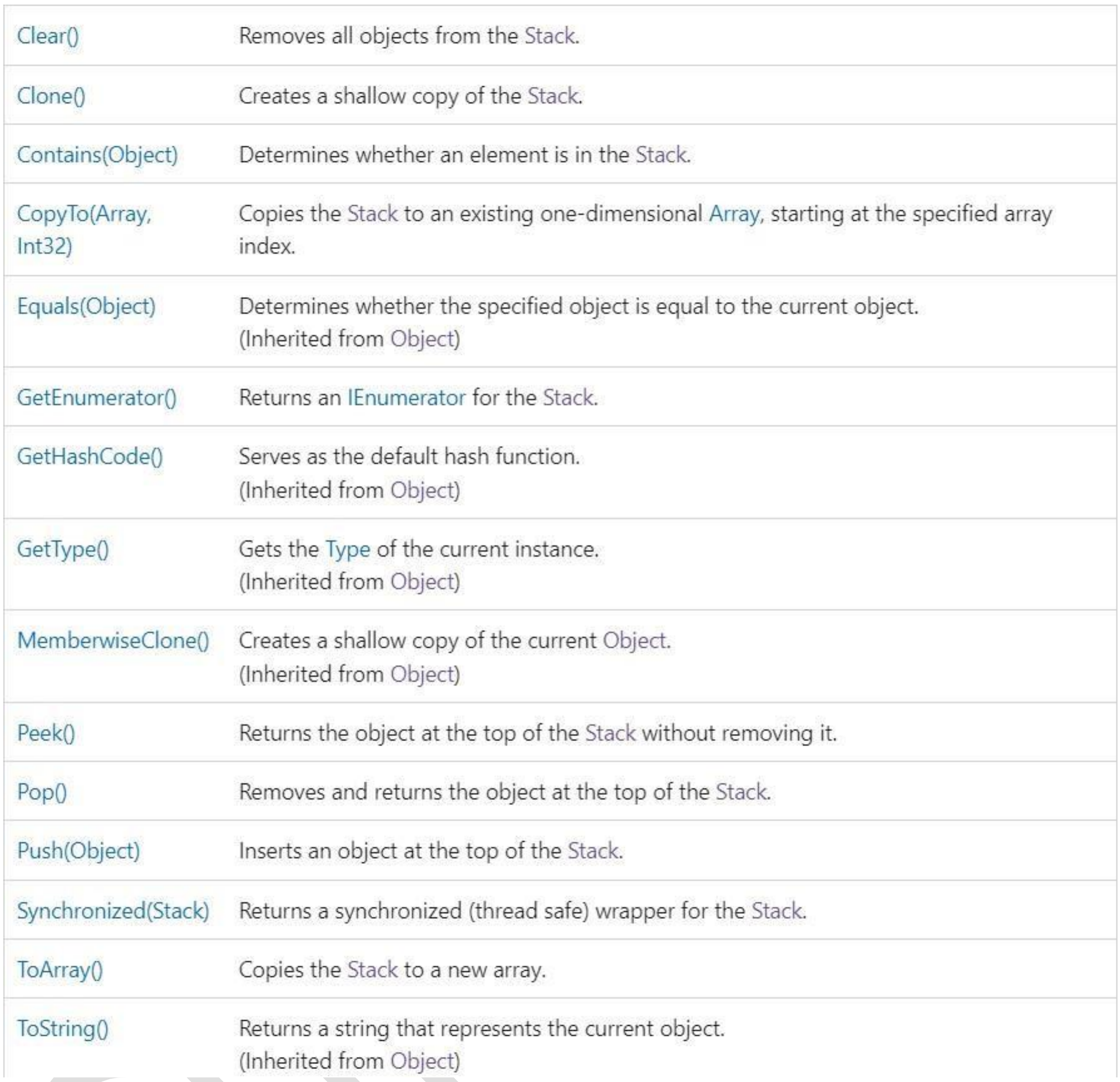
Stack()	Initializes a new instance of the Stack class that is empty and has the default initial capacity.
Stack(ICollection)	Initializes a new instance of the Stack class that contains elements copied from the specified collection and has the same initial capacity as the number of elements copied.
Stack(Int32)	Initializes a new instance of the Stack class that is empty and has the specified initial capacity or the default initial capacity, whichever is greater.

Properties



Count	Gets the number of elements contained in the Stack.
IsSynchronized	Gets a value indicating whether access to the Stack is synchronized (thread safe).
SyncRoot	Gets an object that can be used to synchronize access to the Stack.

Methods



Clear()	Removes all objects from the Stack.
Clone()	Creates a shallow copy of the Stack.
Contains(Object)	Determines whether an element is in the Stack.
CopyTo(Array, Int32)	Copies the Stack to an existing one-dimensional Array, starting at the specified array index.
Equals(Object)	Determines whether the specified object is equal to the current object. (Inherited from Object)
GetEnumerator()	Returns an IEnumerator for the Stack.
GetHashCode()	Serves as the default hash function. (Inherited from Object)
GetType()	Gets the Type of the current instance. (Inherited from Object)
MemberwiseClone()	Creates a shallow copy of the current Object. (Inherited from Object)
Peek()	Returns the object at the top of the Stack without removing it.
Pop()	Removes and returns the object at the top of the Stack.
Push(Object)	Inserts an object at the top of the Stack.
Synchronized(Stack)	Returns a synchronized (thread safe) wrapper for the Stack.
ToArray()	Copies the Stack to a new array.
ToString()	Returns a string that represents the current object. (Inherited from Object)

Reference

Queue class

Namespace: System.Collections.Generic Inheritance: Object -> Queue Implements: IEnumerable IReadOnlyCollection ICollection IEnumerable

- This class implements a generic queue as a circular array. Objects stored in a Queue are inserted at one end and removed from the other. Queues and stacks are useful when you need temporary storage for information; that is, when you might want to discard an element after retrieving its value. Use Queue if you need to access the information in the same order that it is stored in the collection. Use Stack if you need to access the information in reverse order. Use ConcurrentQueue or ConcurrentStack if you need to access the collection from multiple threads concurrently.
- Three main operations can be performed on a Queue and its elements: - Enqueue adds an element to the end of the Queue. - Dequeue removes the oldest element from the start of the Queue. - Peek peek

returns the oldest element that is at the start of the Queue but does not remove it from the Queue.

- The capacity of a Queue is the number of elements the Queue can hold. As elements are added to a Queue, the capacity is automatically increased as required by reallocating the internal array. The capacity can be decreased by calling TrimExcess.
- Queue accepts null as a valid value for reference types and allows duplicate elements.

Examples

The following code example demonstrates several methods of the Queue generic class. The code example creates a queue of strings with default capacity and uses the Enqueue method to queue five strings. The elements of the queue are enumerated, which does not change the state of the queue. The Dequeue method is used to dequeue the first string. The Peek method is used to look at the next item in the queue, and then the Dequeue method is used to dequeue it.

The ToArray method is used to create an array and copy the queue elements to it, then the array is passed to the Queue constructor that takes IEnumerable, creating a copy of the queue. The elements of the copy are displayed.

An array twice the size of the queue is created, and the CopyTo method is used to copy the array elements beginning at the middle of the array. The Queue constructor is used again to create a second copy of the queue containing three null elements at the beginning.

The Contains method is used to show that the string "four" is in the first copy of the queue, after which the Clear method clears the copy and the Count property shows that the queue is empty.

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

class Example
{
    public static void Main()
    {
        Queue<string> numbers = new Queue<string>();
        numbers.Enqueue("one");
        numbers.Enqueue("two");
        numbers.Enqueue("three");
        numbers.Enqueue("four");
        numbers.Enqueue("five");

        // A queue can be enumerated without disturbing its contents.
        foreach( string number in numbers )
        {
            Console.WriteLine(number);
        }

        Console.WriteLine("\nDequeuing '{0}'", numbers.Dequeue());
        Console.WriteLine("Peek at next item to dequeue: {0}",
            numbers.Peek());
        Console.WriteLine("Dequeuing '{0}'", numbers.Dequeue());
        // Create a copy of the queue, using the ToArray method and the
        // constructor that accepts an IEnumerable<T>.
        Queue<string> queueCopy = new Queue<string>(numbers.ToArray());

        Console.WriteLine("\nContents of the first copy:");
        foreach( string number in queueCopy )
        {
            Console.WriteLine(number);
        }

        // Create an array twice the size of the queue and copy the
        // elements of the queue, starting at the middle of the
        // array.
        string[] array2 = new string[numbers.Count * 2];
        numbers.CopyTo(array2, numbers.Count);

        // Create a second queue, using the constructor that accepts an
        // IEnumerable(Of T).
        Queue<string> queueCopy2 = new Queue<string>(array2);

        Console.WriteLine("\nContents of the second copy, with duplicates and
nulls:");
        foreach( string number in queueCopy2 )
        {
            Console.WriteLine(number);
        }

        Console.WriteLine("\nqueueCopy.Contains(\"four\") = {0}",
            queueCopy.Contains("four"));

        Console.WriteLine("\nqueueCopy.Clear()");
        queueCopy.Clear();
        Console.WriteLine("\nqueueCopy.Count = {0}", queueCopy.Count);
    }
}

/* This code example produces the following output:

one
two
three
four
five

Dequeuing 'one'
Peek at next item to dequeue: two
Dequeuing 'two'

Contents of the first copy:
three
four
five
```

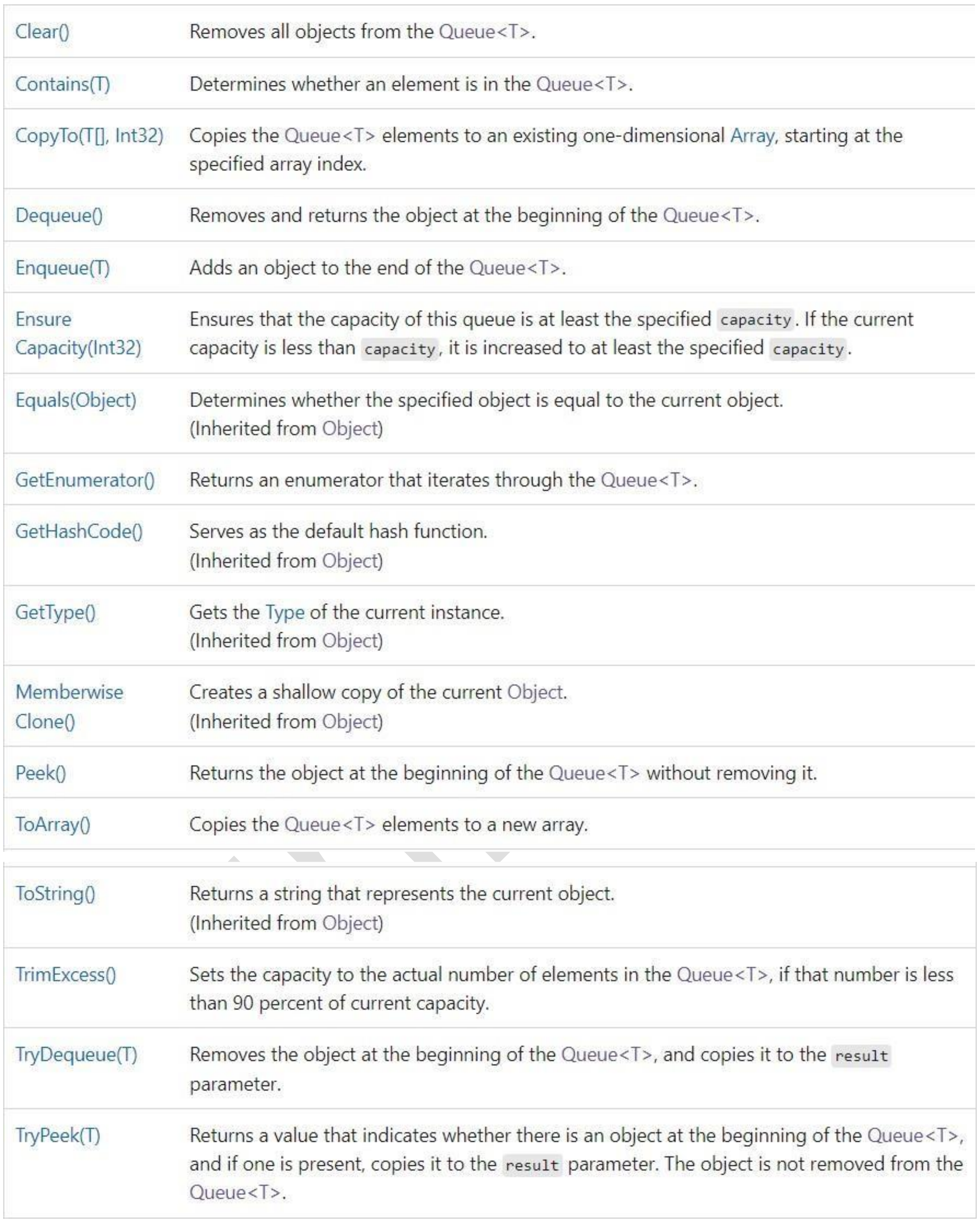
Constructor

Queue<T>()	Initializes a new instance of the Queue<T> class that is empty and has the default initial capacity.
Queue<T>(IEnumerable<T>)	Initializes a new instance of the Queue<T> class that contains elements copied from the specified collection and has sufficient capacity to accommodate the number of elements copied.
Queue<T>(Int32)	Initializes a new instance of the Queue<T> class that is empty and has the specified initial capacity.

Properties

Count	Gets the number of elements contained in the Queue<T>.

Methods



Clear()	Removes all objects from the Queue<T>.
Contains(T)	Determines whether an element is in the Queue<T>.
CopyTo(T[], Int32)	Copies the Queue<T> elements to an existing one-dimensional Array, starting at the specified array index.
Dequeue()	Removes and returns the object at the beginning of the Queue<T>.
Enqueue(T)	Adds an object to the end of the Queue<T>.
Ensure Capacity(Int32)	Ensures that the capacity of this queue is at least the specified `capacity`. If the current capacity is less than `capacity`, it is increased to at least the specified `capacity`.
Equals(Object)	Determines whether the specified object is equal to the current object. (Inherited from Object)
GetEnumerator()	Returns an enumerator that iterates through the Queue<T>.
GetHashCode()	Serves as the default hash function. (Inherited from Object)
GetType()	Gets the Type of the current instance. (Inherited from Object)
Memberwise Clone()	Creates a shallow copy of the current Object. (Inherited from Object)
Peek()	Returns the object at the beginning of the Queue<T> without removing it.
ToArray()	Copies the Queue<T> elements to a new array.
ToString()	Returns a string that represents the current object. (Inherited from Object)
TrimExcess()	Sets the capacity to the actual number of elements in the Queue<T>, if that number is less than 90 percent of current capacity.
TryDequeue(T)	Removes the object at the beginning of the Queue<T>, and copies it to the `result` parameter.
TryPeek(T)	Returns a value that indicates whether there is an object at the beginning of the Queue<T>, and if one is present, copies it to the `result` parameter. The object is not removed from the Queue<T>.

Dictionary class

- In C#, Dictionary is a generic collection which is generally used to store key/value pairs. The working of Dictionary is quite similar to the non-generic hashtable. The advantage of Dictionary is, it is generic type. Dictionary is defined under System.Collections.Generic namespace. It is dynamic in nature means the size of the dictionary is grows according to the need.

Important Points:

- The Dictionary class implements the - IDictionary<TKey,TValue> Interface - IReadOnlyCollection<KeyValuePair<TKey,TValue>> Interface - IReadOnlyDictionary<TKey,TValue> Interface - IDictionary Interface
- In Dictionary, the key cannot be null, but value can be.
- In Dictionary, key must be unique. Duplicate keys are not allowed if you try to use duplicate key then compiler will throw an exception.
- In Dictionary, you can only store same types of elements.
- The capacity of a Dictionary is the number of elements that Dictionary can hold.

Examples

The following code example creates an empty Dictionary<TKey,TValue> of strings with string keys and uses the Add method to add some elements. The example demonstrates that the Add method throws an ArgumentException when attempting to add a duplicate key.

The example uses the Item[] property (the indexer in C#) to retrieve values, demonstrating that a KeyNotFoundException is thrown when a requested key is not present, and showing that the value associated with a key can be replaced.

The example shows how to use the TryGetValue method as a more efficient way to retrieve values if a program often must try key values that are not in the dictionary, and it shows how to use the ContainsKey method to test whether a key exists before calling the Add method.

The example shows how to enumerate the keys and values in the dictionary and how to enumerate the keys and values alone using the Keys property and the Values property.

Finally, the example demonstrates the Remove method.

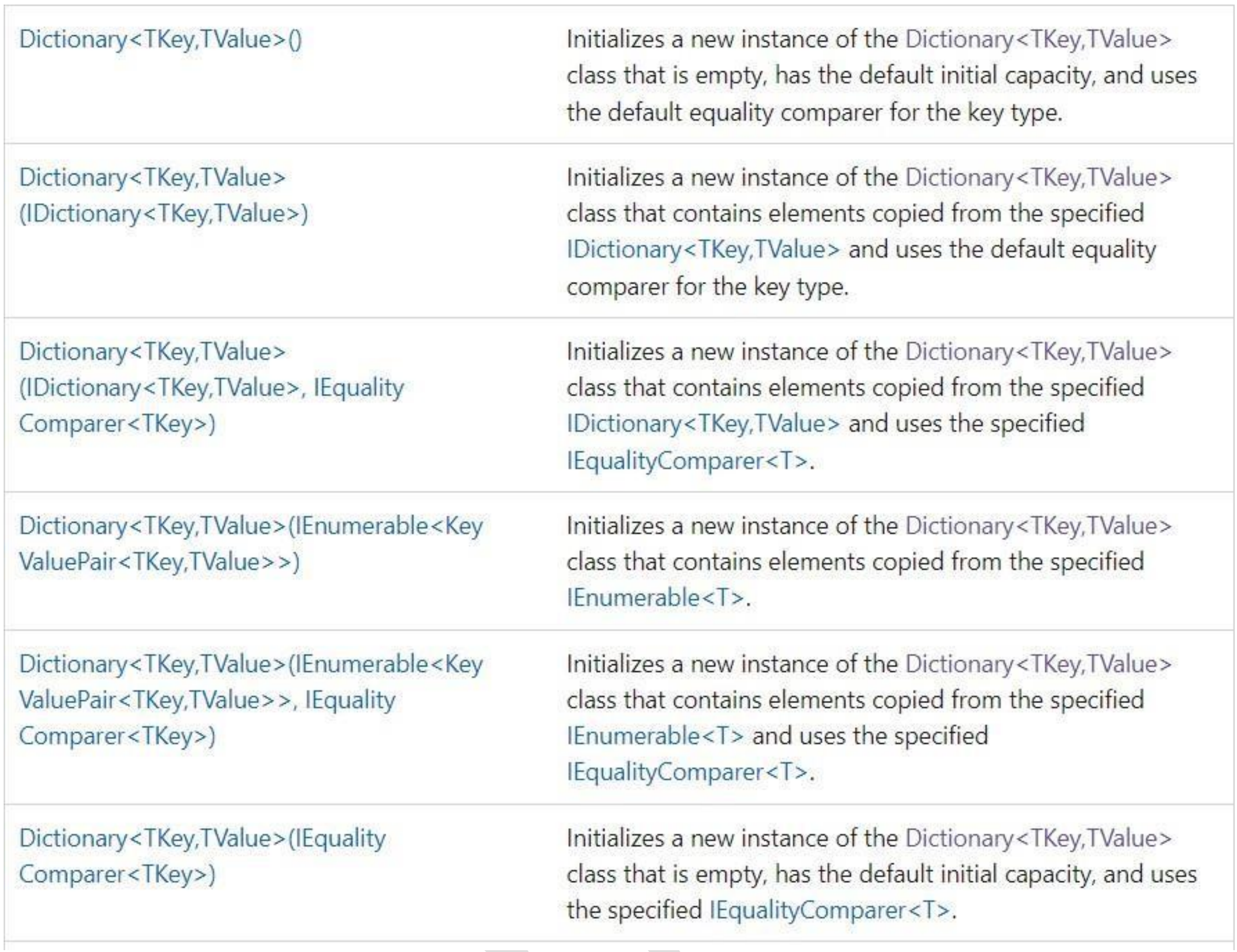
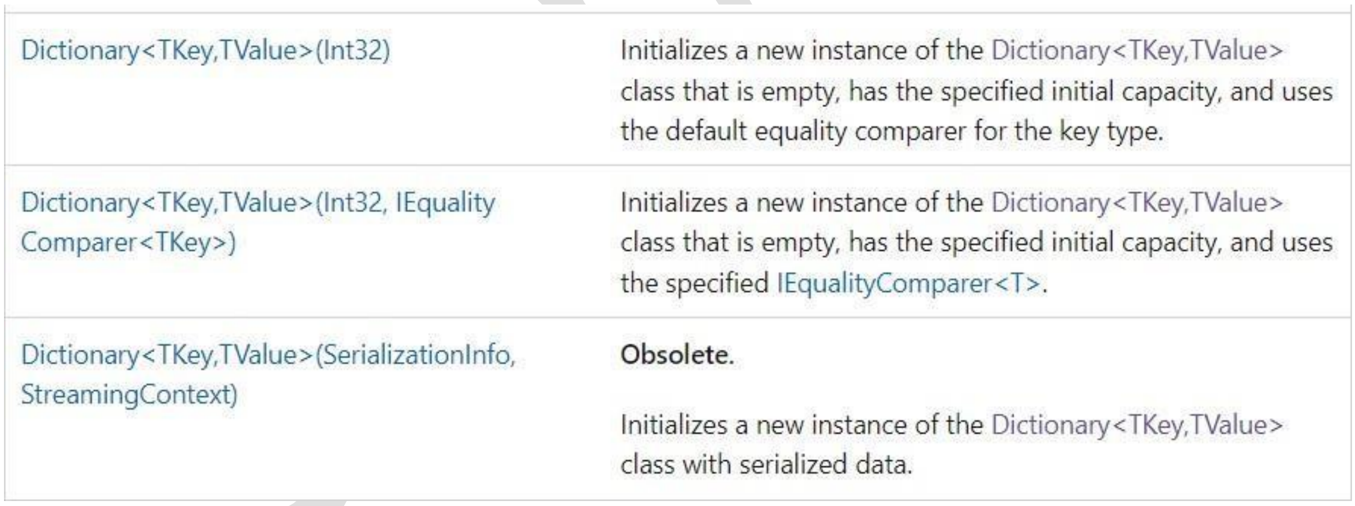
```
// Create a new dictionary of strings, with string keys.
//
Dictionary<string, string> openWith =
    new Dictionary<string, string>();

// Add some elements to the dictionary. There are no
// duplicate keys, but some of the values are duplicates.
openWith.Add("txt", "notepad.exe");
openWith.Add("bmp", "paint.exe");
openWith.Add("dib", "paint.exe");
openWith.Add("rtf", "wordpad.exe");

// The Add method throws an exception if the new key is
// already in the dictionary.
try
{
    openWith.Add("txt", "winword.exe");
}
catch (ArgumentException)
{
    Console.WriteLine("An element with Key = \"txt\" already exists.");
}

// The Item property is another name for the indexer, so you
// can omit its name when accessing elements.
Console.WriteLine("For key = \"rtf\", value = {0}.",
    openWith["rtf"]);

// The indexer can be used to change the value associated
// with a key.
openWith["rtf"] = "winword.exe";
Console.WriteLine("For key = \"rtf\", value = {0}.",
    openWith["rtf"]);

// If a key does not exist, setting the indexer for that key
// adds a new key/value pair.
openWith["doc"] = "winword.exe";

// The indexer throws an exception if the requested key is
// not in the dictionary.
try
{
    Console.WriteLine("For key = \"tif\", value = {0}.",
        openWith["tif"]);
}
catch (KeyNotFoundException)
{
    Console.WriteLine("Key = \"tif\" is not found.");
}

// When a program often has to try keys that turn out not to
// be in the dictionary, TryGetValue can be a more efficient
// way to retrieve values.
string value = "";
if (openWith.TryGetValue("tif", out value))
{
    Console.WriteLine("For key = \"tif\", value = {0}.", value);
}
else
{
    Console.WriteLine("Key = \"tif\" is not found.");
}


// ContainsKey can be used to test keys before inserting
// them.
if (!openWith.ContainsKey("ht"))
{
    openWith.Add("ht", "hypertrm.exe");
    Console.WriteLine("Value added for key = \"ht\": {0}",
        openWith["ht"]);
}

// When you use foreach to enumerate dictionary elements,
// the elements are retrieved as KeyValuePair objects.
```

```
Console.WriteLine();
foreach( KeyValuePair<string, string> kvp in openWith )
{
    Console.WriteLine("Key = {0}, Value = {1}",
        kvp.Key, kvp.Value);
}

// To get the values alone, use the Values property.
Dictionary<string, string>.ValueCollection valueColl =
    openWith.Values;

// The elements of the ValueCollection are strongly typed
// with the type that was specified for dictionary values.
Console.WriteLine();
foreach( string s in valueColl )
{
    Console.WriteLine("Value = {0}", s);
}

// To get the keys alone, use the Keys property.
Dictionary<string, string>.KeyCollection keyColl =
    openWith.Keys;

// The elements of the KeyCollection are strongly typed
// with the type that was specified for dictionary keys.
Console.WriteLine();
foreach( string s in keyColl )
{
    Console.WriteLine("Key = {0}", s);
}

// Use the Remove method to remove a key/value pair.
Console.WriteLine("\nRemove(\"doc\")");
openWith.Remove("doc");

if (!openWith.ContainsKey("doc"))
{
    Console.WriteLine("Key \"doc\" is not found.");
}

/* This code example produces the following output:

An element with Key = "txt" already exists.
For key = "rtf", value = wordpad.exe.
For key = "rtf", value = winword.exe.
Key = "tif" is not found.
Key = "tif" is not found.
Value added for key = "ht": hypertrm.exe

Key = txt, Value = notepad.exe
Key = bmp, Value = paint.exe
Key = dib, Value = paint.exe
Key = rtf, Value = winword.exe
Key = doc, Value = winword.exe
Key = ht, Value = hypertrm.exe

Value = notepad.exe
Value = paint.exe
Value = paint.exe
Value = winword.exe
Value = winword.exe
Value = hypertrm.exe

Key = txt
Key = bmp
Key = dib
Key = rtf
Key = doc
Key = ht

Remove("doc")
Key "doc" is not found.
*/
```

Remnarks

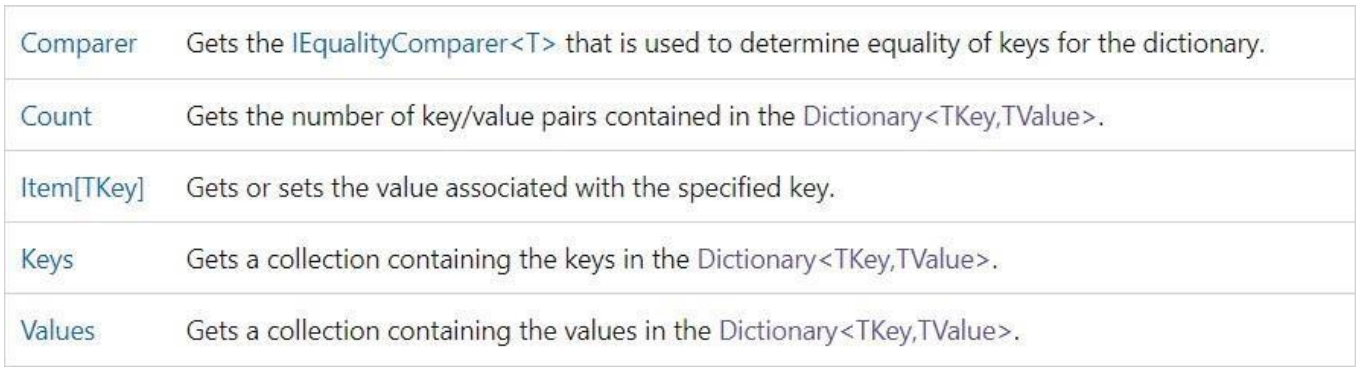
- The Dictionary<TKey,TValue> generic class provides a mapping from a set of keys to a set of values. Each addition to the dictionary consists of a value and its associated key. Retrieving a value by using its key is very fast, close to O(1), because the Dictionary<TKey,TValue> class is implemented as a hash table.
- As long as an object is used as a key in the Dictionary<TKey,TValue>, it must not change in any way that affects its hash value. Every key in a Dictionary<TKey,TValue> must be unique according to the dictionary's equality comparer. A key cannot be null, but a value can be, if its type TValue is a reference type.
- Dictionary<TKey,TValue> requires an equality implementation to determine whether keys are equal. You can specify an implementation of the IEqualityComparer generic interface by using a constructor that accepts a comparer parameter; if you do not specify an implementation, the default generic equality comparer EqualityComparer.Default is used. If type TKey implements the System.IEquatable generic interface, the default equality comparer uses that implementation.
- The capacity of a Dictionary<TKey,TValue> is the number of elements the Dictionary<TKey,TValue> can hold. As elements are added to a Dictionary<TKey,TValue>, the capacity is automatically increased as required by reallocating the internal array.
- .NET Framework only: For very large Dictionary<TKey,TValue> objects, you can increase the maximum capacity to 2 billion elements on a 64-bit system by setting the enabled attribute of the configuration element to true in the run-time environment.
- For purposes of enumeration, each item in the dictionary is treated as a KeyValuePair<TKey,TValue> structure representing a value and its key. The order in which the items are returned is undefined.

Constructor

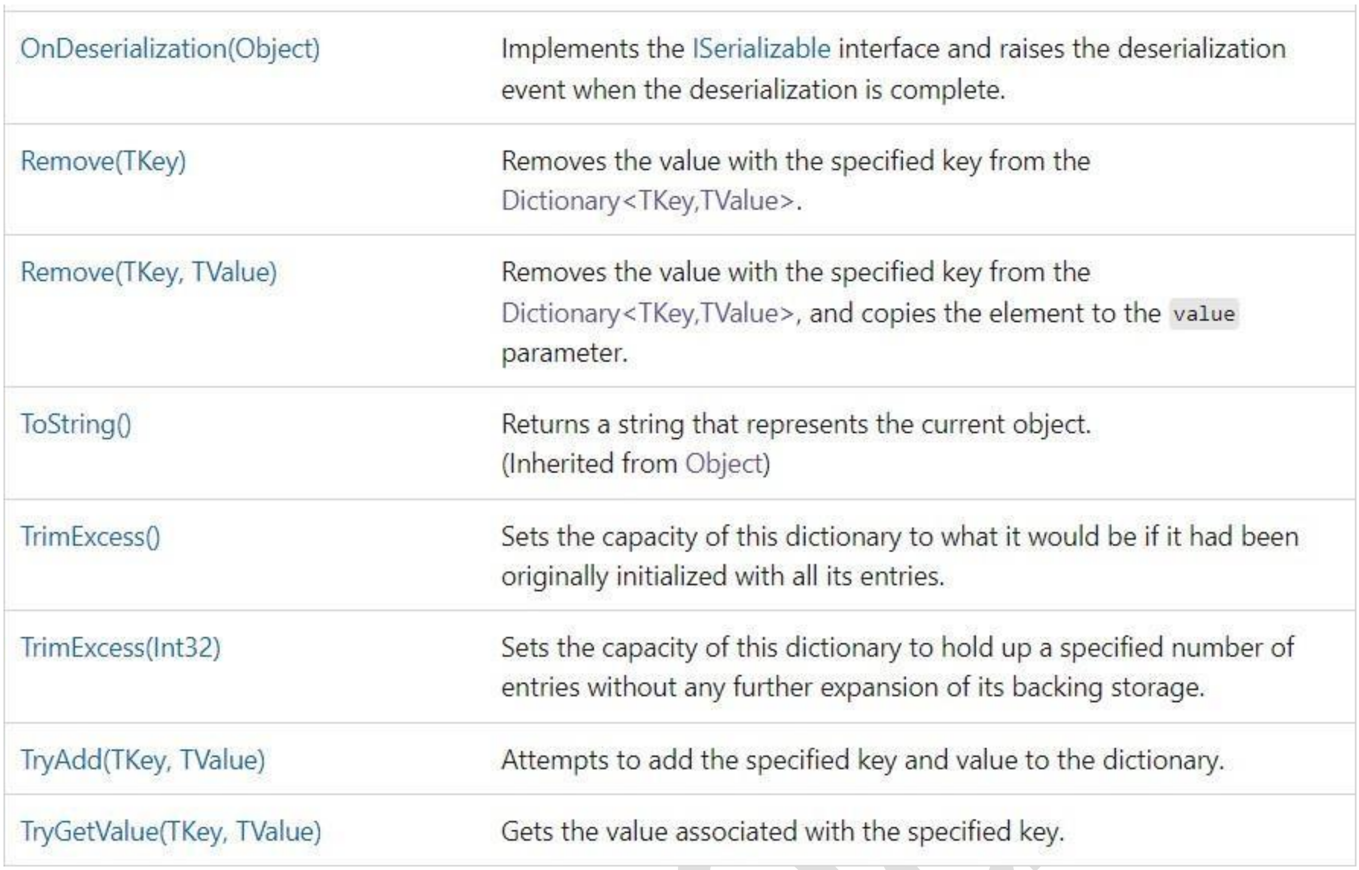
Dictionary<TKey,TValue>()	Initializes a new instance of the Dictionary<TKey,TValue> class that is empty, has the default initial capacity, and uses the default equality comparer for the key type.
Dictionary<TKey,TValue> (IDictionary<TKey,TValue>)	Initializes a new instance of the Dictionary<TKey,TValue> class that contains elements copied from the specified IDictionary<TKey,TValue> and uses the default equality comparer for the key type.
Dictionary<TKey,TValue> (IDictionary<TKey,TValue>, IEquality Comparer<TKey>)	Initializes a new instance of the Dictionary<TKey,TValue> class that contains elements copied from the specified IDictionary<TKey,TValue> and uses the specified IEqualityComparer<T>.
Dictionary<TKey,TValue>(IEnumerable<Key ValuePair<TKey,TValue>>)	Initializes a new instance of the Dictionary<TKey,TValue> class that contains elements copied from the specified IEnumerable<T>.
Dictionary<TKey,TValue>(IEnumerable<Key ValuePair<TKey,TValue>>, IEquality Comparer<TKey>)	Initializes a new instance of the Dictionary<TKey,TValue> class that contains elements copied from the specified IEnumerable<T> and uses the specified IEqualityComparer<T>.
Dictionary<TKey,TValue>(IEquality Comparer<TKey>)	Initializes a new instance of the Dictionary<TKey,TValue> class that is empty, has the default initial capacity, and uses the specified IEqualityComparer<T>.
Dictionary<TKey,TValue>(Int32)	Initializes a new instance of the Dictionary<TKey,TValue> class that is empty, has the specified initial capacity, and uses the default equality comparer for the key type.
Dictionary<TKey,TValue>(Int32, IEquality Comparer<TKey>)	Initializes a new instance of the Dictionary<TKey,TValue> class that is empty, has the specified initial capacity, and uses the specified IEqualityComparer<T>.
Dictionary<TKey,TValue>(SerializationInfo, StreamingContext)	**Obsolete.** Initializes a new instance of the Dictionary<TKey,TValue> class with serialized data.

Properties

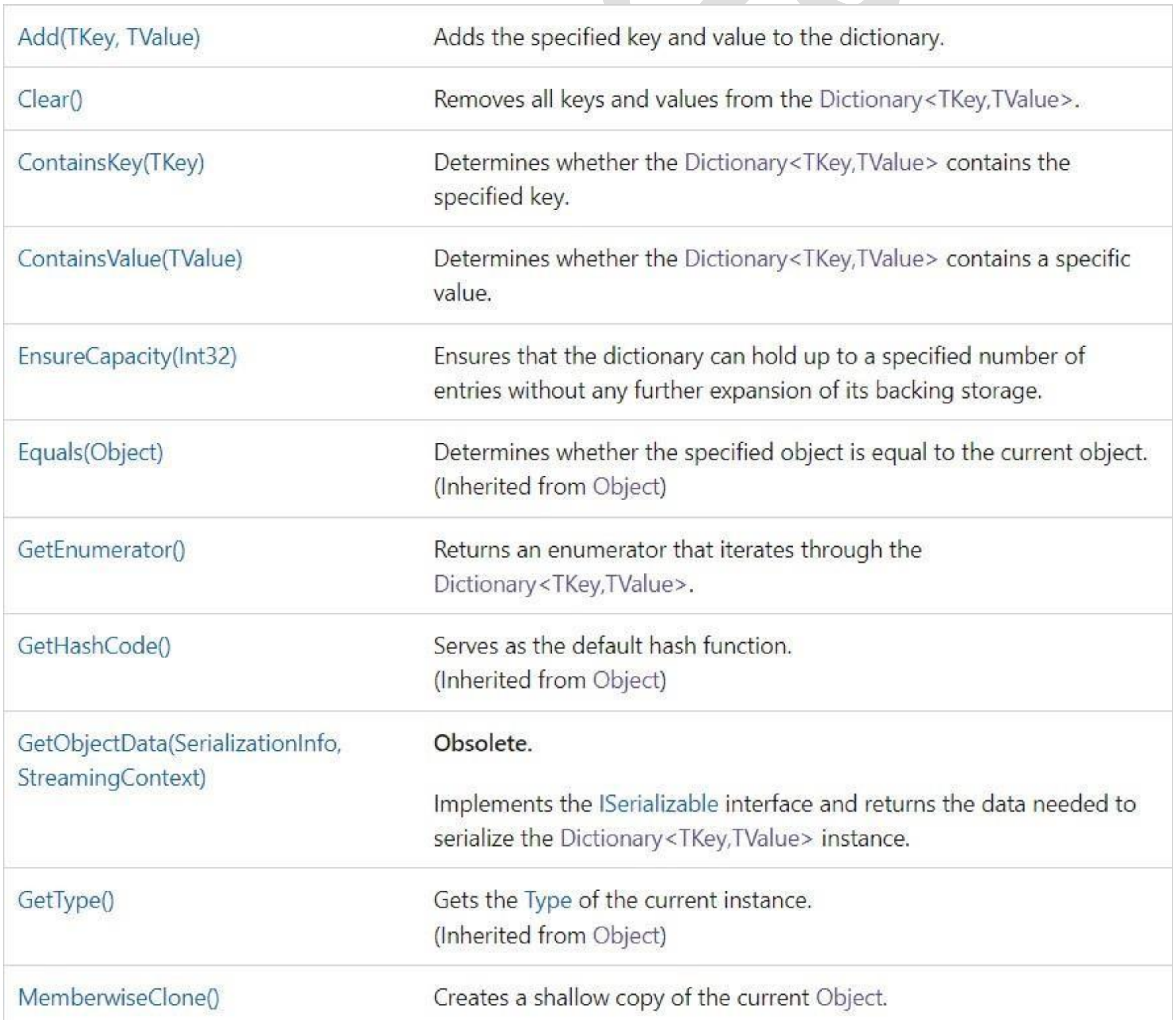


Comparer	Gets the IEqualityComparer<T> that is used to determine equality of keys for the dictionary.
Count	Gets the number of key/value pairs contained in the Dictionary<TKey,TValue>.
Item[TKey]	Gets or sets the value associated with the specified key.
Keys	Gets a collection containing the keys in the Dictionary<TKey,TValue>.
Values	Gets a collection containing the values in the Dictionary<TKey,TValue>.

Methods



OnDeserialization(Object)	Implements the ISerializable interface and raises the deserialization event when the deserialization is complete.
Remove(TKey)	Removes the value with the specified key from the Dictionary<TKey,TValue>.
Remove(TKey, TValue)	Removes the value with the specified key from the Dictionary<TKey,TValue>, and copies the element to the `value` parameter.
ToString()	Returns a string that represents the current object. (Inherited from Object)
TrimExcess()	Sets the capacity of this dictionary to what it would be if it had been originally initialized with all its entries.
TrimExcess(Int32)	Sets the capacity of this dictionary to hold up a specified number of entries without any further expansion of its backing storage.
TryAdd(TKey, TValue)	Attempts to add the specified key and value to the dictionary.
TryGetValue(TKey, TValue)	Gets the value associated with the specified key.



Add(TKey, TValue)	Adds the specified key and value to the dictionary.
Clear()	Removes all keys and values from the Dictionary<TKey,TValue>.
ContainsKey(TKey)	Determines whether the Dictionary<TKey,TValue> contains the specified key.
ContainsValue(TValue)	Determines whether the Dictionary<TKey,TValue> contains a specific value.
EnsureCapacity(Int32)	Ensures that the dictionary can hold up to a specified number of entries without any further expansion of its backing storage.
Equals(Object)	Determines whether the specified object is equal to the current object. (Inherited from Object)
GetEnumerator()	Returns an enumerator that iterates through the Dictionary<TKey,TValue>.
GetHashCode()	Serves as the default hash function. (Inherited from Object)
GetObjectData(SerializationInfo, StreamingContext)	**Obsolete.** Implements the ISerializable interface and returns the data needed to serialize the Dictionary<TKey,TValue> instance.
GetType()	Gets the Type of the current instance. (Inherited from Object)
MemberwiseClone()	Creates a shallow copy of the current Object.

(Inherited from Object)

Reflection :

Reflection objects are used for obtaining type information at runtime.
The classes that give access to the metadata of a running program are
in the System.Reflection namespace.

- Reflection has the following applications :
- It allows view attribute information at runtime.
- It allows examining various types in an assembly and instantiate these types.
- It allows late binding to methods and properties
- It allows creating new types at runtime and then performs some tasks using those types.

The MemberInfo object of the System.Reflection class needs to be initialized for discovering the attributes associated with a class.

```
System.Reflection.MemberInfo info = typeof(MyClass);
```

Reflection Attributes :

Attributes provide a powerful method of associating metadata, or declarative information, with code (assemblies, types, methods, properties, and so forth). After an attribute is associated with a program entity, the attribute can be queried at run time by using a technique called reflection.

Creating Custom Attributes :
Custom attributes are essentially traditional classes that derive directly or indirectly from System.Attribute

- Attributes are metadata extensions that give additional information to the compiler about the elements in the program code at runtime.
- Attributes are used to impose conditions or to increase the efficiency of a piece of code
- The primary steps to properly design custom attribute classes :
  1. Applying the AttributeUsageAttribute
  2. Declaring the attribute class
  3. Declaring constructors
  4. Declaring properties

1. Applying the AttributeUsageAttribute :

Finally, the class YourClass is inherited from the base class MyClass. The method MyMethod shows MyAttribute but not YourAttribute

```
public class YourClass : MyClass
{
// MyMethod will have MyAttribute but not YourAttribute.
public override void MyMethod()
{
//...
}
}
```

2. Declaring the attribute class :

```
[AttributeUsage(AttributeTargets.Method)]
public class MyAttribute : Attribute
{
// . . .
}
```

3. Declaring constructors :

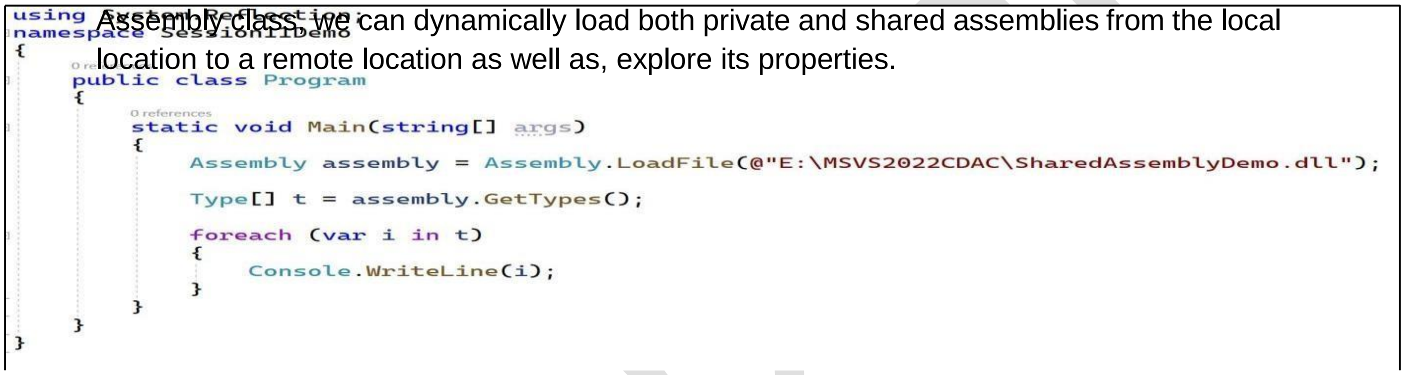
```
public MyAttribute(bool myvalue)
{
this.myvalue = myvalue;
```

}

4. Declaring properties :
public bool MyProperty
{
get {return
this.myvalue;} set
{this.myvalue =
value;}
}

Loading Assembly at Runtime :

The act of loading external assemblies on demand is known as Dynamic Loading. Using the Assembly class, we can dynamically load both private and shared assemblies from the local location to a remote location as well as, explore its properties.



```
using System.Reflection;
namespace Session11Demo
{
    0 references
    public class Program
    {
        0 references
        static void Main(string[] args)
        {
            Assembly assembly = Assembly.LoadFile(@"E:\MSVS2022CDAC\SharedAssemblyDemo.dll");

            Type[] t = assembly.GetTypes();

            foreach (var i in t)
            {
                Console.WriteLine(i);
            }
        }
    }
}
```

Steps to create and use Shared Assembly:

Step 1 : Open VS.NET and Create a new Class Library

Step 2 : Generating Cryptographic Key Pair using the tool SN.Exe

- Make sure that you start Administrator “Developer Command Prompt for VS 2022”
- Write the following command on command prompt C:\> sn -k "C:\mynewkey.snk“

Step 3 : Sign the component with the key and build the class library project (Go to properties of the project in solution explorer -> select signing -> Check the checkbox of Sign the assembly and browse for the key).

Step 4 : Host the signed assembly in Global Assembly Cache
C:\>gacutil -i "E:\MyClassLibrary\bin\Debug\MyClassLibrary.dll"

Step 5 : Test the assembly by creating the client application. Just add the project reference and browse for the shared assembly recently created.

Step 6 : Execute the client program